



Application Development Manual

SpaceChain OS

SPC002008 V1.00 Date: 2018/06/13

Product Manual

类别 Type	内容 Contents
Key Word	SpaceChain OS Brief history Functions
Abstract	Brief history of the operating system

Revision History

Version	Date	Reason
V1.00	2018/06/19	Create file

Preface

Brief introduction

The book describes SylixOS program design interface, including: SylixOS system API function, POSIX standard API function and many functions provided by standard C, and is applicable to all programmers.

As an advanced real-time embedded Operation System, SylixOS has been widely applied in aeronautics and astronautics, industrial automation, communications, new energy and other fields. Similar to many operating systems (such as VxWorks, Linux and so on), SylixOS provides massive service for program execution, such as: open file, read and write file, close file, dynamic loader, dynamic distribution of memory space, dynamic file creation, obtain system time and other services. Through the command under SylixOS Shell, one can conveniently view the system information. For example: view the running thread in the system via **ts** command, view the running process in the system via **ps** command, and view memory status via **free** command. How to use the command built in SylixOS Shell will be introduced in brief introduction to Shell in Chapter 3 of the book.

SylixOS is an open source operating system. Therefore, one can conveniently obtain the source code (obtain SylixOS source code from www.sylixos.com), learn the knowledge in the book through SylixOS source code, and verify the knowledge in each section with the instance in the book step by step.

This book will explain the programming method of the real-time system from the perspective of real-time system and precautions during programming.

Overview of this book

This book describes in detail the application programming method of SylixOS and use of application program interface thereof. The organization structure of this book is as follows:

- Chapter 1 describes the history of SylixOS and its application in various fields, and also describes POSIX standard of SylixOS;
- Chapter 2 describes the introduction and building of SylixOS development environment;
- Chapters 3 and 4 describe use of SylixOS Shell command and how to write the first SylixOS application;
- Chapter 5 provides an in-depth analysis of I/O system of SylixOS, and details the standard functions commonly used in I/O operations. These functions include unbuffered I/O functions, file and directory operation functions, and buffered I/O function and I/O multiplexed functions;

- describes the multi-thread programming method of SylixOS and its thread scheduling principles;
- Chapter 7 describes the inter-thread communication mechanism of SylixOS and how to use the lock in the thread.
- Chapter 8 describes the multi-process programming method of SylixOS and its process principle;
- Chapter 9 describes the inter-process communication mechanism of SylixOS.
- Chapter 10 describes the signal system of SylixOS and how to correctly use the signal during programming;
- Chapter 11 describes how to use SylixOS time management functions.
- Chapter 12 describes the principle of fixed-length memory and variable-length memory of SylixOS and its virtual memory principle, and how to correctly use these memories;
- Chapter 13 describes the standard I/O device operation of SylixO;
- Chapter 14 describes the principle of the hot swapping system and how to use the API;
- Chapter 15 describes the network programming method of SylixOS and how to use its network tools;
- Chapter 16 describes the file system principle of SylixOS.
- Chapter 17 analyzes the log system of SylixOS in detail;
- Chapter 18 describes the multi-user management of SylixOS;
- Chapter 19 describes the dynamic loading principle of SylixOS and how to use its application program interface;
- Chapter 20 describes how to the power management functions of SylixOS;
- Chapter 21 describes the programming method for SylixOS graphical interface Qt and how to transplant the third-party library to SylixOS;
- Chapter 22 describes how the application is migrated from the Linux platform or the VxWorks platform to the SylixOS platform.
- Finally, the appendix lists the standard header files in SylixOS and wrong numbers and their meanings in SylixOS. Appendix C details the SylixOS Makefile files.

If the reader has experience in Linux or VxWorks system programming, it will be easy to understand the knowledge in the book. Of course, you can also easily learn the

knowledge in the book without these experience, because this book contains a large number of instances, which are easy to understand (requiring C language program design basis). By learning this book, readers can quickly understand SylixOS, and be able to start developing their own SylixOS application.

Acknowledgment

This book can be successfully completed thanks to colleagues who have spent a lot of time and energy reviewing and writing. At the same time, we also want to thank enthusiastic netizens for their suggestions for revision of this book. Due to the limited level of writing staff, there will inevitably be some inadequacies in the book. Welcome readers to offer criticism and revision suggestions.

Beijing Acoinfo Technology Co., Ltd.

Contents

Noun explanation and agreement	11
Chapter 1 Brief introduction to SylixOS operating system	13
1.1 Brief history of the operating system	13
1.2 Functions of the operating system	14
1.3 Classification of operating systems	15
1.4 Brief introduction to POSIX Standards	16
1.5 POSIX restrictions	17
1.6 SylixOS Overview	21
1.7 SylixOS application field	23
1.8 Current application cases of SylixOS	24
Chapter 2 Integrated development environment	26
2.1 Introduction to ARM processor	26
2.1.1 Brief introduction	26
2.1.2 Features	26
2.1.3 Operating mode	27
2.1.4 Register organization	28
2.1.5 Instruction structure	30
2.2 Introduction to RealEvo-IDE	30
2.2.1 Brief introduction	30
2.2.2 Functions	32
2.3 Introduction to GCC toolchain	33
2.4 Installation of RealEvo-IDE	33
2.4.1 Acquisition of SylixOS development kit	33
2.4.2 Installation of SylixOS development kit	33
Chapter 3 Brief introduction to Shell	35
3.1 What's Shell	35
3.2 Instructions for common Shell commands	35
3.2.1 System command	36
3.2.2 File command	41
3.2.3 Network command	44
3.2.4 Time command	47
3.2.5 Dynamic loading command	49
3.2.6 Other commands	50
3.3 Environment variable	51
3.4 Root file system	52
3.5 Operation application	55
3.6 I/O redirection	55
Chapter 4 Compiling the first program	57
4.1 Hello world application	57
4.1.1 Create SylixOS.base project	57

4.1.2 Create the Hello World Project	60
4.1.3 Compiling the Hello world project	62
4.1.4 Deployment file	63
4.1.5 Run the Hello world application	64
4.1.6 Debug the Hello world application	65
4.1.7 Non-stop debugging mode.....	72
4.2 Hello Library	73
4.2.1 Create Hello Library Project.....	74
4.2.2 Compile Hello Library Project	74
4.2.3 Deploy the library file.....	75
4.2.4 Modify Hello World application.....	75
4.2.5 Run the Hello world Application	77
4.2.6 Debug Hello world applications and dynamic library	77
Chapter 5 I/O System	79
5.1 I/O System.....	79
5.1.1 File type.....	79
5.1.2 File descriptor.....	81
5.1.3 I/O System structure	82
5.2 Standard I/O access	85
5.2.1 File I/O.....	86
5.2.2 Files and directories.....	104
5.2.3 Standard I/O library.....	120
5.3 Asynchronous I/O access.....	132
5.3.1 POSIX asynchronous I/O.....	133
5.4 Advanced I/O access.....	141
5.4.1 Decentralized aggregation operation.....	141
5.4.2 Non-blocking I/O	144
5.4.3 I/O multiplexing	144
5.4.4 File record lock.....	150
5.4.5 File memory mapping.....	154
Chapter 6 Thread management.....	155
6.1 Thread	155
6.2 Thread state machine.....	155
6.3 SylixOS thread.....	157
6.3.1 Thread creation	157
6.3.2 Thread control	164
6.3.3 End of thread.....	167
6.3.4 Multi-thread security.....	168
6.4 POSIX thread	173
6.4.1 Thread attribute	173
6.4.2 Thread creation	179
6.4.3 Thread exit	181
6.4.4 Thread cancel.....	184

6.5 POSIX thread key value	192
6.6 SylixOS thread scheduling	195
6.6.1 Priority scheduling	195
6.6.2 RR (Round-Robin) scheduling	197
6.7 POSIX thread scheduling	199
6.8 SylixOS RMS scheduling	203
6.9 SylixOS coroutine	208
Chapter 7 Inter-thread communication	213
7.1 Shared resource	213
7.2 Inter-thread communication	214
7.3 SylixOS semaphore	215
7.3.1 Binary semaphore	216
7.3.2 Counting semaphore	225
7.3.3 Mutex semaphore	232
7.3.4 Read-write semaphore	238
7.4 POSIX semaphore	241
7.4.1 POSIX anonymous semaphore	241
7.5 Priority inversion	246
7.5.1 What's priority inversion	246
7.5.2 How to solve priority inversion	247
7.6 POSIX mutex semaphore	247
7.6.1 Mutex semaphore attribute block	248
7.6.2 Mutex semaphore	251
7.7 Deadlock	257
7.7.1 What's deadlock	257
7.7.2 Generation conditions of deadlock	257
7.7.3 Deadlock prevention	258
7.8 POSIX read-write lock	260
7.8.1 Read-write lock attribute block ^①	261
7.8.2 Read-write lock	262
7.9 SylixOS condition variable	266
7.9.1 Condition variable attribute block	268
7.9.2 Condition variable	269
7.10 POSIX condition variable	273
7.10.1 Condition variable attribute block	274
7.10.2 Condition variable	275
7.11 SylixOS message queue	279
7.11.1 Message queue	280
7.12 SylixOS event set	289
7.12.1 Event set	290
7.13 POSIX thread barrier	296
7.13.1 Thread barrier attribute block	297

7.13.2 Thread barrier.....	298
7.14 POSIX spin lock.....	300
7.14.1 Spin lock.....	302
7.15 SylixOS atomic number.....	305
7.15.1 Atomic number.....	306
7.16 One-time initialization.....	310
7.16.1 pthread_once_t variable.....	310
Chapter 8 Process management.....	314
8.1 Real-time process.....	314
8.2 Process state machine.....	315
8.3 POSIX processAPI.....	315
8.3.1 Execute program.....	316
8.3.2 Create process.....	319
8.3.3 Process scheduling.....	328
8.3.4 Process relation.....	333
8.3.5 Process control.....	338
8.3.6 Process environment.....	347
8.4 SylixOS process API.....	349
8.4.1 Create processes by using SylixOS API.....	349
8.4.2 SylixOS process control API.....	351
Chapter 9 Inter-Process Communication.....	356
9.1 Definition of IPC.....	356
9.2 Anonymous Pipe.....	356
9.2.1 Operate Anonymous Pipe.....	357
9.3 Named Pipe.....	360
9.3.1 Operate Named Pipe.....	360
9.4 POSIX Named Semaphore.....	363
9.4.1 Named Semaphore.....	364
9.5 POSIX Named Message Queue.....	368
9.5.1 Attribute Block of Named Message Queue.....	368
9.5.2 Named Message Queue.....	369
9.6 POSIX Shared Memory.....	376
9.7 XSI IPC.....	377
9.7.1 XSI Identifiers and Keys.....	377
9.7.2 XSI Permission Structure.....	378
9.7.3 XSI IPC Semaphore.....	380
9.7.4 XSI IPC Message Queue.....	382
9.7.5 XSI IPC Shared Memory.....	385
Chapter 10 Signal System.....	389
10.1 Signal System.....	389
10.1.1 Unreliable Signals and Reliable Signals.....	392
10.2 Signal Installation.....	393
10.2.1 Function signal.....	393

10.2.2 Function sigaction	393
10.3 Signal Set	398
10.4 Signal Transmission	403
10.4.1 非排队信号.....	错误!未定义书签。
10.4.1 Non-queued Signal	404
10.4.2 Queued Signal	405
10.4.3 Timer Signal	407
10.5 Signal Blocking	417
10.6 Process and Signal	420
10.7 Influence of Signal	421
10.7.1 System Call Interrupt ^①	421
10.7.2 Reentrancy Effect of Function.....	422
Chapter 11 Time Management	426
11.1 SylixOS Time Management	426
11.1.1 System Time.....	426
11.1.2 RTC Time.....	427
11.2 POSIX Time Management.....	429
11.2.1 UTC Time and Local Time.....	429
11.2.2 Time Form Transformation	431
11.2.3 High-precision Time.....	435
11.2.4 Get Process or Thread Clock Source	437
11.2.5 Time-related Extension Operations.....	437
Chapter 12 Memory Management.....	440
12.1 Fixed Length Memory Management	440
12.1.1 Create Memory Partition	440
12.1.2 Delete Memory Partition	441
12.1.3 Get/return Memory Block	442
12.1.4 Get Current State of Memory Partition	442
12.1.5 Get Memory Partition Name	443
12.2 Variable Length Memory Management	447
12.2.1 Create Memory Area	447
12.2.2 Delete Memory Area	447
12.2.3 Memory Area Increases Memory Space.....	448
12.2.4 Memory Allocation.....	448
12.2.5 Allocate Memory of Address Aligned	449
12.2.6 Dynamic Memory Adjustment	450
12.2.7 Free the Memory.....	451
12.2.8 Get Current State of Memory Area	451
12.2.9 Get Memory Area Name	452
12.3 POSIX Standard Memory Management	455
12.3.1 Memory Allocation.....	455
12.3.2 Allocate Memory Specifying Alignment Value.....	457

12.3.3 Free the Memory	458
12.3.4 Memory Allocation Function with Security Detection	458
12.4 Virtual Memory Management	459
12.4.1 Memory Division	459
12.4.2 Process Page Management	460
12.4.3 Virtual Memory Mapping	460
12.4.4 Other Operations of Virtual Memory	472
Chapter 13 Standard I/O Devices	474
13.1 /dev/null	474
13.2 /dev/zero	474
13.3 Terminal	474
13.4 Virtual Terminal	477
13.5 Graphic Device	477
13.6 Input Device	483
13.6.1 Mouse Device	483
13.6.2 Keyboard Device	487
13.7 Memory Device	490
13.8 Random Device	492
13.9 Audio Device	493
13.9.1 Basics	493
13.9.2 Audio Programming	494
13.9.3 Mixer Programming	497
13.10 Audio Device	500
13.10.1 Device Description	500
13.10.2 Description of Device Channels	501
13.10.3 Description of Image Format of the Device Channel	502
13.10.4 Setting of Device Channel	504
13.10.5 Setting of Device Buffer	505
13.10.6 Video Capture Control	507
13.10.7 Summary of Operation Commands for the Video Device	508
13.10.8 Video Device Application Examples	510
13.11 Real-Time Clock Device	513
13.12 GPIO Devices	513
13.13 CAN Bus Device	517
13.14 Virtual Device Files	523
13.14.1 eventfd	523
13.14.2 timerfd	526
13.14.3 hstimerfd	531
13.14.4 signalfd	532
Chapter 14 Hot-plug System	537
14.1 Introduction of the Hot-plug System	537
14.2 Hot-plug Message	538
14.2.1 Format of Hot-Plug Messages	538

14.2.2 Processing the Hot-Plug Messages.....	538
Chapter 15 Network I/O	543
15.1 socket interface	543
15.1.1 Network endian	544
15.1.2 Socket address	546
15.1.3 Socket function.....	551
15.1.4 Socket option	553
15.2 Brief introduction to TCP/IP	558
15.2.1 Layering of TCP/IP	558
15.2.2 IP address	561
15.2.3 Data encapsulation	562
15.2.4 Data demultiplexing.....	563
15.2.5 Port number.....	564
15.2.6 Link layer	564
15.2.7 IP Internet protocol.....	568
15.2.8 ARP address desorption protocol	570
15.2.9 ICMP message control protocol.....	571
15.2.10 UDP user datagram protocol	575
15.2.11 TCP transmission control protocol	575
15.3 Network communication instance	580
15.3.1 UDP instance	580
15.3.2 TCP instance.....	588
15.3.3 Raw socket (RAW) instance	597
15.4 Introduction to DNS	601
15.5 AF_UNIX domain protocol.....	612
15.5.1 AF_UNIX instance.....	613
15.6 AF_PACKET link layer communication	622
15.6.1 AF_PACKET instance	624
15.6.2 AF_PACKET and mmap.....	627
15.7 Network event detection.....	634
15.8 Standard network function library.....	638
15.8.1 ifconfig tool	638
15.8.2 TFTP	639
15.8.3 FTP.....	641
15.8.4 Telnet.....	647
15.8.5 ping.....	650
15.8.6 PPP	652
15.8.7 Network address translation	656
15.8.8 SylixOS network routing.....	659
15.8.9 netstat.....	676
15.8.10 npf	681
15.9 Control interface of the standard network card	686
15.10 Brief introduction to wireless communications and ad-hoc network.....	694

Chapter 16 File system	700
16.1 Introduction of the file system	700
16.2 TPSFS file system	702
16.3 FAT file system	704
16.3.1 FAT command	704
16.4 NFS file system	705
16.4.1 Basic operations of NFS	705
16.5 ROM file system	708
16.6 RAM file system.....	709
16.7 ROOT file system	710
16.8 PROC file system	712
16.8.1 /proc/pid process related information.....	713
16.8.2 /proc/ksymbol kernel symbol table.....	715
16.8.3 /proc/posix POSIX subsystem information	716
16.8.4 /proc/net network subsystem	716
16.8.5 /proc/power power management subsystem.....	717
16.8.6 Subsystem of the /proc/fs file system	719
16.8.7 /proc/version kernel version information	720
16.8.8 /proc/kernel kernel information.....	720
16.8.9 /proc/cpuinfo processor information.....	722
16.8.10 /proc/bspmem memory mapping information	723
16.8.11 proc/self auxiliary information.....	723
16.8.12 /proc/yaffs YAFFS partition information	724
16.9 YAFFS file system	726
16.9.1 The difference between NAND Flash and NOR Flash	726
16.9.2 YAFFS proper nouns.....	727
16.9.3 Memory Technology Device (MTD)	728
16.9.4 YAFFS partition	729
16.9.5 YAFFS command	731
16.10 File system Shell command	733
Chapter 17 Logging System	750
17.1 SylixOS logging system.....	750
17.2 POSIX logging system	752
Chapter 18 Multi-user Management.....	760
18.1 Introduction of POSIX User Management.....	760
18.1.1 Users	760
18.1.2 User Groups.....	761
18.2 Management of POSIX Authority	762
18.2.1 File Authority and Expression Method	763
18.2.2 File Authority Management Command chmod	764
18.3 User Management-related Files in the /etc Directory	765
18.3.1 /etc/passwd File	765
18.3.2 /etc/shadow File	767

18.3.3 /etc/group File	769
18.4 POSIX User Operations	770
18.4.1 User Password Operation.....	770
18.4.2 User Shadow Password Operation.....	772
18.4.3 User Group Operation.....	773
18.4.4 User's Additional Group Operation	774
18.5 Multi-user Management Database	776
18.5.1 User Operation.....	776
18.5.2 Group Operation	777
18.5.3 Password Operation.....	778
18.5.4 User Shell Commands	778
Chapter 19 Dynamic Loading	782
19.1 Principle of Dynamic Link Library	782
19.1.1 Format of ELF File	782
19.1.2 ELF Files in SylixOS	782
19.1.3 SylixOS Dynamic Loader Features.....	783
19.2 Autoloading of Dynamic Library	783
19.2.1 Linking of Dynamic Library.....	783
19.2.2 Downloading of Dynamic Library	784
19.2.3 Loading of Kernel Module	785
19.3 POSIX Dynamic Link Library API	785
19.3.1 Common API of Dynamic Library.....	785
19.3.2 Other APIs	788
19.4 Dynamic Link Library Shell Command.....	790
19.4.1 Viewing Dynamic Link Libraries	790
19.4.2 Loading Kernel Modules	791
19.4.3 Unloading Kernel Modules.....	791
Chapter 20 Power Management.....	794
20.1 SylixOS Power Management	794
20.2 Power Management API.....	795
Chapter 21 Introduction to Standard Third-Party Software	798
21.1 Qt Graphical Interface Software	798
21.1.1 Qt Porting in SylixOS	798
21.1.2 Verification of Qt Graphical Interface Library.....	800
21.1.3 QtCreator Installation and Configuration	802
21.2 Zlib File Compression Library.....	808
21.2.1 Porting of Zlib Library	808
21.2.2 Verification of Zlib Library.....	811
21.3 SQLite3 Database	811
21.3.1 Porting of SQLite3 in SylixOS	811
21.3.2 Verification of SQLite3 Library	811
21.4 OpenSSL Encryption Library.....	812
21.4.1 Introduction to OpenSSL.....	812

21.4.2 Porting of OpenSSL Library	813
21.4.3 Verification of OpenSSL Library	815
21.5 GoAhead Web Server	816
21.5.1 Porting of GoAhead	816
21.5.2 GoAhead Verification	817
21.6 C-Language Interpreter	817
21.6.1 Picoc Porting	818
21.6.2 Picoc Usage Verification	818
Chapter 22 Platform Porting	819
22.1 From Linux to SylixOS	819
22.2 From VxWorks to SylixOS	820
22.2.1 Development of VxWorks Applications in RealEvo-IDE	820
Appendix A Standard Header File	827
A.1 C Standard Header File	827
A.2 POSIX Standard Header File	827
Annex B SylixOS Error Number	830
B.1 POSIX Error Number	830
B.2 IPC/Web Error Number	830
B.3 SylixOS Kernel Error Number	832
B.4 Thread Error Number	833
B.5 Message Queue Error Number	835
B.6 TIMEREror Number	835
B.7 Memory Operation Error Number	836
B.8 I/O System Error Number	837
B.9 Shell Operation Error Number	837
B.10 Other Error Numbers	837
Annex C Description of SylixOS Makefile	839
C.1 Description of SylixOS Makefile	839
C.1.1 Directory Structure of SylixOS	839
C.1.2 config.mk File	841
C.1.3 Makefile File	843
Annex D Description of SylixOS Open Source Community	851
References	852

Noun explanation and agreement

This book strives to introduce the application development technology of the SylixOS real-time operation system with simplified language and space. The following computer vocabulary will be frequently used in the book. The following explanations and conventions are made for the computer terminology and its acronyms used.

- **CPU:** i.e., central processing unit (CPU) is the computing core and control core of a computer. It is collectively referred to as the three core components of the computer together with internal memory and input / output device.
- **RISC:** reduced instruction set computer, which adopts superscalar and superpipelined structures; there are only a few dozens of instructions, but the parallel processing capability is greatly enhanced.
- **SMP:** i.e., symmetric multi-processing, referring to the CPU which collects a group of same instruction set on a computer. Various CPUs shares the memory subsystem and bus structure. It is usually called as multi-core CPU system.
- **AMP:** i.e., Asynchronous Multiprocessing, referring to collection of a set of CPUs with different instruction sets and different functions on a computer. They are usually linked in a loosely coupled organization, and are responsible for processing different data respectively.
- **Compiler:** it is the program which translates the advanced computer language program (C/C++ and so on) to the machine language (binary code).
- **Assembler:** it is the language which translates the assembly language to machine language.
- **Linker:** it is a program which links one or more target files generated by the compiler or assembler with the dependent library to form an executable file.
- **GNU:** GNU Project was publicly launched by Richard Stallman on September 27, 1983, and it is aimed to provide a completely free operating system. Richard Stallman created the Free Software Foundation in 1985 to provide the technical, legal, and financial support for the GNU Project. Linux, GCC, EMAC and other software are from or enter GNU Project.
- **GCC:** it is the abbreviation of GNU Compiler Collection. GCC referred in particular to C compiler released by GNU. Due to rapid development of GCC, it

has not been just a compiler, but a development tool chain which integrates the compiler, linker, debugger, object analysis and other functions.

- **Multi-task:** it refers to that the user can run multiple applications within the same time, and each application is called as a task. In the single CPU system structure, multiple tasks operate on a CPU alternatively. In the multi-CPU system structure, the tasks with the equal quantity can operate simultaneously.
- **Scheduler:** it is the core of the operating system. It is actually a memory-resident program. It constantly scans the thread queues, utilizes a specific algorithm to find out the thread which needs to run more than the thread currently occupying the CPU, and the use right of CPU is deprived from the previous thread and transferred to threads that need to run.
- **Embedded system:** the embedded system refers to a dedicated computer system based on computer technology and with strict requirements for software and hardware tailoring, function, reliability, cost, size and power consumption in a narrow sense. An embedded system is a kind of dedicated computer system as a part of the device or equipment. The generalized embedded system refers to all computer systems except the server and PC.
- **Preemptive system:** it refers to the system which immediately abandons the current task, and turns to handle the more important event if any.
- **Version management:** it is the foundation of software configuration management, which manages and protects the developer's software resources. The main functions include: centralized file management; software version upgrade management; locking function; provide comparison of different versions of source programs.
- **BUG tracking:** it is the foundation of software defect management. The main functions include: recording and saving the problem-solving process; recording and saving the process and basis of a design decision. It can effectively record the process from software defect discovery to correction.
- **BSP:** abbreviation of board support packet. It is the collection of underlying programs running on the hardware platform of the operating system, generally including: startup program, driver, interrupt service program and other basic programs.
- **TCM:** the tightly coupled memory is an RAM with fixed size, which is closely coupled to the processor kernel, and provides the performance equivalent to Cache. Compared with Cache, the advantage is that the program code can accurately control the position of the function or code (stored in RAM).

- Cross compilation: that is to say, generate the executable code on a platform on another platform. For example, the executable program on ARM platform can be developed on x86 platform.
- **Host machine:** the computer used for development during cross compilation.
- **Target machine:** sometimes referred to as the target system or device, it is the target computer of cross compilation. The computer or device is used to run the cross-compiled executable program.

Chapter I Brief introduction to SylixOS operating system

1.1 Brief history of the operating system

The operating system (OS for short) is a computer program which manages and controls the computer hardware and software resources. It is the most basic system software which directly runs on the "bare machine".

The operating system is the interface between the user and the computer, and also the interface between the computer hardware and other software. The functions of the operating system include managing hardware, software and data resources of the computer system, controlling program operation, improving the human-machine interface, and providing support for other application software. The operating system can maximize functions of the computer system resources.

There are many kinds of operating systems. From simple to complex, they can be divided into smart card operating system, real-time operating system, sensor node operating system, embedded operating systems, personal computer operating systems, multi-processor operating system, network operating system and mainframe operating system. There are mainly three types according to the application field: desktop operating system, server operating system and embedded operating system.

In the middle of the 20th century, humans entered the age of information with the birth of computers. At that time, the first-generation computers did not have an operating system, because the way to build the early personal computers (same with building the mechanical computer) and performance were insufficient to execute such program. In 1947, due to invention of transistors and the micro-programming method invented by Maurice Vincent Wilkes, the computer was no longer a mechanical device but an electronic product. System management tools and programs which simplify the hardware operation process quickly emerged, which became the foundation of the operating

system.

By the mid-1950s, the commercial computer manufacturers had created the batch processing system which can serialize establishment, scheduling and execution of the work. At that time, the computer manufacturer writes different operating systems for each different model of computer, so the program written for one computer cannot be transplanted to other computers for execution, even if it is of the same model.

In 1963, General Electric Company worked with Bell Labs and MIT to develop the Multics operating system in PL/I language. Its appearance was a source of inspiration for establishment of many operating systems in the 1970s. The UNIX system established by Dennis Ritchie and Ken Thompson from AT&T Bell Laboratories. The operating system was rewritten in C Language in 1969 in order to realize portability of the platform.

The UNIX operating system written in C Language is of inter-temporal significance. It is the first modern operating system in the real sense. Later-born systems, such as Linux, BSD, Mactonish and Solaris, all came from UNIX system in the aspects of principle and application program interface. The impact of UNIX system on operating systems continues to this day.

1.2 Functions of the operating system

The theoretical researchers of the operating system sometimes divide the operating system into four major parts:

- **Device driver:** the bottom part with direct control and monitoring of all types of hardware. Their responsibilities are to hide the hardware details and provide an abstract and generic interface to other parts.
- **Kernel:** the kernel part of the operating system, usually running at the highest privilege level, responsible for providing basic and structural functions.
- **Interface library:** it is a series of special program libraries. Their responsibilities are to wrap the basic services provided by the system into the programming interface (API) which can be used by the application, and it is the part closest to the application.
- **Periphery:** refers to all other parts of the operating system other than the above three types, and is usually used to provide specific advanced services. For example, in a microkernel structure, most system services and various daemons in UNIX/Linux are usually classified in this list.

The main functions of the operating system are resource management, program control, human-machine interaction and so on. The resources of the computer system can be divided into two major categories of device resources and information resources. Device resources refer to hardware devices constituting the computer, such as central

processors, main memory, magnetic disc memory, printer, tape memory, monitor, keyboard, mouse and other devices. Information resources refer to various data stored in the computer, such as files, program library, system software, application software and so on.

The operating system is located between the underlying hardware and application software or users, and is the bridge between the two. The application or the user can operate the computer via various interfaces provided by the operating system. A standard operating system shall provide the following functions:

- Task management.
- Memory Management.
- File System.
- Networking.
- Security.
- User Interface.
- Device Drivers.

1.3 Classification of operating systems

Due to differences in features and application fields, the operating systems can be divided into the following types:

Batch Processing Operating System: the user submits the job to the system operator. The system operator composes the jobs of multiple users into a batch of jobs, and then enters it into the computer to form a continuous job flow with automatic transfer in the system. Then start the operating system, and the system will automatically executes each job in turn. The batch processing operating system is divided into the simple batch processing system and the multi-channel batch processing system.

Time Sharing Operating System: multiple users can access the system through the terminal simultaneously. Since multiple users share the processor time, the technology is called Time Sharing. The user interactively submits commands to the system. In response to the request, the system accepts each user's commands and controls each user program to be executed alternately in a short time unit (this type of time unit is called as **time slice**). This technology makes each user think that they own the processor. The time-sharing system is characterized by multiplexing, interactivity, "exclusivity" and timeliness.

Real Time Operating System: referring to an operating system which enables the computer to respond to the request of an external event in a timely manner, complete processing of the event within a defined "strict time", and control all real-time devices and

real-time tasks to work in concert. The goal of the real-time operating system is to: respond to external requests within a "strict time" range, which has the characteristics of high reliability, integrity, resource allocation, and real-time task scheduling. Resource allocation and real-time task scheduling are its main features. In addition, the real-time operating system shall have strong fault tolerance. The real-time operating system is divided into the hard real-time operating system and the soft real-time operating system. The hard real-time operating system can guarantee that all real-time events can be properly responded within a certain period. The soft real-time operating system can only do its utmost to strive for the real-time event to get response within a certain time.

Distributed Software Systems: the operating system configured for distributed computing systems. A large number of computers are linked together via the network, so as to obtain extremely high calculation capability and extensive data sharing. This system is called as the distributed system. It differs from other operating systems in resource management, communication control and operating system structure. At the same time, the distributed operating system must support parallel processing. Therefore, the communication mechanism it provides is different from that provided by the network operating system. It requires a fast communication speed. The structure of the distributed operating system is also different from those of other operating systems. It is distributed on various computers in the system and can handle various demands of the users in parallel. It has strong fault tolerance.

The Embedded Operating System is an operating system which runs on an embedded device. The embedded operating system is an operating system widely used. The embedded device generally adopts the dedicated embedded operating systems. They are usually the real-time operating system, such as SylixOS, VxWorks, and some are functionally-reduced versions of Linux kernel operating systems, such as Android, Tizen, MeeGo and so on.

1.4 Brief introduction to POSIX Standards

Since the first modern operating system - UNIX was born in 1970, there have been a variety of modern operating systems, such as: Windows, Linux, BSD, Solaris, etc. In order to facilitate the transplantation of applications and middleware, most operating systems adopt UNIX-compatible API (except for Windows). The POSIX standard was established to ensure mutual compatibility of operating system API.

POSIX is a collective term for a series of interconnected standards defined by the IEEE (Institute of Electrical and Electronics Engineers) to regulate the API interfaces provided by various UNIX operating systems. It is officially called as IEEE1003, and the international standard name is ISO/IEC9945. This standard originated from a project which began approximately in 1985. The name of POSIX is an easy-to-remember name proposed by Richard Stallman at the request of the IEEE. It is basically the abbreviation of Portable Operating System Interface, and X indicates its inheritance of Unix API.

The POSIX standard defines a sub-protocol called as 1003.1b for real-time operating system. This protocol defines the basic behavior of the standard real-time operating system. SylixOS satisfies requirements of this protocol.

The current POSIX is mainly divided into four parts: Base Definitions, System Interfaces, Shell and Utilities and Rationale. SylixOS is compatible with most specifications in these four parts.

The current operating systems conforming to the POSIX standard protocol are: UNIX, BSD, Linux, iOS, Android, SylixOS, VxWorks, RTEMS, etc. Due to the support of POSIX by SylixOS, applications on other compatible POSIX systems can be easily transplanted to SylixOS operating system.

Table A.2 lists the POSIX standard header files supported by SylixOS.

1.5 POSIX restrictions

POSIX defines a number of constants which involve operating system implementation restrictions. Unfortunately, this is one of the most puzzling parts of POSIX. Although POSIX defines a lot of restrictions and constants, we only care about the parts related to the basic POSIX interface. These limits and constants are divided into the following 7 categories.

- Numeric restrictions: LONG_BIT, SSIZE_MAX and WORD_BIT;
- Minimum value: as shown in Table 1.1;
- Maximum value: _POSIX_CLOCKRES_MIN;
- Value which can be increased during operation: CHARCLASS_NAME_MAX, COLL_WEIGHTS_MAX, LINE_MAX, NGROUPS_MAX and RE_DUP_MAX;
- Invariant value; during operation;
- Other invariant values: NL_ARGMAX, NL_MSGMAX, NL_SETMAX and NL_TEXTMAX;
- Variable value of path name: FILESIZEBITS, LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, PIPE_BUF and SYMLINK_MAX.

Among these restrictions and constants, some may be defined in <limits.h>, and the rest may be defined and may not be defined according to specific conditions .

These minimum values are constant (they do not change with the system). They specify the most restrictive values of these features. A POSIX-compliant implementation shall provide such a large value at least. This is why they are called as the minimum values, even though their names all include MAX. In addition, in order to ensure portability, a strictly POSIX-compliant application shall not require a larger value.

Table 1.1 Minimum value of POSIX in <limits.h>

Name of the minimum value	Note
_POSIX_CHILD_MAX	Number of child processes per actual user ID
_POSIX_DELAYTIMER_MAX	Maximum number of timer overruns
_POSIX_HOST_NAME_MAX	Length of the host name returned by the gethostname function
_POSIX_LINK_MAX	Number of links of the file
_POSIX_LOGIN_NAME_MAX	Length of the login name
_POSIX_MAX_CANON	Number of bytes in terminal specification input queue
_POSIX_MAX_INPUT	Available space of the terminal input queue
_POSIX_NAME_MAX	Number of bytes in the file name, excluding terminating null bytes
_POSIX_NGROUPS_MAX	Number of group IDs added by each process at the same time
_POSIX_OPEN_MAX	Number of open files per process
_POSIX_PATH_MAX	Number of bytes in the path name, including terminating null bytes
_POSIX_PIPE_BUF	Number of bytes which can be atomically written to a pipe
_POSIX_RE_DUP_MAX	Times of repetitions of basic regular expression allowed by the regex and regcomp functions when the interval notation $\{m,n\}$ is used
_POSIX_RTSIG_MAX	Number of real-time signal numbers reserved for the application
_POSIX_SEM_NSEMS_MAX	Number of semaphores which can be used by a process simultaneously
_POSIX_SEM_VALUE_MAX	Value which can be held by the semaphore
_POSIX_SIGQUEUE_MAX	Number of queued signals which can be sent and suspended by a process
_POSIX_SSIZE_MAX	Value which can exist in the ssize_t object
_POSIX_STREAM_MAX	Number of standard I/O streams which can be opened by a process simultaneously
_POSIX_SYMLINK_MAX	Number of bytes in the symbolic link
_POSIX_SYMLOOP_MAX	Number of compliant links which can be traversed when parsing path names
_POSIX_TIMER_MAX	Number of timers per process
_POSIX_TTY_NAME_MAX	Length of the terminal device name, including the terminating null bytes
_POSIX_TZNAME_MAX	Time zone name bytes

A specific value may not be defined in this header file, because the actual value of a given process may depend on the total amount of storage in the system. If not defined in the header file, they cannot be used as numeric boundaries during compilation. Therefore, POSIX provides 3 runtime functions for call: sysconf, pathconf, and fpathconf. The actual implementation value can be obtained at runtime with these 3 functions..


```
#include <unistd.h>
long sysconf(int name);
long fpathconf(int fd, int name);
long pathconf(const char *path, int name);
```

Function `sysconf` prototype analysis:

- For success of the function, return the corresponding value. For failure, return -1 and set the error number;
- The parameter *name* is the request constant name, as shown in Table 1.2.

Function `fpathconf` prototype analysis:

- For success of the function, return the corresponding value. For failure, return -1 and set the error number;
- The parameter *fd* is the open file descriptor;

The parameter *name* is the request constant name, as shown in Table 1.3.

Function `pathconf` prototype analysis:

- For success of the function, return the corresponding value. For failure, return -1 and set the error number;
- The parameter *path* is the file path name;
- The parameter *name* is the request constant name, as shown in Table 1.3.

Table 1.2 Sysconf call (partial) *name* parameter

<i>Name</i> parameter	Return value
_SC_ARG_MAX	ARG_MAX
_SC_CLK_TCK	LW_TICK_HZ
_SC_DELAYTIMER_MAX	__ARCH_INT_MAX
_SC_IOV_MAX	__ARCH_LONG_MAX
_SC_LINE_MAX	LINE_MAX
_SC_LOGIN_NAME_MAX	LOGIN_NAME_MAX
_SC_OPEN_MAX	LW_CFG_MAX_FILES
_SC_PAGESIZE	PAGESIZE
_SC_RTSIG_MAX	RTSIG_MAX
_SC_SEM_NSEMS_MAX	SEM_NSEMS_MAX
_SC_SEM_VALUE_MAX	__ARCH_UINT_MAX
_SC_SIGQUEUE_MAX	SIGQUEUE_MAX
_SC_TIMER_MAX	LW_CFG_MAX_TIMERS
_SC_TZNAME_MAX	TZNAME_MAX

Table 1.3 Pathconf and fpathconf calls (partial) *name* parameters

<i>name</i> parameter	Return value
_PC_FILESIZEBITS	FILESIZEBITS
_PC_LINK_MAX	Internal definition of the system
_PC_MAX_CANON	MAX_CANON
_PC_MAX_INPUT	MAX_INPUT
_PC_NAME_MAX	NAME_MAX
_PC_PATH_MAX	PATH_MAX
_PC_PIPE_BUF	PIPE_BUF
_PC_SYMLINK_MAX	Internal definition of the system

The difference between the latter two functions is that: one takes the file descriptor as the parameter, and the other uses the path name as the parameter.

If *name* is not a correct constant, these 3 functions return to -1, and set errno to EINVAL. Some *name* will return a variable value or -1, -1 represents an indeterminate value. At the moment, errno value will not be changed.

1.6 SylixOS Overview

SylixOS is a large-scale embedded real-time operating system, which was born in 2006. At first, it was only a small multi-tasking scheduler. After years of development, SylixOS has become a stable and reliable embedded system software development platform with full functions and excellent performance.

Among the real-time operating system similar to SylixOS, VxWorks (mainly used in aeronautics, astronautics, military and industrial automation) and RTEMS (originated from the missile and rocket control real-time system of the US Department of Defense) are well-known in the world.

As a latecomer of the real-time operating system, SylixOS has borrowed many design ideas from many real-time operating systems, making SylixOS reach or exceed the level of many real-time operating systems in terms of functionality and specific performance, and making it one of the most best representatives in domestic real-time operating systems^①.

SylixOS, as a preemptive multi-task hard real-time operating system, has the following features and characteristics:

- Compatible with IEEE1003 (ISO/IEC9945) operating system interface specifications;

- 1003.1b (ISO/IEC 9945-1) real-time programming standards;
- Excellent real-time performance (task scheduling and switching, interrupt response algorithms are O(1) time complexity algorithm);
- Support unlimited multitasking;
- Preemptive scheduling supports 256 priorities;
- Support coroutines (called as fiber in windows);
- Support virtual process;
- Support priority inheritance to prevent priority inversion;
- Extremely stable kernel, many products based on SylixOS development require 7x24-hour continuous operation;
- The CPU occupancy rate of the kernel is low;
- Flexible system (Scalable);
- The core code is written in C Language, which has good portability;
- Support tightly-coupled homogeneous multi-processor (SMP), such as: ARM Cortex-A9 SMP Core;
- Unique hard real-time multi-core scheduling algorithm;
- Support standard I/O, multiple I/O multiplexing and asynchronous I/O interfaces;
- Support multiple emerging asynchronous event synchronization interfaces, such as signalfd, timerfd and eventfd;
- Support many standard file systems: TPSFS, FAT, YAFFS, RAMFS, NFS, ROMFS, etc;
- Support file record lock, and can support database;
- Support the uniform block device Cache model;
- Support memory management unit (MMU);
- Support third-party GUI graphic libraries, such as Qt, Microwindows and emWin;
- Support dynamic loading applications, dynamic link libraries and modules;
- Support extended system symbol interface;
- Support standard TCP/IPv4/IPv6 dual network protocol stack, and provide standard socket operation interface;
- Support AF_UNIX, AF_PACKET, AF_INET, AF_INET6 protocol domain;

- Integrated with many network tools, such as: FTP, TFTP, NAT, PING, TELNET, NFS, etc.

- Integrated with shell interface, support environment variables (basically compatible with Linux operation habits);
- Integrated with re-entry ISO/ANSI C library (supports over 80% of standard functions);
- Supports many standard device abstractions, such as TTY, BLOCK, DMA, ATA, GRAPH, RTC and PIPE. Meanwhile, support a variety of industrial equipment or bus models, such as: PCI, USB, CAN, I2C, SPI, SDIO, etc.;
- Provide high-speed timer device interface, and can provide timing service higher than the master clock frequency;
- Support hot swapping device;
- Support device power management;
- Kernel, drivers, and applications support GDB debugging;
- Provide kernel behavior tracker to facilitate application performance and failure analysis.

1.7 SylixOS application field

SylixOS adopts the preemptive, multitasking and hard real-time approach to design the whole operating system. The core target of its technical implementation is real-time control, stability and reliability. Therefore, SylixOS is suitable for (but not limited to) the following fields where the requirements for real-timeness and stability are particularly prominent:

- **Industrial real-time control:** mainly including industrial robot system, site security monitoring and protection system, industrial field bus communication management system and so on;
- **Aeronautics and astronautics:** mainly including aircraft flight control systems, aerospace data acquisition and recording system, high-precision surveying and mapping system, aerospace communication system and so on;
- **National defense security:** mainly including encryption communication system, sensor terminal system, virtual instrument system, data acquisition and recording system, fire control system and so on;
- **Financial terminal:** mainly including POS charging system, terminal payment system, ATM and so on;

**High-reliability civil use:**

mainly including vehicle traveling data recorder system, central control system for vehicles and marine engines, production line testing system, medical instrument system, distributed unattended system and so on.

1.8 Current application cases of SylixOS

Since 2006, many projects or products have been developed based on SylixOS, covering a wide range of fields, and the products have stable and reliable operation. The following describes some products based on SylixOS from industrial automation, military, communication, civil use and other fields. Most products require 7×24 hour uninterrupted operation, and many SylixOS system nodes have run over 50,000 hours without interruption.

1. Industrial automation

- Universal configuration development human-machine interface.
- Electrical fire alarm system.
- Toll-by-weight and overlimit detector.
- Special vehicle and marine engine state monitor.
- Power environment monitoring station.
- Access control system event server.
- Universal PLC system.

2. Communications

- A variety of industrial fieldbus protocol converters.
- Industrial high-reliability IP router.
- Coal mine wireless personnel positioning system.

3. New energy

- Small photovoltaic power generation real-time data manager.
- Large-scale photovoltaic power generation node manager.
- Super capacitor car balance charge and discharge controller.

4. Weapon system

- Conventional submarine battery monitoring system.
- Wheeled armored vehicle real-time monitoring system.

Chapter 2 Integrated development environment

This chapter introduces the functions and use of the integrated development environment of the SylixOS operating system. This book takes the most common ARM processor on the market as an example (similar to the development, debugging and deployment of x86, MIPS, PowerPC processor and ARM processor).

2.1 Introduction to ARM processor

2.1.1 Brief introduction

ARM (Advanced RISC Machines) can be deemed as a company's name, the generic name for ARM core microprocessor, or the name of a technology.

ARM was founded in Cambridge, UK in 1991, and mainly sells authorization of chip design technology. At present, the microprocessor adopting ARM technical intellectual property (IP) core (i.e., ARM processor mentioned normally) has been widely used in industrial control, consumer electronic product, communication system, network system, wireless system and various product markets. ARM-based microprocessor application occupies over 75% of market share of 32-bit RISC microprocessor, and ARM technology is gradually seeping into every aspect of daily life.

ARM processors can be divided into different series according to different kernels used. Division of these series is based on ARM7, ARM9, ARM10 and ARM11 kernels (suffix numbers of 7,9,10 and 11 indicate different kernel designs). Digital ascending indicates improvement in performance and complexity. In each series, there are also multiple changes in memory management, Cache and TCM processor extension.

2.1.2 Features

- Small volume, low power consumption, low cost and high performance;
- Support Thumb (16-bit) /ARM (32-bit) dual instruction sets, and be compatible with 8-bit / 16-bit device;
- Extensive use of registers, realizing rapid instruction execution;
- Most data operations are completed in the register;
- The addressing mode is flexible and simple with high execution efficiency;
- The instruction length is fixed.

2.1.3 Operating mode

ARM processor has 7 operating modes in total, as shown in Table 2.1.

Table 2.1 Operating mode of ARM processor

Operating mode	Instructions
User mode (User)	For execution of normal program
System mode (System)	Operate the operating system tasks with privilege
General interrupt mode (Irq)	For normal interrupt processing
Fast interrupt mode (Fiq)	For fast interrupt processing
Management mode (Supervisor)	Protected mode used for the operating system
Abort mode (Abort)	Enter the mode when data or instruction prefetch is terminated
Undefined mode (Undefined)	Enter the mode when the undefined instruction is executed

1. Privileged mode

Except for the user mode, other modes are privileged modes. ARM internal registers and some on-chip peripherals can only be accessed in the privileged mode on the hardware design. In addition, the privileged mode can be freely switched to the processor mode, and the user mode cannot be directly switched to other modes.

2. Exception mode

Except for the system mode, other 5 modes in the privileged modes are also collectively referred as exception modes. In addition to entry through program switching in privilege, they can enter it from specific abnormality. For example, enter the interrupt exception mode when the hardware generates the interrupt signal, enter the abort exception mode when the data without authority is read, and enter the abort exception mode of undefined instruction when the undefined instruction is executed. Where, the management mode is also called as the superuser mode, and the specific mode providing software interrupt for the operating system. It is precisely because of software interrupt that the user program can switch to the management mode through system call.

2.1.4 Register organization

There are 37×32-bit registers in ARM processor, including 31 general registers and 6 status registers. These registers cannot be accessed at the same time. Programmability and Accessibility of the register depends on working state and specific operation mode of the microprocessor. However, general registers R0 to R14 and the program counter and

one or two status registers can be accessed at any time.

1. General register

The general registers include R0 to R15, which can be divided into three categories:

- Unbanked register R0 to R7.
- Banked register R8 to R14.
- Program counter PC (R15).

In all operation modes, unbanked registers are pointed at the same physical register, and not used for special purpose by the system. Therefore, damage to the register data may be caused when conversion of operation modes is performed for interrupt or exception handling.

R8 to R12 registers respectively correspond to two different physical registers. Access registers of R8_fiq to R12_fiq in fiq mode, and access registers of R8_usr to R12_usr in other modes.

R13 and R14 registers respectively correspond to 6 different physical registers, one of the physical registers is shared by the user mode and the system mode, and other 5 physical registers correspond to other 5 different operation modes. R13_<mode> and R14_<mode> modes are adopted to identify different physical registers (modes are USR, FIQ, IRQ, SVC, ABT and UND).

R13 is often used as the stack pointer in ARM instruction, but it is only a idiomatic usage. The user can also use other registers as stack pointers. Each operation mode of the processor has its own independent physical register R13. Therefore, R13 in the corresponding mode shall be initialized in the initialization part. When program operation enters the exception mode, the register to be protected can be placed into the stack pointed by R13. However, when the program returns from the exception mode, it shall be recovered from the corresponding stack.

R14 is also called as the subroutine link register or the link register LR. When the call instruction of BL subroutine is executed, R15 (program counter PC) backup can be obtained from R14, and R14 can be used to save the return address of the subroutine in each operation mode.

The register R15 is called as the program counter (PC). Although R15 can be used as the general register, but it is not used in this way usually, because there are some special restrictions on use of R15, and execution results of the program are unknown if these restrictions are violated.

2. Status register

Status registers include the current program status register and the saved program status register. CPSR can be accessed in any operation mode, which includes the condition flag bit, interrupt disable bit, pattern flag bit of current processor, and other related control and status bits. There is a special physical status register SPSR in each exception mode, SPSR is used to save the current value of CPSR in case of any exception, and CPSR can be recovered by SPSR when quit from exception. Since the user mode and the system mode do not belong to the exception mode, they do not have SPSR, and results are unknown when SPSR is accessed in the two modes.

2.1.5 Instruction structure

ARM microprocessor supports two instruction sets in newer system structure: ARM instruction set and Thumb instruction set. Where, the length of ARM instruction set is 32 bits, the execution cycle is mostly single cycle, and the instructions are executed with conditions; The Thumb instruction can be seen as a subset of the ARM instruction set in the compressed form, it has 16-bit code density, while the execution efficiency is lower than that of ARM instruction. The Thumb instruction has the following features:

- The instruction execution condition will not be used frequently.
- The source register is same with the target register frequently.
- The number of registers used is relatively less.
- The value of the constant is relatively small.
- The Barrel Shifter in the kernel is not used frequently.

2.2 Introduction to RealEvo-IDE

2.2.1 Brief introduction

RealEvo-IDE is the dedicated integrated development environment for the SylixOS operating system. It can enable development of SylixOS operating system application, BSPs, driver and shared library to be simple and efficient. RealEvo-IDE includes the following parts:

- SylixOS Base project: The project includes libsylixos (high-performance SylixOS kernel), libVxWorks (VxWorks compatible library), libcextern (C extension library),

liblua, libluaplugin (Lua script support), libzmodem (zmodem protocol support) and libsqlite3 (SQLite database) and so on;

- Various project templates: BSP project template, application project template, shared library project template, kernel module project template, as shown in Figure 2.1;
- Development tools: automatic upload tools, RealEvo-Simulator, kernel behavior monitor and so on;
- Integrated development environment: It can help the use manage and build the project, compile corresponding code according to different hardware platforms, and can organize and manage communication, debugging and operation with SylixOS target system;
- Compiler: having powerful code editing function;
- Compiler: including ARM, MIPS, PowerPC, x86, c6x, SPARC, Lite and other platform compilers.

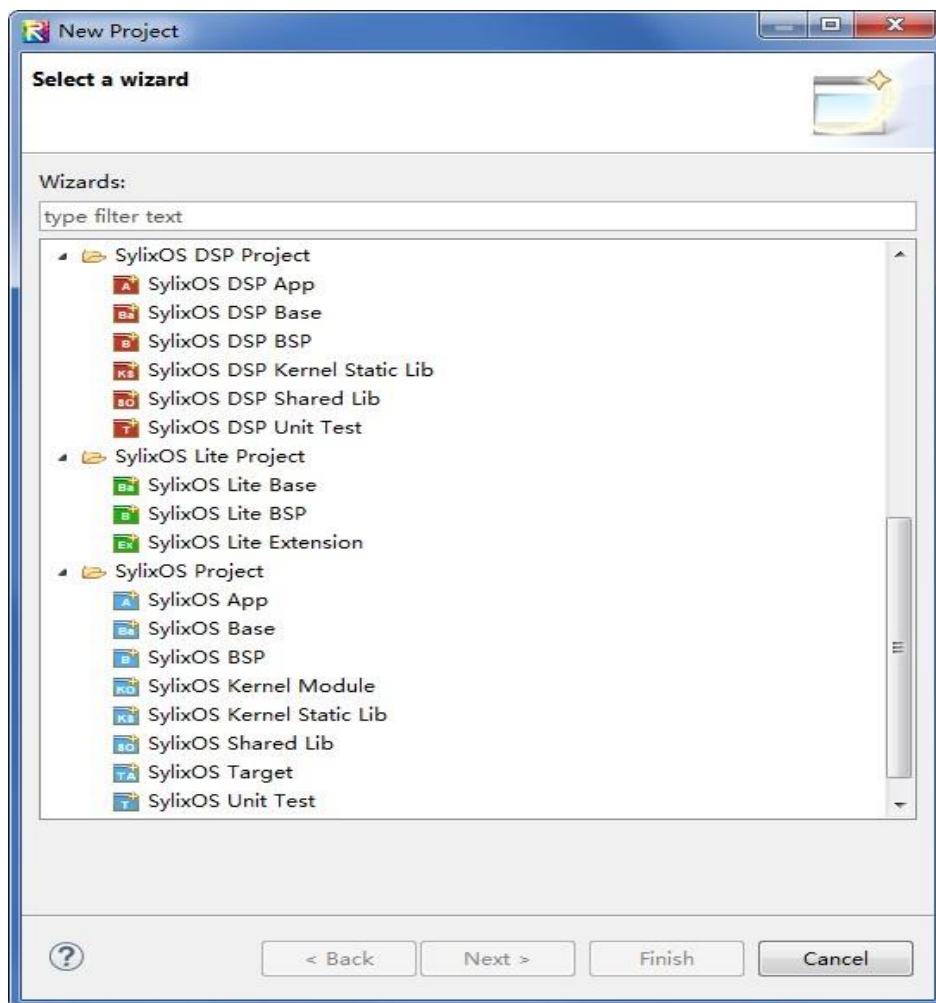


Figure 2.1 RealEvo-IDE project template

Note: the experience version of RealEvo-IDE does not include the SylixOS Lite Project, and the function can be obtained by purchasing the Pro version of RealEvo-IDE.

Figure 2.2 shows the communication and debugging relationship between RealEvo-IDE integrated development environment and SylixOS target system.

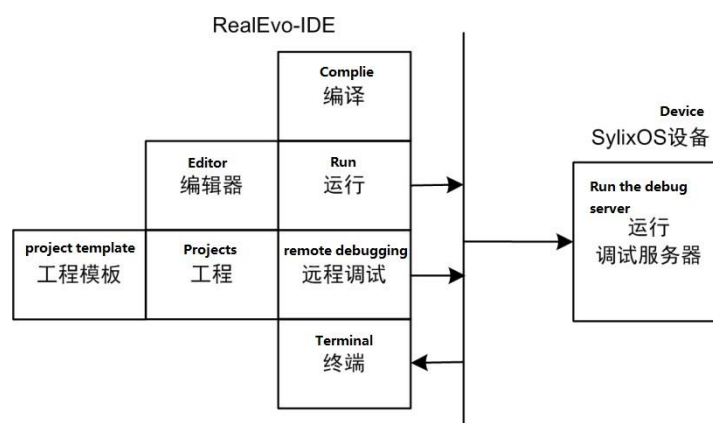


Figure 2.2 RealEvo-IDE and target machine

2.2.2 Functions

1. Edit

Code edition is one of the most fundamental and important work for software development, and an efficient code editor will yield twice the result with half the effort. The RealEvo-IDE editor has multiple color assortment schemes and functions of code static analysis and supplement. Therefore, code development is more efficient.

2. Compilation

The RealEvo-IDE toolbar contains a one-key compilation button on the left, as shown in Figure 2.3. After the project you want to compile is selected, click this button for compilation. In addition, you can click the right mouse button on the project requiring compilation and select "Build Project" for compilation.



Figure 2.3 compilation button

3. Debugging

SylixOS has a debugging server with powerful function, which can realize online debugging of applications on devices with SylixOS running. RealEvo-IDE provides the supported debugging tool, which can conveniently debug SylixOS application. At present, RealEvo-IDE supports three debugging methods: automatic push debugging, manual debugging, and remote attachment debugging (Attach).

4. Push

In order to more conveniently and rapidly deploy SylixOS application and driver to the SylixOS device, RealEvo-IDE provides the function of one-key push to conveniently deploy the program compiled on the SylixOS target system (see Section 4.1.4).

2.3 Introduction to GCC toolchain

The GCC toolchain is a group of compilation kit (GCC), plus some binary files (such as the linking tool LD, object file packaging tool AR and so on) and some standard compilation kits consisting of C library. The GCC toolchain with RealEvo-IDE is the compilation tool formed by adding SylixOS-related elements and more efficient library files based on standard GCC and satisfying SylixOS requirements.

2.4 Installation of RealEvo-IDE

2.4.1 Acquisition of SylixOS development kit

SylixOS development kit is divided into experience and commercial versions, and you can obtain a experience version of the development kit in the following ways: visit SylixOS official website (www.sylixos.com or www.acoinfo.com), and you can apply for a complete set of SylixOS integrated development environment.

2.4.2 Installation of SylixOS development kit

After a set of SylixOS development kit is obtained successfully, one can carefully read the document in CD to obtain more information about the SylixOS integrated development kit.

The following process is to install RealEvo-IDE through the SylixOS development kit CD:

Open SylixOS IDE, double click to open the InstallWizard.exe file, and you will see

installation toolset of the SylixOS Development Kit, precautions, and SylixOS website information. It can be seen from Figure 2.4 that SylixOS IDE includes installation of “RealEvo-IDE”, “QtCreator” and “RealEvo-QtSylixOS”.

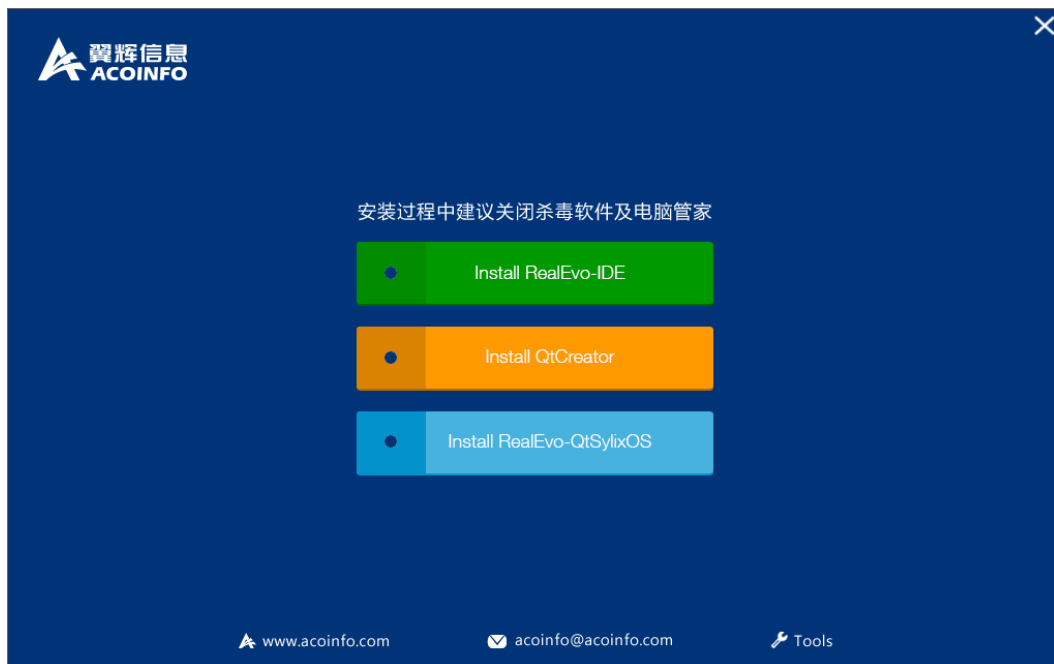


Figure 2.4 Integrated development environment toolset

It is installed according to the sequence in Figure 2.4, RealEvo-IDE in SylixOS integrated development environment can be used only after registration. For the specific registration process and the place to be noticed during registration, view the file of RealEvo Software Registration Procedure in CD.

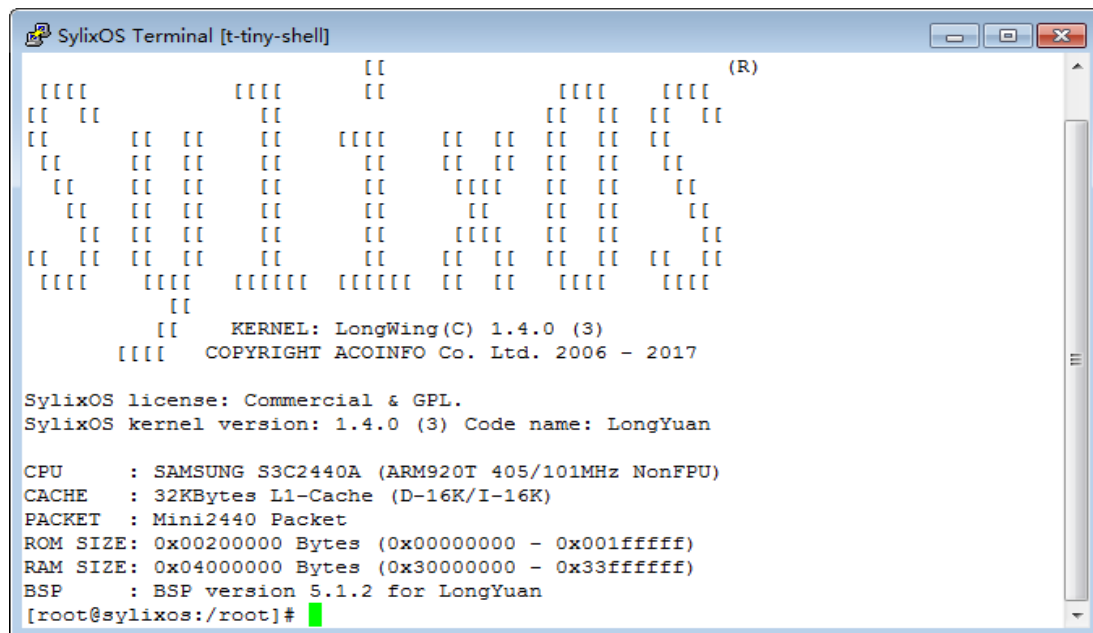
After above process, installation of SylixOS integrated development environment is completed.

Chapter 3 Brief introduction to Shell

3.1 What's Shell

Shell is the "shell" program of the operating system, it provides the user with a user interface based on command-line type, and can be called as the command parser. The system developer usually use the interface to operate the computer. Almost all operating systems include Shell programs, for example: the more common shell in Linux is the Bash program, and the shell program in Windows is cmd.exe. SylixOS is no exception, and also includes its own Shell program: `tinyShell`.

The `tinyShell` program is the simplest and most convenient interface for the system developer to operate the SylixOS operating system. It has the functions similar to those of the Linux system, what's different is that `tinyShell` runs in the kernel space, and it is not an application^①. The `tinyShell` can run application, and many common commands solidified in SylixOS kernel are built in. The operation interface of `tinyShell` program is shown in Figure 3.1.



```
SylixOS Terminal [t-tiny-shell] (R)
[[[[[   [[[[[   [[[[[   [[[[[   [[[[[
[[  [[   [[    [[    [[    [[    [[    [[    [[    [[
[[    [[  [[   [[    [[[[[[  [[  [[   [[    [[    [[
[[      [[  [[   [[    [[    [[    [[    [[    [[
[[        [[  [[   [[    [[    [[    [[    [[    [[
[[          [[  [[   [[    [[    [[    [[    [[    [[
[[[[[[  [[[[[[  [[[[[[  [[[[[[  [[[[[[  [[[[[[  [[[[[[
[[
[[          KERNEL: LongWing(C) 1.4.0 (3)
[[[[[[  COPYRIGHT ACOINFO Co. Ltd. 2006 - 2017

SylixOS license: Commercial & GPL.
SylixOS kernel version: 1.4.0 (3) Code name: LongYuan

CPU      : SAMSUNG S3C2440A (ARM920T 405/101MHz NonFPU)
CACHE    : 32KBytes L1-Cache (D-16K/I-16K)
PACKET   : Mini2440 Packet
ROM SIZE: 0x00200000 Bytes (0x00000000 - 0x001fffff)
RAM SIZE: 0x04000000 Bytes (0x30000000 - 0x33ffffff)
BSP      : BSP version 5.1.2 for LongYuan
[root@sylixos:/root]#
```

Figure 3.1 `tinyShell` operation interface

3.2 Instructions for common Shell commands

Some common `tinyShell` built-in commands will be briefly introduced in the section, which are divided into the system command, file command, network command, time command, dynamic load command and other commands, and detailed instructions can be

viewed on the Sylix OS device with the *help* [keyword] command.

The kernel version is different from clipping configuration. Therefore, *ttinyShell* built-in commands will be different on SylixOS systems with different versions and configurations.

3.2.1 System command

For *ttinyShell* built-in system commands, see Table 3.1.

Table 3.1 Common built-in system commands of SylixOS

Command name	Brief description
<i>help</i>	Display list of all built-in commands of <i>ttinyShell</i>
<i>free</i>	Display current memory information of the system
<i>echo</i>	Echo parameters entered by the user
<i>ts</i>	View thread information in the system
<i>tp</i>	Check information of the thread blocked in the system
<i>ss</i>	View status conditions of all threads and interrupt system stacks in the system
<i>ps</i>	View information of all system processes
<i>touch</i>	Create a regular file
<i>ints</i>	View information of the system interrupt vector table
<i>mems</i>	View memory utilization of kernel and system memory heap of the operating system
<i>zones</i>	View partition status conditions of physical pages of the operating system
<i>env</i>	View the global environment variable table of the operating system
<i>varsave</i>	Save the environment variable table of current operating system, and the default save path is <i>/etc/profile</i>
<i>varload</i>	Extract the load environment variable table from the file with specified parameters, and from <i>/etc/profile</i> at default
<i>vardel</i>	Delete a appointed system environment variable
<i>cpuus</i>	View cpu utilization rate
<i>top</i>	View cpu utilization rate
<i>kill</i>	Send the signal to the appointed thread or process, SIGKILL signal at default
<i>drvlics</i>	Display table information of all device drivers installed in the system
<i>devs</i>	Display all devices mounted in the system
<i>buss</i>	Display all bus information mounted in the system
<i>tty</i>	Display tty file corresponding to current Shell terminal
<i>clear</i>	Clear the current screen.
<i>aborts</i>	Display statistical information of exception handling of current operating system
<i>sprio</i>	Set priority of the appointed thread
<i>renice</i>	Set priority of the appointed process
<i>hostname</i>	Display or set the host name of current SylixOS mirror image
<i>login</i>	Switch the user and log in again

who	View identity of the current login user
shutdown	Shut down or restart the system
monitor	Start, shut down, or set the kernel tracker
pcis	Print related information of PCI bus and equipment of system
lusb	Print related information of USB bus and USB device of the system (dependent on USB library)
which	Check the file location appointed by the parameter
exit	Quit current Shell terminal

The following are several common and more important commands.

Command *ts*

The information of current running thread of SylixOS system can be viewed with “*ts*” command.

[Command format]

```
ts [pid]
```

[Common option]

None

[Instructions for parameters]

```
pid : process ID
```

The following are detailed meanings of output information of the *ts* command:

```
# ts①
thread show >>

      NAME           TID    PID  PRI  STAT  ERRNO    DELAY    PAGEFAILS    FPU  CPU
-----
t_idle           4010000     0 255  RDY      0         0         0         0
.....
thread : 16
```

Various meanings of output are as follows:

① “#” represents the administrator user in SylixOS, and here is no user name or other information compared with the reality ([root@sylixos_station:/]#). In order to avoid differences among different users, “#” is used to represent operation under ttinyShell in the book, and the path information is ignored.

- NAME: it is the thread name, for example, t_idle represents the IDLE thread of SylixOS;
- TID: it is the thread ID (handle), represented with hexadecimal system, such as 4010001;
- PID: it is ID of the process belonging to the thread, represented with decimal system, which is same with the representation method in UNIX system (0 represents the kernel thread of the operating system);
- PRI: it is the priority of the thread, represented with decimal system (the smaller the value, and the higher the priority.), such as 255;
- STAT: it is the current state of the thread, and RDY represents the ready state (for the thread state, see 6.2 Thread State Machine);
- ERRNO: it is the running error number;
- DELAY: it is the thread delay;
- PAGEFAILS: it is missing page interruption counting;
- FPU: it represents whether the hardware floating-point unit (fpu) is used;
- CPU: it represents on which CPU the thread is running (there may be other values on the multi-core system);
- thread: it represents the total number of current running thread.

2. Command tp

The blockage information of current running thread of SylixOS system can be viewed with the command.

[Command format]

```
tp [pid]
```

[Common option]

```
None
```

[Instructions for parameters]

```
pid : process ID
```

The following are detailed meanings of output information of the **tp** command:

```
# tp
thread pending show >>

      NAME           TID      PID  STAT   DELAY      PEND EVENT      OWNER
```

```

-----
t_except          4010002      0 SEM          0 10010003:job_sync
.....
pending thread : 14

```

Various meanings of output are as follows:

- NAME: it is the name of the thread
- TID: it is ID of the thread;
- PID: it is ID of the process;
- STAT: it is the current state of the thread;
- DELAY: it is the thread delay;
- PEND EVENT: it represents what kind of event the thread is currently blocking, such as semaphore, message queue and so on;
- OWNER: it represents the thread ID occupying the blocking object when the thread is blocked (the domain will include the thread ID occupying the lock with occupy lock in case of any deadlock);
- pending thread: it represents that 14 threads among the running threads are blocked on an event.

3. Command ps

The information of the running process of SylixOS system can be viewed with the **ps** command.

[Command format]

```
ps
```

[Common option]

```
None
```

[Instructions for parameters]

```
None
```

The following are detailed meanings of output information of the **ps** command:

```

# ps

```

NAME	FATHER	STAT	PID	GRP	MEMORY	UID	GID	USER
kernel	<orphan>	R	0	0	0KB	0	0	root

```

app          <orphan>      R      2      2      196KB    0      0
root
total vprocess: 2

```

Various meanings of output are as follows:

- NAME: process name (program name);
- FATHER: it represents the father process, orphan represents an orphan process, i.e., there is no father process;
- STAT: it represents the process state, as shown in Table 3.2;
- PID: it is ID of the process;
- GRP: it is ID of the process group;
- MEMORY: it is the total memory consumed by the process (unit: byte);
- UID: it is the user ID of the process;
- GID: it is the user group ID of the process;
- USER: it is the user name of the process, such as root.

Table 3.2 Process state

State label	Note
I	Initial state of the process, the process has not started to run yet;
R	Running state of the process, the process is running;
T	Stop state of the process, the process stops running for a certain reason;
Z	Zombie state of the process, the process has exited, waiting for the resource to be recycled.

4. Command *ints*

The interrupt vector information of SylinOS system can be displayed with the *ints* command.

[Command format]

```
ints [cpuid start] [cpuid end]
```

[Common option]

None

[Instructions for parameters]

cpuid start : CPU ID at the beginning

cpuid end : CPU ID in the end

The following are detailed meanings of output information of the *ints* command:

```
# ints
interrupt vector show >>

IRQ      NAME          ENTRY    CLEAR    PARAM    ENABLE  RND  PREEMPT    CPU 0
-----
  7  dm9000_isr    20013978      0  2c62fbe8  true
.....
interrupt nesting show >>

CPU  MAX NESTING    IPI
-----
  0          1          0

interrupt vector base : 0x2c7a96a8
```

Various meanings of output are as follows:

- IRQ: it is the interrupt number;
- NAME: it is the registered interrupt name;
- ENTRY: it is the address of the interrupt service function, represented with hexadecimal system, such as 20013978;
- CLEAR: it is the address of the interrupt cleaning function;
- PARAM: it is the parameter address of the interrupt service function;
- ENABLE: it represents whether the interrupt is enabled;
- RND: it represents whether it can be used as the system random number seed;
- PREEMPT: it represents whether occupation is allowed;
- CPU 0 (0 represents the CPU number): it represents the number of interrupts generated on CPU0. For example, dm9000_isr generates 4068 interrupts on CPU0.

3.2.2 File command

Built-in file commands of ttinyShell are shown in Table 3.3.

Table 3.3 Common built-in file commands of the system

Command name	Brief description
<i>ls</i>	List files in the appointed directory, current directory at default
<i>ll</i>	List the detailed information of the file in the appointed directory, current directory at default
<i>files</i>	List the information of the file opened in the system kernel (excluding the file opened by the process)
<i>fdentrys</i>	Lists all file information on which the operating system is working (including files opened by the process)
<i>sync</i>	Write all system cache files, devices, and disk information in the corresponding physical device
<i>logfileadd</i>	Add the appointed kernel file descriptor to the kernel log print function
<i>logfileclear</i>	Clear the appointed kernel file descriptor from the list of the kernel log print files
<i>logfiles</i>	Display the list of kernel log print files
<i>loglevel</i>	Display or set the print level of current kernel log
<i>cd</i>	Switch the current directory
<i>pwd</i>	View current working directory
<i>df</i>	View the file system information of the appointed directory
<i>tmpname</i>	Obtain a temporary file name which can be created
<i>mkdir</i>	Create a directory
<i>mkfifo</i>	Create a naming pipeline. Notice: it can only be created under the device of the root file system
<i>rmdir</i>	Delete a directory
<i>rm</i>	Delete a file
<i>mv</i>	Move or rename a file
<i>cat</i>	View contents of a file
<i>cp</i>	Copy a file
<i>cmp</i>	Compare contents of two files
<i>dsiz</i>	Calculate all file information contained in an appointed directory
<i>chmod</i>	Set the permission bit of of the file or directory
<i>mkfs</i>	Format the appointed disk
<i>shfile</i>	Execute the appointed Shell script
<i>mount</i>	Mount a volume
<i>umount</i>	Uninstall a volume
<i>showmount</i>	View all volumes mounted in the system
<i>ln</i>	Create the symbolic link file
<i>dosfslabel</i>	View the volume label of the fat file system
<i>fatugid</i>	Set the user and group id of the fat file system
<i>mmaps</i>	Display the system mmap information
<i>fdisk</i>	Disk partition

1. Command *fdisk*

The disk partition can be displayed or the partition table of the disk device can be created with the *fdisk* command

[Command format]

```
fdisk [-f] [block I/O device]
```

[Common option]

```
-f: Specified disk device
```

[Instructions for parameters]

```
block I/O device: block device, such as /dev/blk/sdcard0
```

The following is how to use the *fdisk* command:

Display *udisk0* partition table:

```
# fdisk /dev/blk/udisk0
```

Create the partition table:

```
# fdisk -f /dev/blk/udisk0
```

4 partitions can be created with *fdisk* (number of partitions: 1 to 4), the size percentage of each partition shall be indicated (such as 40%), one can select whether the appointed partition is the active partition (including: active and inactive). The file system types currently supported include: 1:FAT, 2: TPSFS (SylixOS power-failure security file system), and 3: LINUX.

3.2.3 Network command

Built-in network commands of *tinyShell* are shown in Table 3.4.

Table 3.4 Common built-in network commands of the system

Command name	Brief description
<i>route</i>	Add, delete, modify or view the system routing table
<i>netstat</i>	View the network state
<i>ifconfig</i>	Configure the network interface information
<i>ifup</i>	Enable a network interface
<i>ifdown</i>	Disable a network interface
<i>arp</i>	Add, delete, or view the ARP table
<i>ping</i>	ping command
<i>ping6</i>	IPv6 ping command
<i>tftpdpath</i>	View or set the local path of tftp server
<i>tftp</i>	Receive and send a file with the tftp command
<i>ftps</i>	Display ftp server information
<i>ftpdpath</i>	View or set the initialization path of ftp server
<i>nat</i>	Start, shut down, or set service of NAT virtual network address
<i>nats</i>	View the service state of current NAT virtual address
<i>npfs</i>	View the state of the network packet filter
<i>npfruleadd</i>	Add a rule of the network packet filter
<i>npfruledel</i>	Delete a rule of the network packet filter
<i>npfattach</i>	Enable the network packets filter on the appointed network interface
<i>npfdetach</i>	Disable the network packets filter on the appointed network interface
<i>flowctl</i>	ioctl flow control

1. Command *ifup* and *ifdown*

The *ifup* command enables the appointed network interface, and can open and close dhcp lease at the same time. The *ifdown* command can disable the appointed network interface.

[Command format]

```
ifup [netifname] [{-dhcp | -nodhcp}]
```

```
ifdown [netifname]
```

[Common option]

```
-dhcp : Open DHCP lease
```

```
-nodhcp : close DHCP lease
```

[Instructions for parameters]

```
netifname : network interface name (eg: en1)
```

The following shows how to use *ifup* and *ifdown* commands:

- Enable network interface en1;

```
ifup en1
```

- Enable network interface en1 and open dhcp lease;

```
ifup en1 -dhcp
```

- Enable network interface en1 and stop dhcp mode;

```
ifup en1 -nodhcp
```

- Stop network interface en1.

```
ifdown en1
```

2. Command *flowctl*

The *flowctl* command enables the flow control function, which can perform flow control for the network interface, IPv4 and IPv6.

[Command format]

```
flowctl [cn] [type] ips ipe [proto] ps pe dev [ifname] [dl][ul] bufs
```

[Common option]

```
cn :
  add : add
  del : delete
  chg : change

type :
  ip : traffic control for IP address
  if : traffic control for network interface

proto :
  tcp : TCP protocol
  udp : UDP protocol
  all : default protocol
```

[Instructions for parameters]

```
When type is ip:
  ips : IP address at the beginning
  ipe : IP address in the end
  ps : port number at the beginning
  pe : port number in the end

when type is if:
  Do not need to enter ips, ipe, proto, ps, and pe
```

```
ifname : network interface name (eg: en1)
bufs : buffer size
dl : download speed
ul : upload speed
```

The following shows how to use the **flowctl** command:

- Add the flow control information of the network interface;

```
flowctl if dev en1 50 100 64
```

- Add IPv4 flow control information;

```
flowctl add ip 192.168.1.1 192.168.1.10 tcp 20 80 dev en1 50 100 64
```

- Delete IPv4 flow control information;

```
flowctl del ip 192.168.1.1 192.168.1.10 tcp 20 80 dev en1
```

- Modify IPv4 flow control information.

```
flowctl chg ip 192.168.1.1 192.168.1.10 tcp 20 80 dev en1 100 200
```

For flow control information, the uplink and downlink speed can only be modified, and modification in ip and the port number is invalid.

For detailed information of other network commands, see Chapter I15 Network I/O.

3.2.4 Time command

Built-in time commands of tinyShell are shown in Table 3.5.

Table 3.5 Common built-in time commands of SyinxOS

Command name	Brief description
<i>date</i>	Display or set the system time
<i>times</i>	Display the current time of the system
<i>hwclock</i>	Display or synchronize the operating system and hardware RTC clock

1. Command *date*

The system time can be displayed or set with the *date* command

[Command format]

```
date [-s {time | date}]
```

[Common option]

```
-s: set time
```

[Instructions for parameters]

```
time: hour、minute、second format
```

```
date: year、month、day format
```

The following shows how to use the *date* command:

- Display the system time;

```
date
```

- Set 24h time format of the system;

```
date -s 18:15:09
```

- Set system date.

```
date -s 20150918
```

2. Command *hwclock*

hwclock command can display or synchronize hardware RTC clock.

[Command format]

```
hwclock [--show | --hctosys | --systohc]
```

[Common option]

```
--show : show RTC time
```

```
--hctosys : Synchronize RTC time to system time
--systemhc : Synchronize system time to RTC time
```

[Instructions for parameters]

None

The following shows how to use the *hwclock* command:

- Display hardware RTC time;

```
hwclock -show
```

- Synchronize hardware RTC time to system time;

```
hwclock --hctosys
```

- Synchronize system time to hardware RTC time.

```
hwclock --systemhc
```

3.2.5 Dynamic loading command

The tinyShell built-in dynamic loading command is shown in Table 3.6.

Table 3.6 SylixOS common built-in dynamic loading commands

Command name	Brief description
<i>debug</i>	Debug a process
<i>dlconfig</i>	Configure working parameter of the dynamic linker
<i>modulereg</i>	Register a kernel module
<i>moduleunreg</i>	Uninstall a kernel module
<i>modulestat</i>	View a kernel module or dynamic link library file information
<i>lsmmod</i>	See all kernel module information loaded by the system
<i>modules</i>	View all kernel module and process dynamic link library information loaded by the system

1. Command *debug*

The *debug* command is used to debug the SylixOS application.

[Command format]

```
debug [connect options] [program] [arguments...]
```

[Common option]

None

[Instructions for parameters]

```
connect options : Connection options
program : program name
arguments... : Program parameter list
```

The following shows how to use the **debug** command:

- Through network debugging, the SylixOS device starts the debugger. localhost:1234 represents the debugging port number of the SylixOS device: 1234, ./app is the application program to be debugged in the current directory;

```
debug localhost:1234 ./app
```

- Through serial debugging, the SylixOS device starts the debugger. The default serial port baud rate is 115200 bps, the number of data bits is 8, the stop bit is 1 bit, and there is no parity check.

```
debug /dev/ttyS1 ./app
```

3.2.6 Other commands

The tinyShell built-in other commands are shown in Table 3.7.

Table 3.7 SylixOS common built-in other commands

Command name	Brief description
shstack	Display or set the Shell task stack size, and setting is only valid for the shell started later
leakchkstart	Start system memory leak tracker
leakchkstop	Turn off system memory leak tracker
leakchk	Memory leak check
xmodems	Send a file thorough the xmodem protocol
xmodemr	Receive a file thorough the xmodem protocol
untar	Unpack or unzip a tar or tar.gz file package
gzip	Compress or decompress a file
vi	Start the vi editor

1. Commands *leakchkstart*, *leakchkstop*, and *leakchk*

The above commands can detect system memory leak.

[Command format]

```
leakchkstart [max save node number] [pid]
```

```
leakchkstop
```

```
leakchk
```

[Common option]

None

[Instructions for parameters]

```
max save node number : Maximum storage nodes  
pid : Process ID
```

The following shows how to use the above commands:

Start the memory leak tracker, the second parameter 2048 is the maximum number of nodes tracked, and the third parameter 0 is detection of kernel memory. When the parameter is less than 0, it represents detection of all memory (kernel and user process). If the parameter is larger than 0, it represents detection of memory of the specified process,

```
leakchkstart 2048 0
```

Detect memory leaks,

```
leakchk
```

Stop the memory leak tracker.

```
leakchkstop
```

Note: the number of nodes in the second parameter is the number of memory blocks in the fixed-length memory, and these memories are used for temporary cache. For detailed description of fixed-length memory, see Chapter 12 Memory Management.

3.3 Environment variable

Environment variable is an object with specific name in the operating system. It contains the information which one or more applications will use. It is generally used to specify some parameters of the operating system or the application execution environment, such as the temporary directory location, application search location, etc. The configuration information of environment variables is usually saved in the `/etc/profile` file.

A complete environment variable definition table is saved in Shell environment. When the system is started, BSP will automatically import definition of environment variables in this file into the Shell environment. Users can use the `env` command to view all environment variables of the system. When an application is started, this environment variable will be imported into the application process (meanwhile, creating several environment variables which indicate application parameters, such as HOME, etc.).

The environment variable is equivalent to some parameters set for the system or the user application, and the specific role is related to the specific environment variable. The common environment variables of SylixOS are shown in Table 3.8.

Table 3.8 Description of SylixOS environment variables

Name of the environment variable	Brief description
XINPUT_PRIO	xinput subsystem priority
LANG	Language selection
LD_LIBRARY_PATH	Dynamic loader search path
PATH	Application search path
NFS_CLIENT_PROTO	NFS client protocol
NFS_CLIENT_AUTH	NFS login authentication mode
SYSLOGD_HOST	syslogd remote address
SO_MEM_PAGES	Number of page initially occupied by the application memory heap
TSLIB_TSDEVICE	Touch screen device
MOUSE	xinput subsystem detection mouse device set
KEYBOARD	xinput subsystem detection keyboard device set
TZ	System time zone
TMPDIR	Temporary folder

The user can also quote the value of the environment variable in the command line, and the reference format is `${VAR_NAME}`. When this parameter is encountered, tinyShell will automatically use the contents of the environment variable to substitute `${VAR_NAME}`. For example: tinyShell executes the `echo ${PATH}` command, and the system will echo the contents of PATH environment variable.

Users can add their own environment variables in the following format:

```
VAR=VALUE
```

Use command **varsave** to save the Shell environment variable to the default configuration file `/etc/profile`. An environment variable can have multiple different values. Different values are separated with a colon, as shown below:

```
PATH /usr/bin:/bin:/usr/local/bin:/home/user
```

3.4 Root file system

The first file system automatically mounted after SylixOS is started is called as the root file system. Unlike Linux system, SylixOS root file system is a virtual file system. After the power failure, modification of the file system will not be saved, so SylixOS can work on a machine without non-volatile memory (usually a hard disk or other magnetic disc

memories). On this file system, SylixOS will automatically create the dev, media and mnt directories. Other standard directories require BSP to do mounts or symbolic links at the initialization phase.

The standard directory structure used by SylixOS is shown in Table 3.9.

Table 3.9 Instructions for directory of SylixOS root file system

Directory name	Brief description
qt	Dynamic link library of Qt graphics system and other Qt resource directories
tmp	Temporary directory
var	Store various changed file directories, such as logs, buffer files etc.
root	Home directory of root user
home	Home directory of other users
apps	Application directory
sbin	System program directory
bin	Common Shell program directory
usr	User program library and environment directory
lib	System program library and environment directory
etc	Directory of system or other application configuration files
boot	Directory of operating system boot image
media	Directory of automatic mount of file system device (such as USB, SD card and so on)
proc	Directory of system kernel information file
mnt	Directory of dynamic file system mount
dev	System device file directory

3.5 Operation application

ttinyShell can not only execute the built-in command, but also execute user application. The method of executing the application is the same with that of executing the built-in command. When the user types the command name and parameters in the Shell interface and clicks Enter, ttinyShell will firstly detect whether the command is a user application. If so, execute the user application first. If not, check whether the command is a built-in command. If not, ttinyShell will print the error message.

The ttinyShell command detection sequence is as follows:

- (1) Detect whether the command is a file path. If the file exists, execute the specified application.
- (2) Detect the path specified by the PATH environment variable. if the file exists, execute the specified application.
- (3) Detect whether the command is a built-in command. If so, execute the built-in command.

There are two ways for ttinyShell to execute the built-in command or application: synchronous and asynchronous.

- **Synchronous mode:** when ttinyShell executes the built-in command, the command code is in the context of the ttinyShell thread at default. When the application is executed, ttinyShell will create a process, and load the application code in the process. At the same time, ttinyShell blocks itself, and waits for the process to end to resume execution;
- **Asynchronous:** when the user enters & symbol behind the command keyed in, ttinyShell will execute the command asynchronously. When ttinyShell executes the built-in command, it will create a kernel thread to execute the command code. When the application is executed, ttinyShell will create a process and load application code in this process. Unlike the synchronization method, ttinyShell does not wait for the command to complete, but is immediately ready to receive the user's next command.

3.6 I/O redirection

Each application has three standard file descriptors (for detailed introduction, see Chapter 5 I/O System): 0, 1 and 2.

- 0 represents standard input, i.e., files read by scanf, getc and other functions;
- 1 represents standard output, i.e, files written by printf, putc and other functions;
- 2 represents standard error, i.e., files written by perror and other functions.

At default, `tinyShell` will use current terminal device as the standard file, and the process created by `tinyShell` will inherit settings of `tinyShell` standard file. Of course, the user can also set the standard file for the command. When there is the I/O redirection parameter behind command string, `tinyShell` will analyze redirection expression, and set the new standard file descriptor. The setting method is as follows:

- Redirect standard output to the `file_path` file;

```
echo "aaa" 1>file_path
```

- If it is required to add a certain file, one can use `1>>file_path`. The method of defining a standard error file is similar to that of a standard output file, such as `2>file_path①`.
- It is required to locate standard input to a certain file, and parameter `0<file_path` can be added with the command at the end.

Chapter 4 Compiling the first program

4.1 Hello world application

The chapter will focus on how to use the RealEvo-IDE development environment to create and compile SylixOS application and dynamic link library program.

The Hello world program is always the first program to appear in various programming language books as a classic example. We will use the Hello world project to introduce how to compile and run SylixOS application in the following.

4.1.1 Create SylixOS.base project

Open RealEvo-IDE, select the "File" → "New" → "SylixOS Base" template, and you can freely set "Project name", for example: fill in "SylixOS" here. Click "Next" to pop up the dialog box of project settings, as shown in Figure 4.2. In the dialog box, you can select the type of compilation toolchain (the default is arm-sylixos-toolchain), the debugging level of compilation code (the default is Debug) and CPU Type (the default is arm920t, and the option shall be selected according to the CPU type of the actual SylixOS target system.). After above settings, click "Next" to pop up the dialog box of default library selection, as shown in Figure 4.3. SylixOS Base includes libsylxos and libcextern at default (you can select other libraries as required). Finally, click "Finish" to complete creation of the SylixOS Base project. After the SylixOS Base project is created, use the method introduced in 2.2.2 for completion, and the project will generate multiple related libraries of the operating system kernel after completion.

Note: libsylxos is the the source code library of the SylixOS kernel, libcextern is c standard library of the application, which provides support of standard C function library for the application.

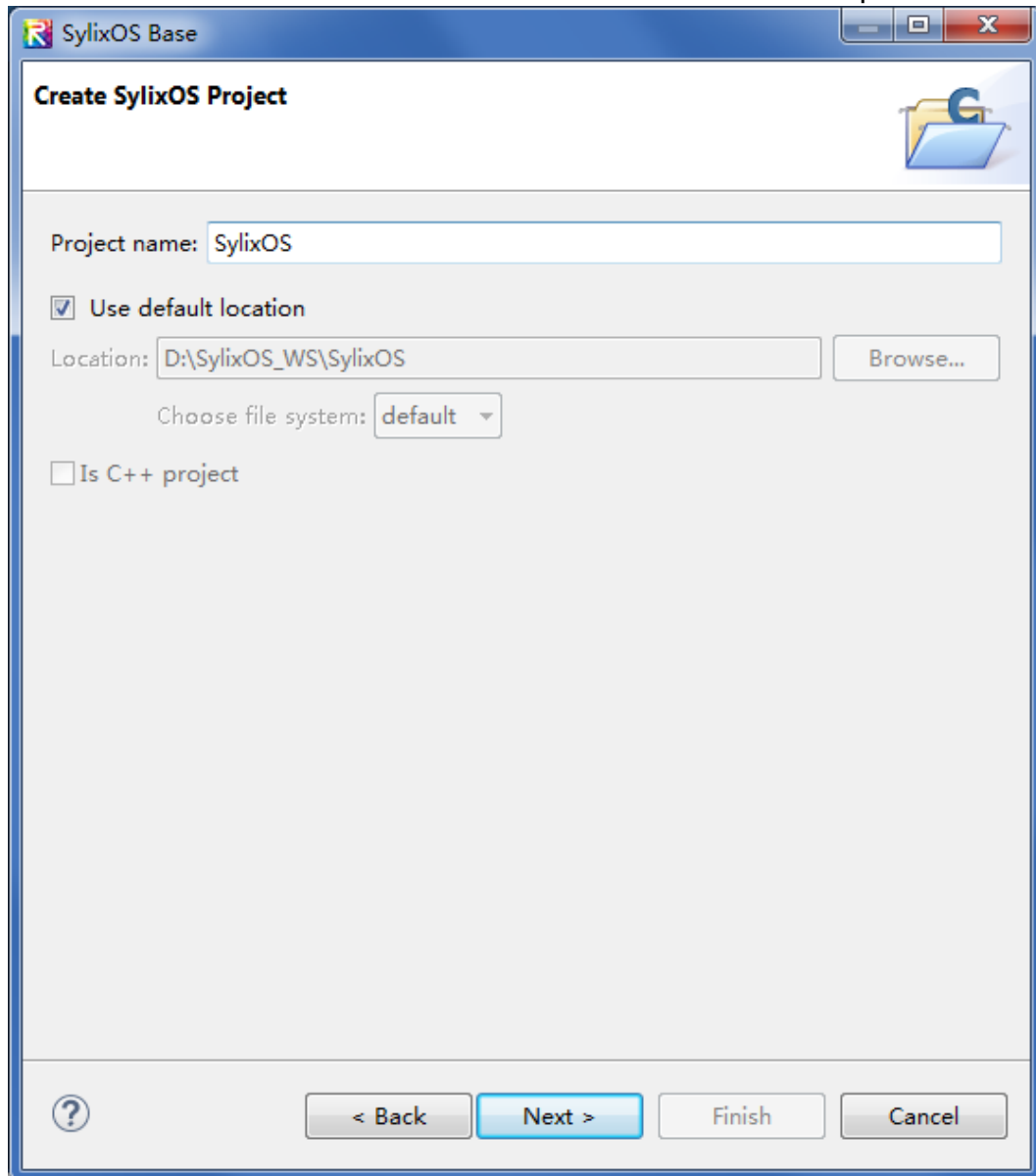


Figure 4.1 Base project creation

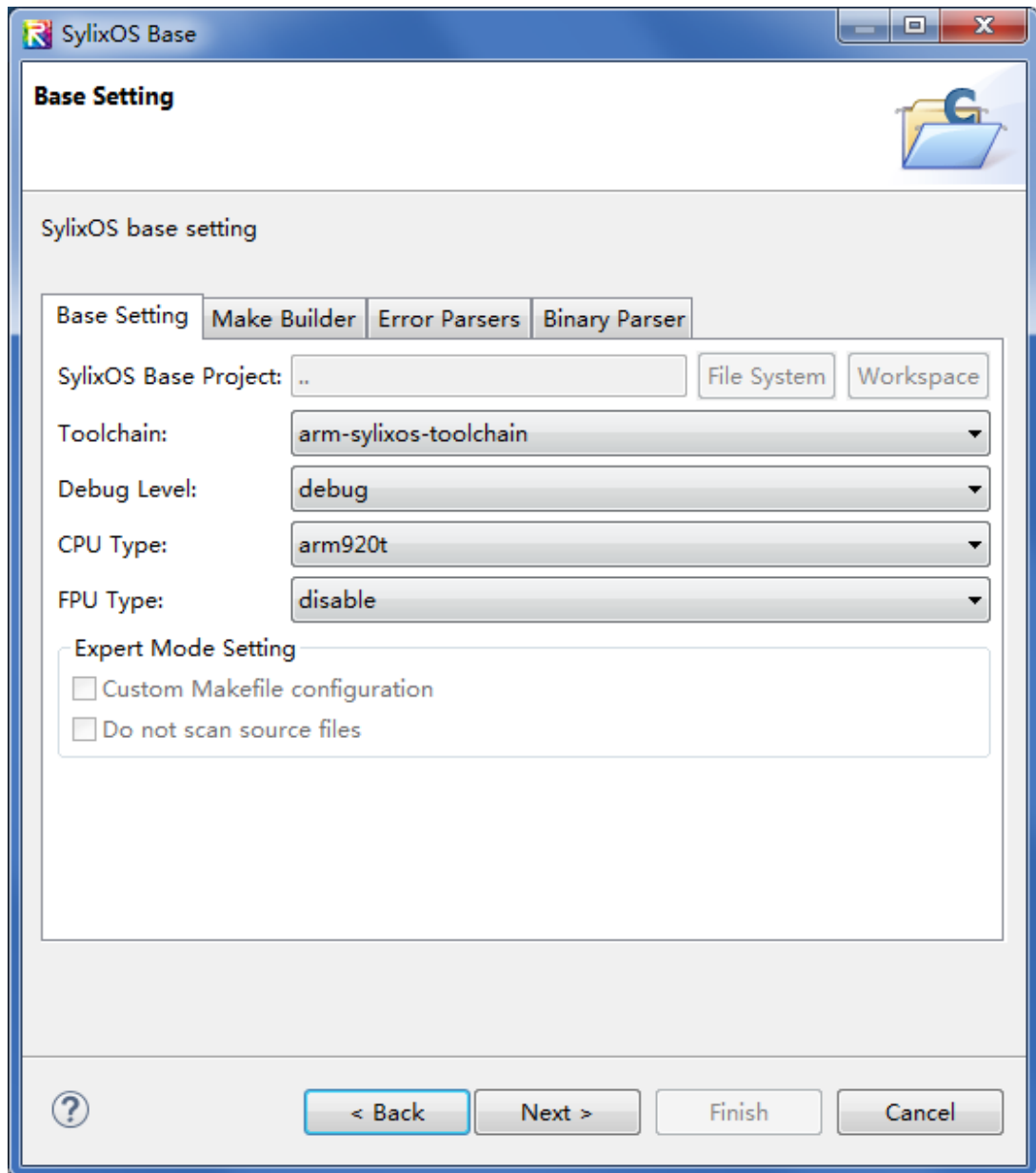


Figure 4.2 Base project settings

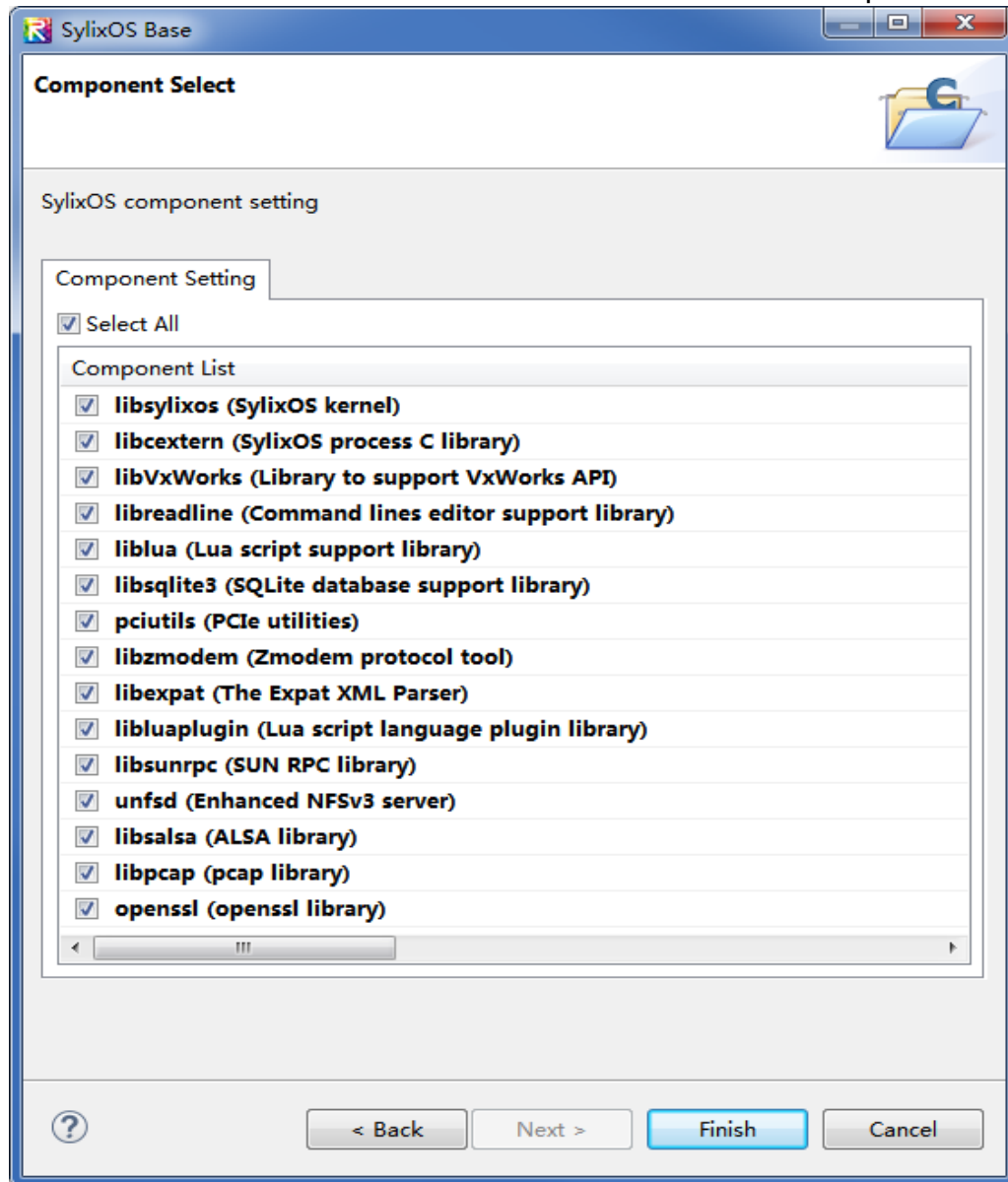


Figure 4.3 Default Library Selection

4.1.2 Create the Hello World Project

Open RealEvo-IDE, select "File" → "New" → "SylixOS App" template, fill "helloworld" in "Project name", Click "Next" to pop up the dialog box of project creation, as shown in Figure 4.5. In the "SylixOS Base Project" box, select the base project folder created and click "Finish" to complete creation of the helloworld project.

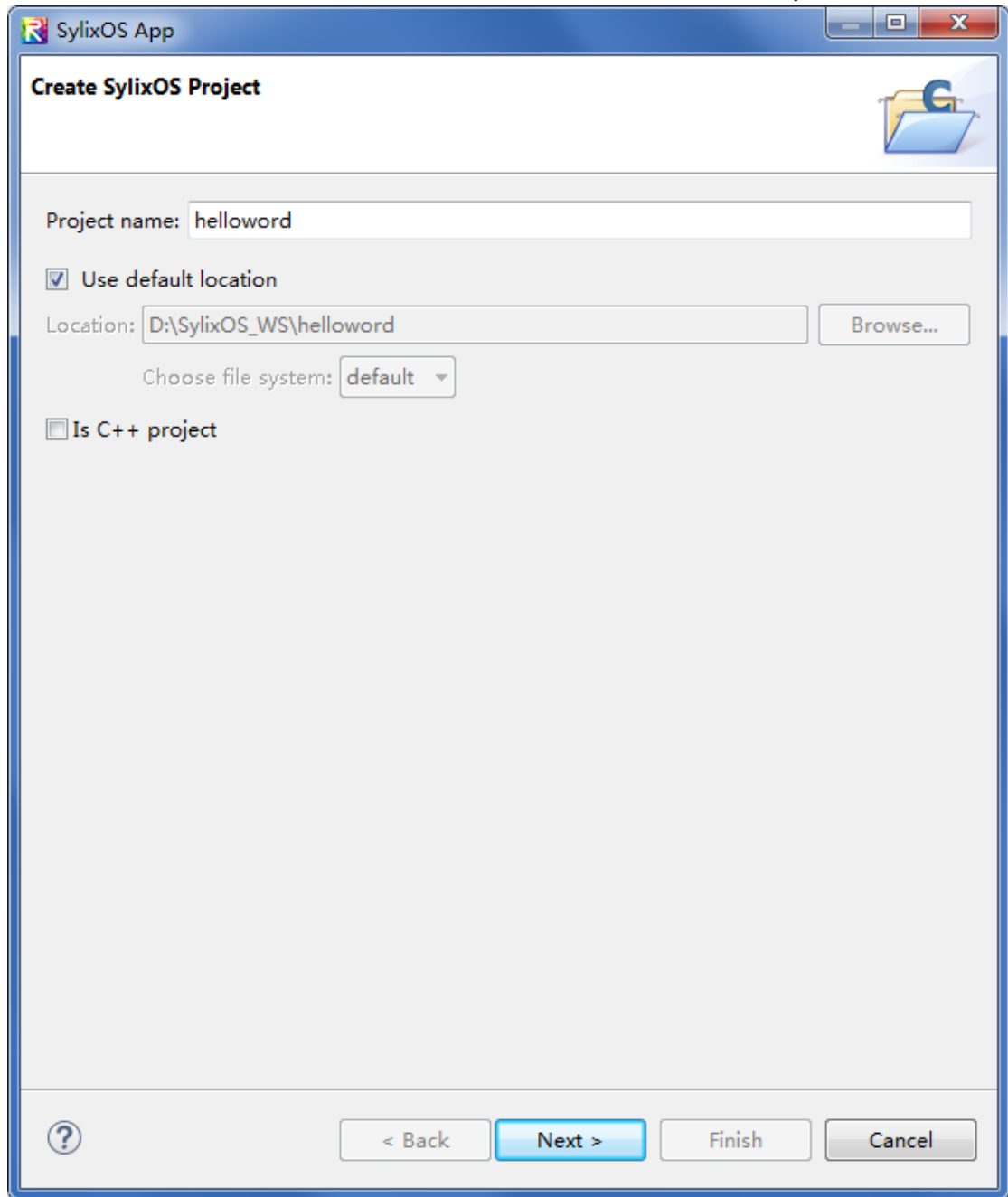


Figure 4.4 App project creation

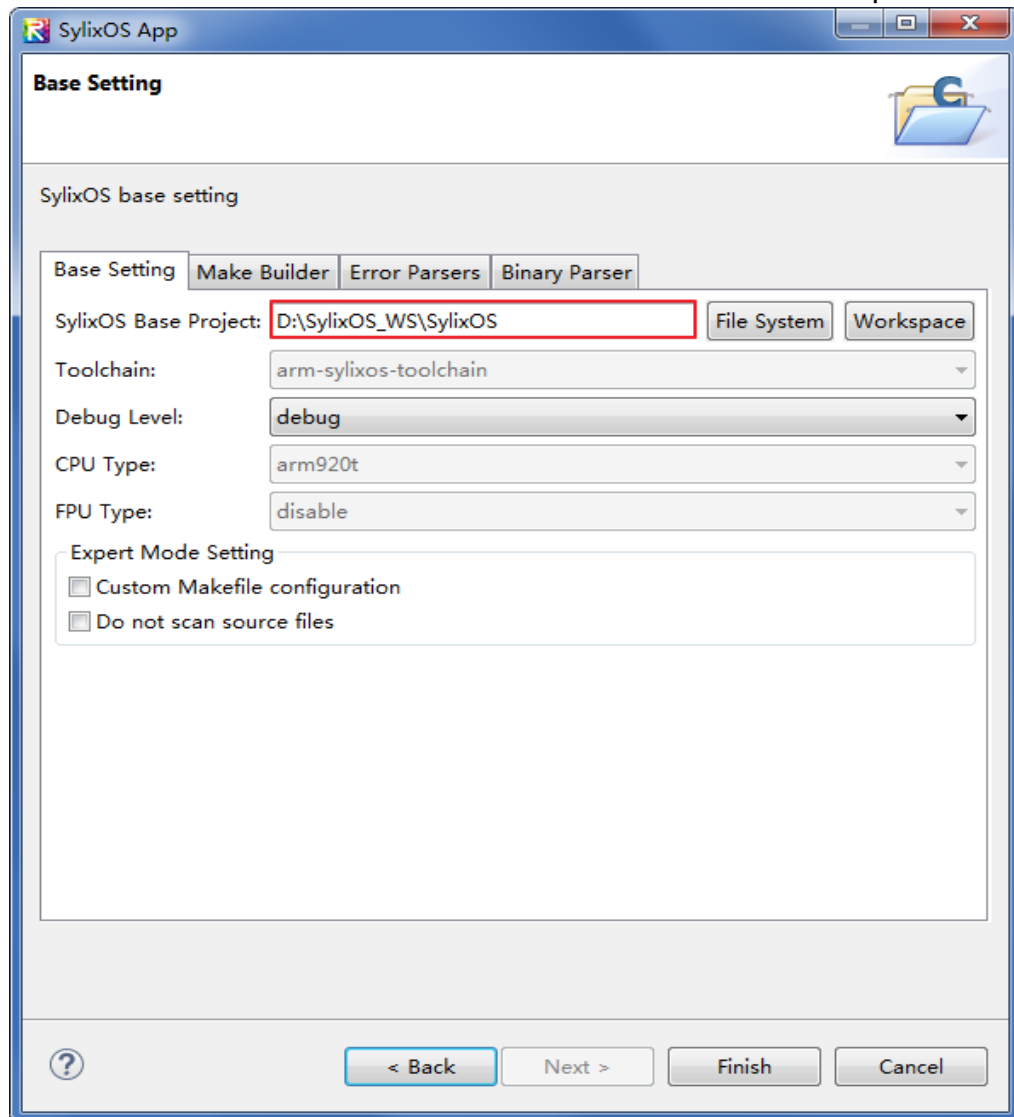


Figure 4.5 Project settings

4.1.3 Compiling the Hello world project

Select the created helloworld project in "Project Explorer", click the right mouse button and select "Build Project" to complete compilation of the helloworld project (you can also use the method introduced in 2.2.2 for compilation). After compilation, the Debug folder will be generated under the helloworld project, and the Hello World executable file compiled will be generated under the Debug folder, as shown in Figure 4.6.



Figure 4.6 Debug folder

4.1.4 Deployment file

SylixOS supports ftp server, and you can use ftp client (such as software FileZilla) to upload files to the SylixOS target system. You can obtain more information about use of the ftp client tool through the Internet, and the section focuses on how to upload files to the SylixOS target system through RealEvo-IDE.

1. Deployment Settings

Select the helloworld project, click the right mouse button to select "Properties" → "SylixOS Project" → "Device Setting", and click "New Device" to add a new device configuration, as shown in Figure 4.7. Finally, select the newly added device on the "Device Setting" configuration page and click OK to complete setting.

The image shows a "New Device" dialog box with the following fields and settings:

- Device Name:** Use default IP, Device Name: 192.168.7.30
- Host Settings:** Device IP: 192.168.7.30, FTP Port: 21, GDB Port: 1234, Telnet Port: 23
- User Settings:** User Name: root, Password: [masked]

Figure 4.7 New Device

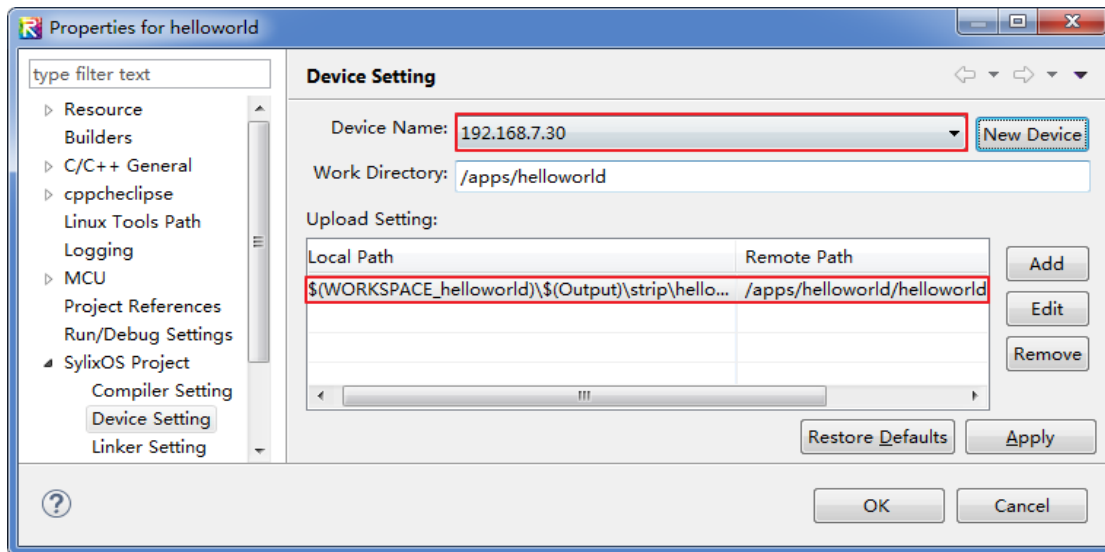


Figure 4.8 Device Setting

2. File uploading

After the target file is set, select the helloworld project, click the right mouse button and select the "SylixOS" option in the pop-up dialog box, then select "Upload", and RealEvo-IDE start uploading files. If the file is uploaded successfully, the printed word of "Upload file success!" will be indicated in the pop-up "Console". If the file upload fails, the printed word of "Upload file failed!" will be indicated. The situation is usually caused due to the Internet or other reasons, such as incorrect Ip address, wrong user name password, firewall interception and so on.

4.1.5 Run the Hello world application

There are two ways to run the helloworld program in the SylixOS device.

1. Running under the SylixOS Shell

The method of run the program under the SylixOS Shell is the same with that of the Linux system. Firstly, use the **cd** command to switch the directory to /apps / helloworld/, and use the **ls** command to view the file in the current directory to confirm that the file helloworld is included. Enter the ./helloworld executive program file, and running results are as shown below.

```
#ls
helloworld
#./helloworld
Hello World!
```

2. RealEvo-IDE start

Select the helloworld project and click the right mouse button to select "Running As→SylixOS Remote Application", and the SylixOS application will run automatically.

4.1.6 Debug the Hello world application

During program development, it is inevitable that the program will not run properly due to some programming problems, we can find the problem by checking the code specification or analyzing the code logic at the moment, and can quickly locate problems through the method when the amount of code is small. However, when the amount of code reaches tens of thousands of lines or even more, the situation will become worse. One can use the cross debugging technique for the situation to simplify the complicated problem, and can rapidly locate the problem. The following we will use the helloworld project as an example to show how to debug SylixOS application through RealEvo-IDE.

1. One-key push debugging

One-key push debugging function, it shall be configured in RealEvo-IDE as follows:

- Select the project to be debugged. Select the "helloworld" project here, and select the menu "Run" -> "Debug Configurations" to open the debugger interface;
- Select "SylixOS Remote Application", click "New launch configuration" button to create a new debugger object, and fill in the debugger object name in the "Name" edit box;

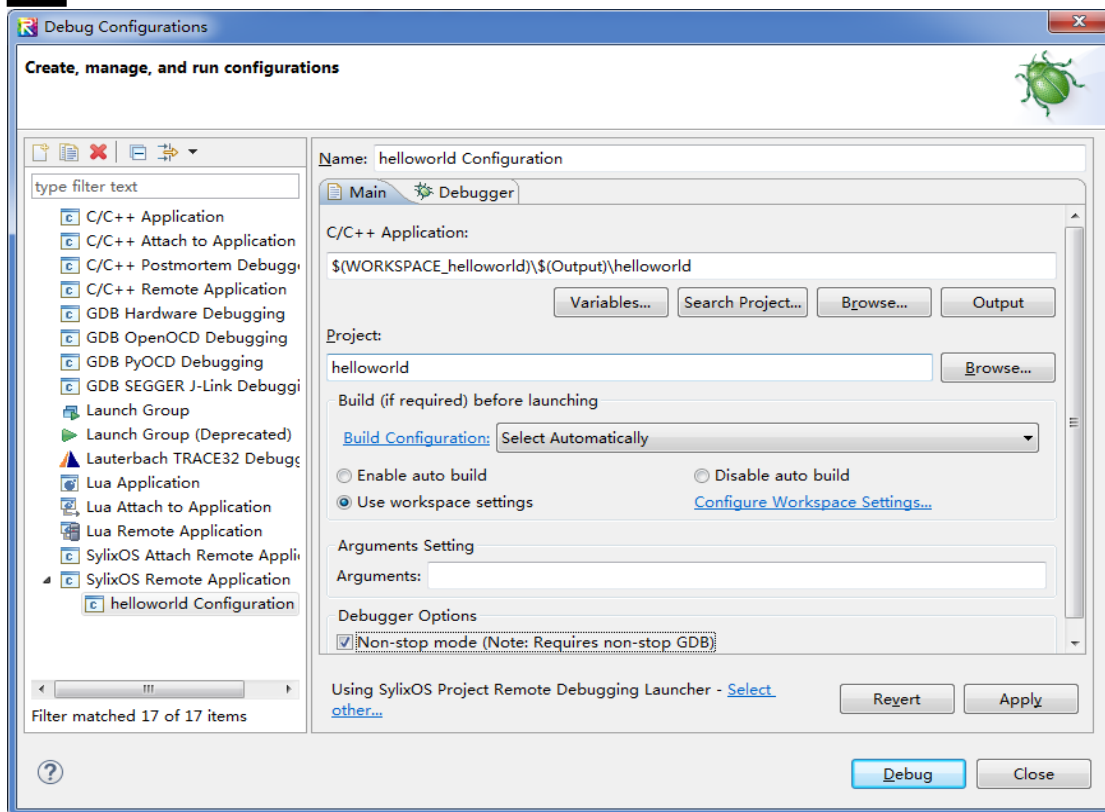


Figure 4.9 Debug configuration

Note: one can quickly start debugging by right-click on the successfully compiled application to select "Debug As" → "SylixOS Remote Application".

Click Figure 4.9 Debug button to enter the debugging interface as shown in Figure 4.10. Here are some RealEvo-IDE debugging commands:

- Start debugging (full speed operation) "Run" → "Resume (F8)";
- Step into the function "Run" → "Step Into (F5)";
- Step over "Run" → "Step Over (F6)";
- Quit from the function "Run" → "Step Return (F7)";
- Stop debugging "Run" → "Terminate (Ctrl + F2)";
- Set the breakpoint "Run" → "Toggle Breakpoint (Ctrl + Shift + B)".

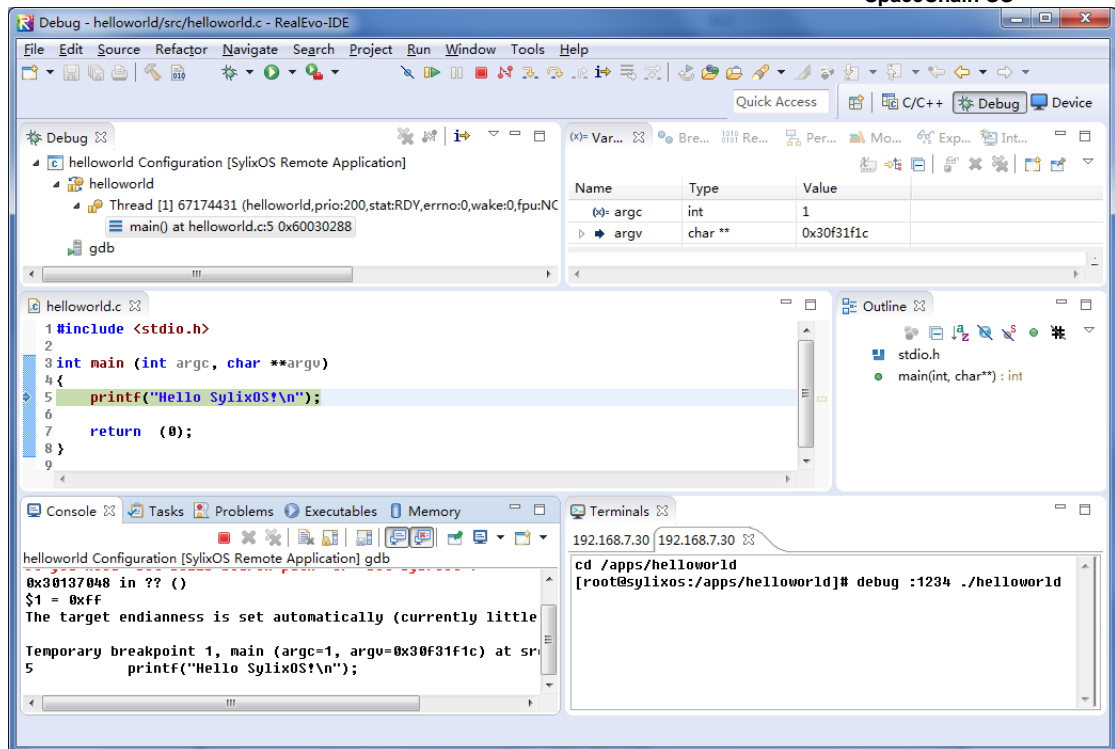


Figure 4.10 Debug operation interface

2. Manually start debugging

To manually start debugging, it is required to manually start GDB server in SylixOS Shell, and the specific method is as follows:

- Use the **debug** command to start the application (such as helloworld) to be debugged, as shown in Figure 4.11 (for use of the **debug** command, see 3.2.5);

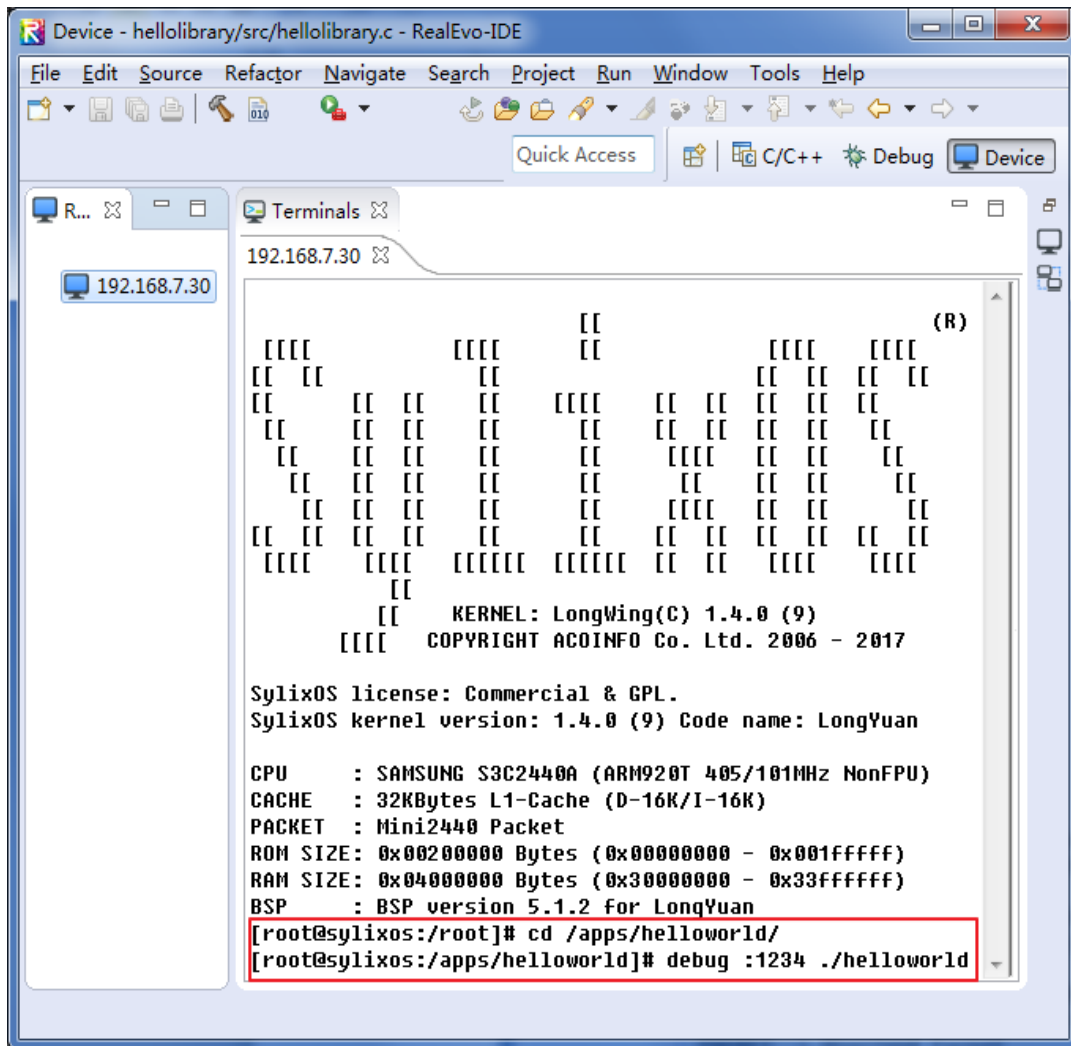


Figure 4.11 Manually start debug

- Select the project to be debugged (such as "helloworld" project), and select the menu "Run" -> "Debug Configurations" to open the debugger interface; Select "SylixOS Remote Application", create a new debugger object, select the "Select other..." button, enable "Use configuration specific settings" in the pop-up box, select "SylixOS Manual Remote App Debugging Launcher" in the drop-down list, click "OK" to return to the "Debug Configurations" dialog box, as shown in Figure 4.12 (the arrow indicates the operation sequence);

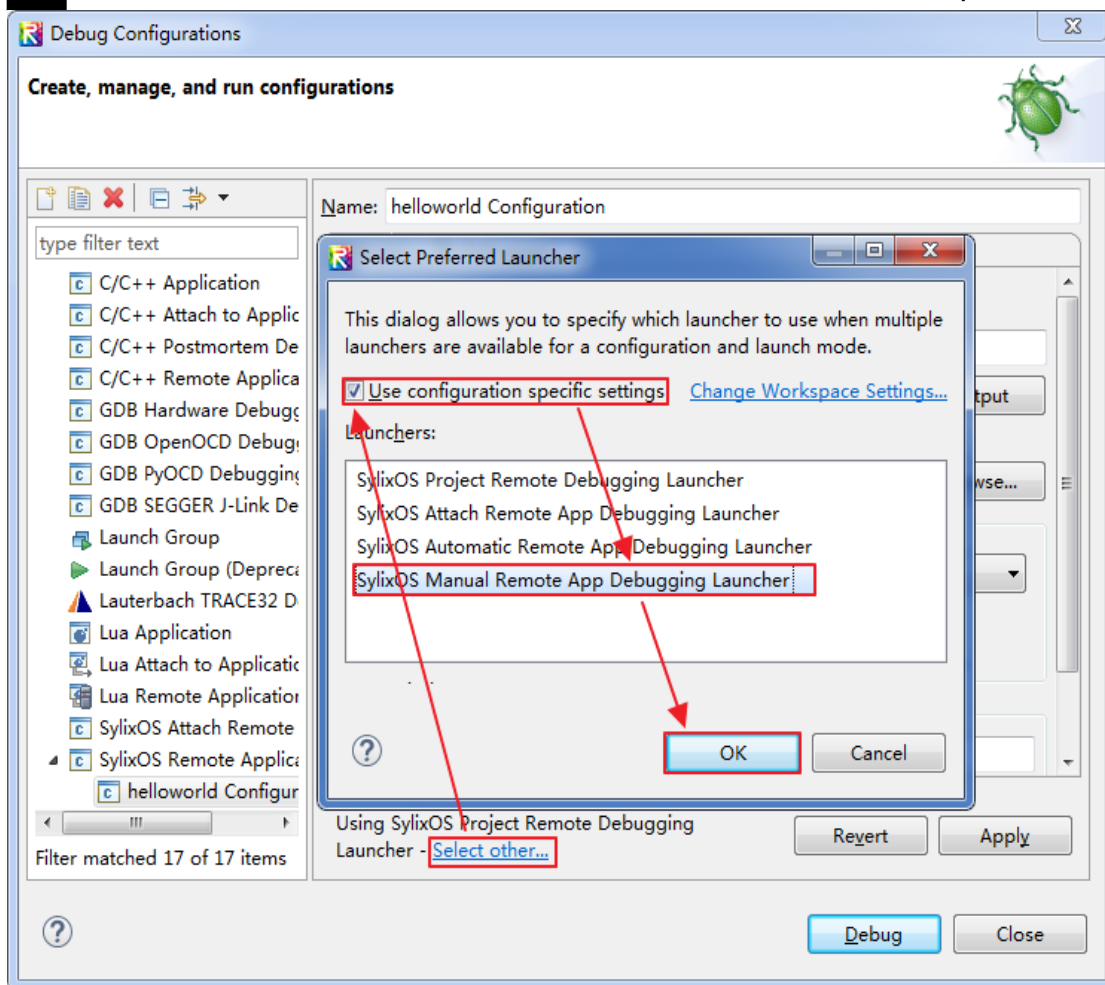


Figure 4.12 Manual debugging of configuration

- Open “Debugger”→“Connection” tab, select “TCP” in the “Type” drop-down box, enter the IP address of SylixOS device in the “Host name or IP address” edit box, and enter the port number in the “Port number” edit box (the port number must be consistent with that when the *debug* command is started. Click "Apply" to apply modification, and click Debug to start debugging, as shown in Figure 4.13

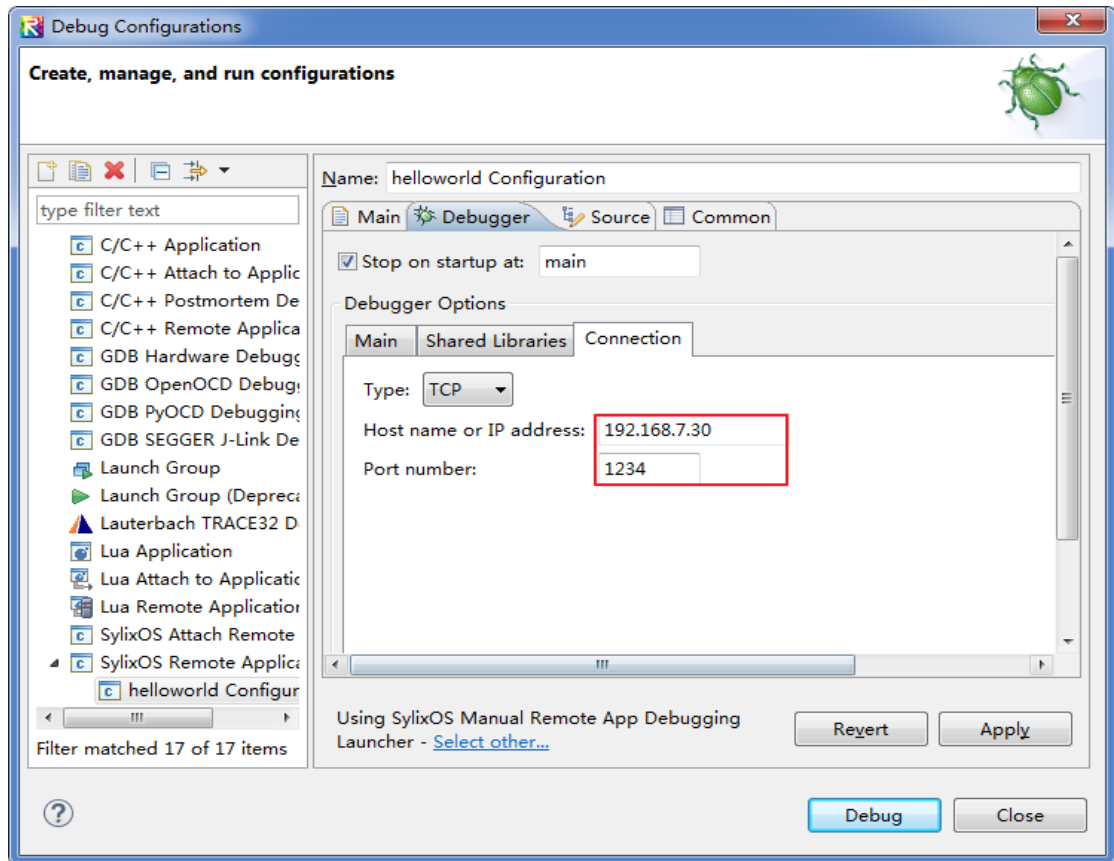


Figure 4.13 Set debug configuration

3. Attach debugging

When the program is running (the program may have been running for a long time), a serious error (the program did not stop because of this error) appears. Because the program cannot stop at the moment, we need a special method to find the problem when the program is running. The Attach debugging method is a method which can debug the run the program. The debugging method exactly satisfies the above conditions. SylixOS supports the Attach debugging method. The following introduces how to use the Attach method to debug the SylixOS application:

- Modify the helloworld project code, as shown in program list 4.1;
- Start the application manually, as shown in Figure 4.14, and use the *ps* command to view the process ID;

Program List 4.1 helloworld project code

```
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
```

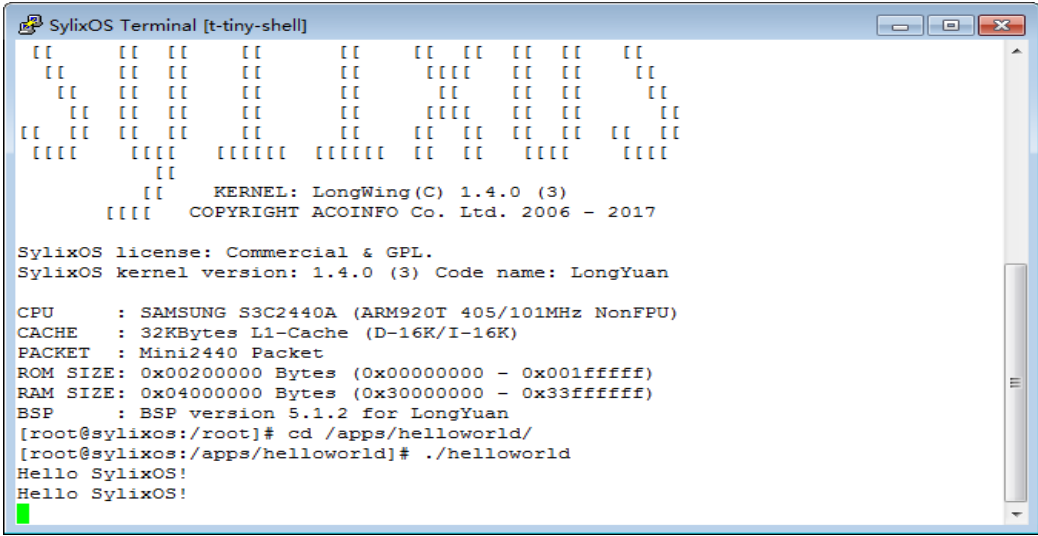
```

{
while (1) {
    printf("Hello SylixOS!\n");
    sleep(1);
}

return (0);
}

```

Note: the program code is added in the program code, and the purpose is to let the program run continuously. sleep is a sleep function, which will make the program run with time delay by a second, so that other threads of SylixOS have the opportunity to be executed.



```

SylixOS Terminal [t-tiny-shell]
[[ [[ [[ [[ [[ [[ [[ [[ [[ [[
[[ [[ [[ [[ [[ [[ [[ [[ [[ [[
[[ [[ [[ [[ [[ [[ [[ [[ [[ [[
[[ [[ [[ [[ [[ [[ [[ [[ [[ [[
[[[[ [[[[ [[[[ [[[[ [[ [[ [[ [[ [[ [[
[[
[[ KERNEL: LongWing(C) 1.4.0 (3)
[[[[ COPYRIGHT ACOINFO Co. Ltd. 2006 - 2017

SylixOS license: Commercial & GPL.
SylixOS kernel version: 1.4.0 (3) Code name: LongYuan

CPU      : SAMSUNG S3C2440A (ARM920T 405/101MHz NonFPU)
CACHE   : 32KBytes L1-Cache (D-16K/I-16K)
PACKET  : Mini2440 Packet
ROM SIZE: 0x00200000 Bytes (0x00000000 - 0x001fffff)
RAM SIZE: 0x04000000 Bytes (0x30000000 - 0x33ffffff)
BSP     : BSP version 5.1.2 for LongYuan
[root@sylixos:/root]# cd /apps/helloworld/
[root@sylixos:/apps/helloworld]# ./helloworld
Hello SylixOS!
Hello SylixOS!

```

Figure 4.14 Helloworld program

Select the "helloworld" project, and select the menu "Run" → "Debug Configurations" to open the debugger interface. Select "SylixOS Remote Application", create a new debugger object, the default name, select the "Select other..." button, enable "Use configuration specific setting" in the pop-up box, select "SylixOS Attach Remote App Debugging Lancher" in the drop-down list, click "OK" to return to the "Debug Configurations" dialog box, as shown in Figure 4.15 (the arrow indicates the operation sequence);

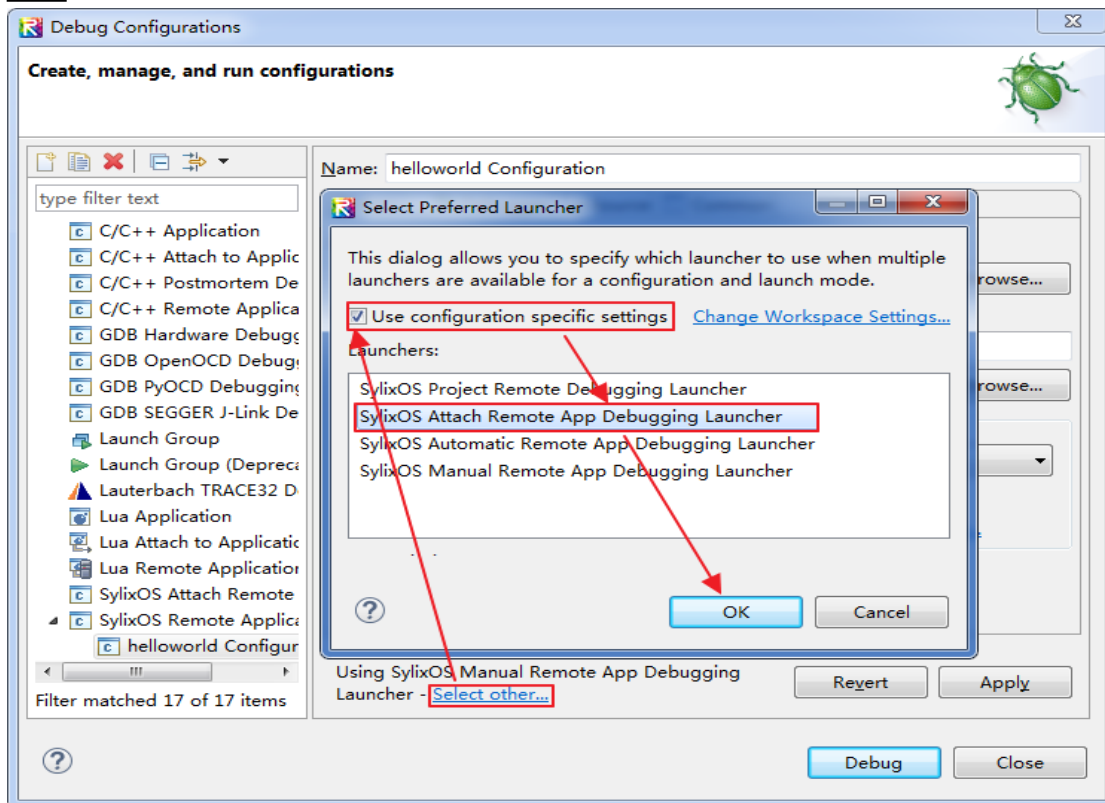


Figure 4.15 Attach debugging

Set “Target process ID”. This option is filled in with the process ID previously viewed. Finally, click “Apply” to complete setting.

4.1.7 Non-stop debugging mode

During multi-thread debugging, it is often necessary to debug a specific thread. Non-stop mode allows the debugger to stop only the thread set with breakpoint when it encounters a breakpoint, and the other threads continue to run. In SylixOS system, the three debugging modes described in Section 4.1.6 all support Non-stop mode. The setting method is as follows: select "Run" → "Debug Configurations" dialog box, open "Debugger" → "Main" property page, enable "Non-stop mode" to open Non-stop mode, as shown in Figure 4.16.

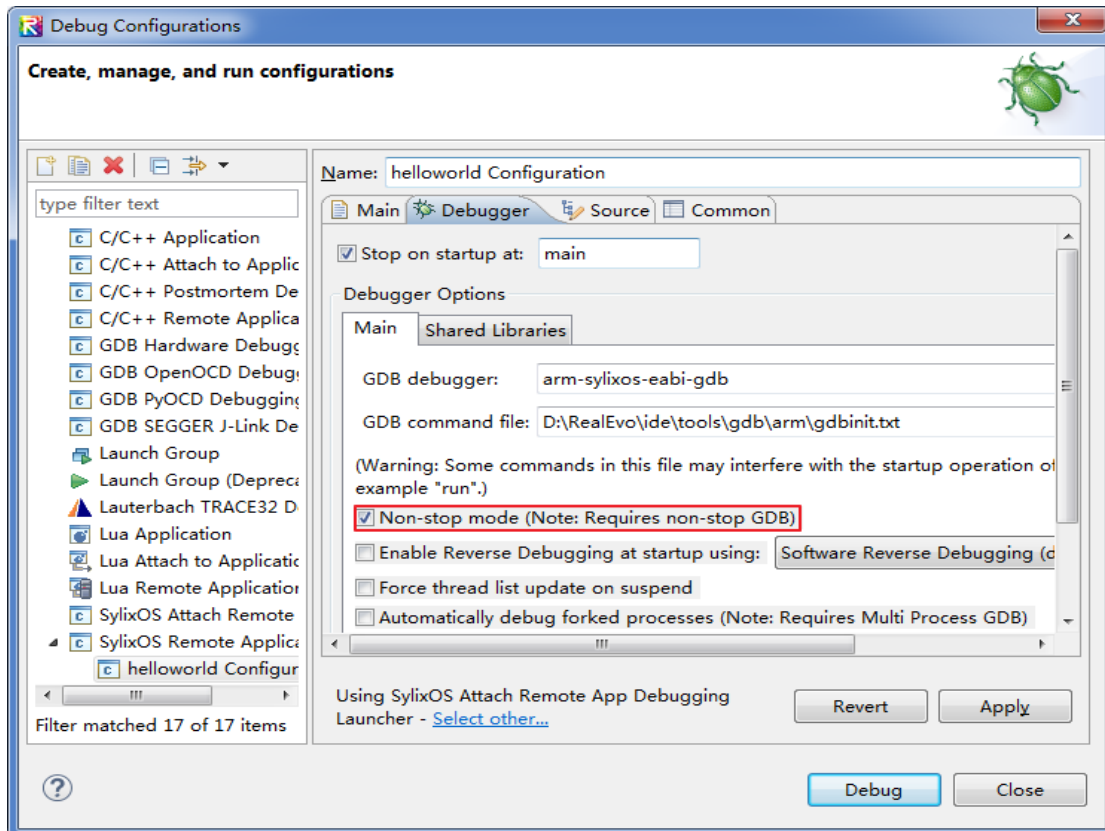


Figure 4.16 Non-stop mode

4.2 Hello Library

In essence, the library is a binary form of executable code which can be loaded into the memory for execution for the operating system. SylixOS's libraries are divided into static libraries and shared libraries (also called dynamic libraries or dynamic link libraries). The difference is that the library code is loaded at different times:

- The static library is linked to the object code during program compilation (the file extension name is usually .a). When the program is running, it is no longer necessary to dynamically load the library. If multiple applications requires the the same static library, each application code part is linked to this static library. Therefore, the application code has large size;
- The dynamic library will not be linked to the object code (the file extension name is usually .so) when the program is compiled, but is only loaded when the program runs. Therefore, the dynamic library file shall be stored in the target system when the program is running. When multiple applications use the same dynamic library, the dynamic library is shared by several application processes in the memory. Therefore, the application code has small size.

4.2.1 Create Hello Library Project

The method for creating a library project is similar to that for creating an application project. The difference is that you need to select "SylixOS Shared Lib" on the project template, as shown in Figure 4.17. Click "Finish" to complete creation of the library project.

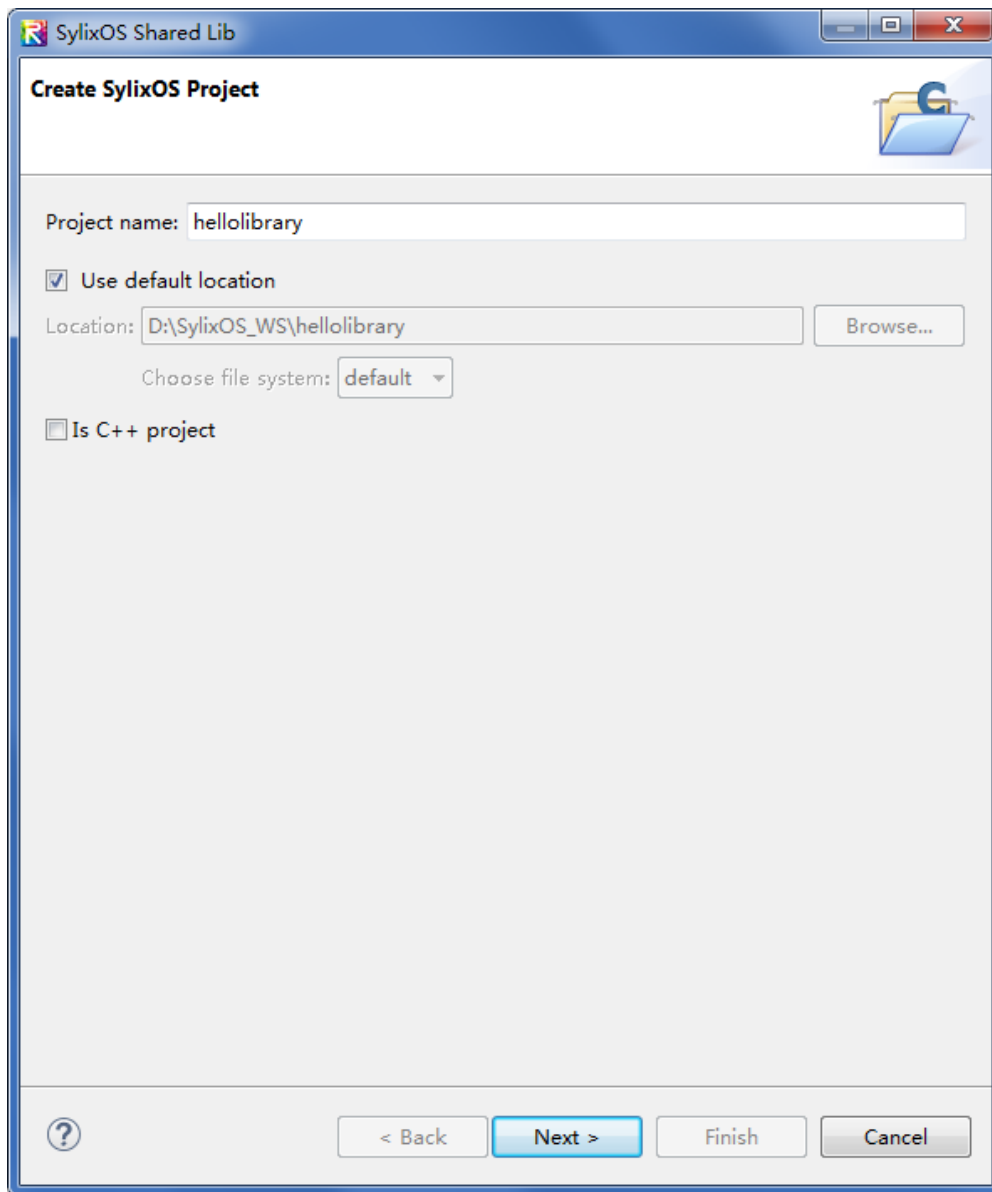


Figure 4.17 Library project settings

4.2.2 Compile Hello Library Project

The compilation method is the same with the method to compile helloworld project.

After compilation, the file list shown in Figure 4.18 will be generated after compilation.

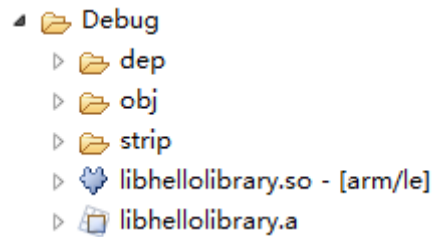


Figure 4.18 List of library files

4.2.3 Deploy the library file

The compiled result file is used to deploy the library file to the target system according to the method described in Subsection 4.1.4. SylixOS shared library file is usually stored in the directory specified under /lib of the SylixOS system or the environment variable LD_LIBRARY_PATH. The corresponding adjustment can be made according to the specific situation.

4.2.4 Modify Hello World application

This section describes how SylixOS application uses the dynamic library and makes the following changes based on the helloworld project:

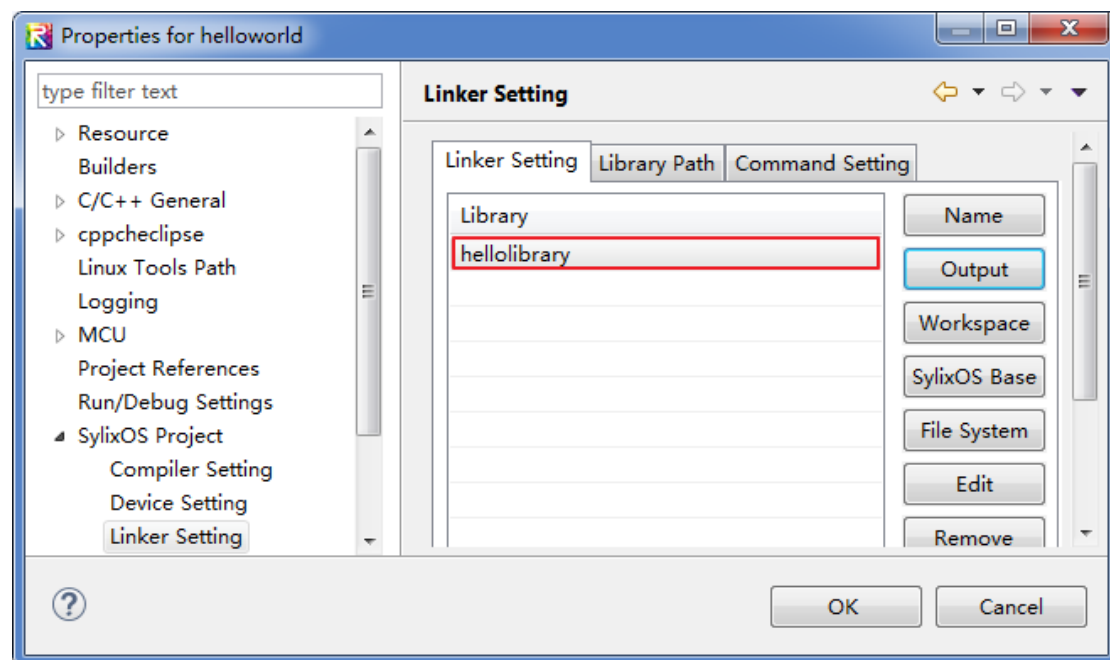


Figure 4.19 Link library settings

Note: Modify the helloworld source file, as shown in Program List 4.2;

Program List 4.2 helloworld code

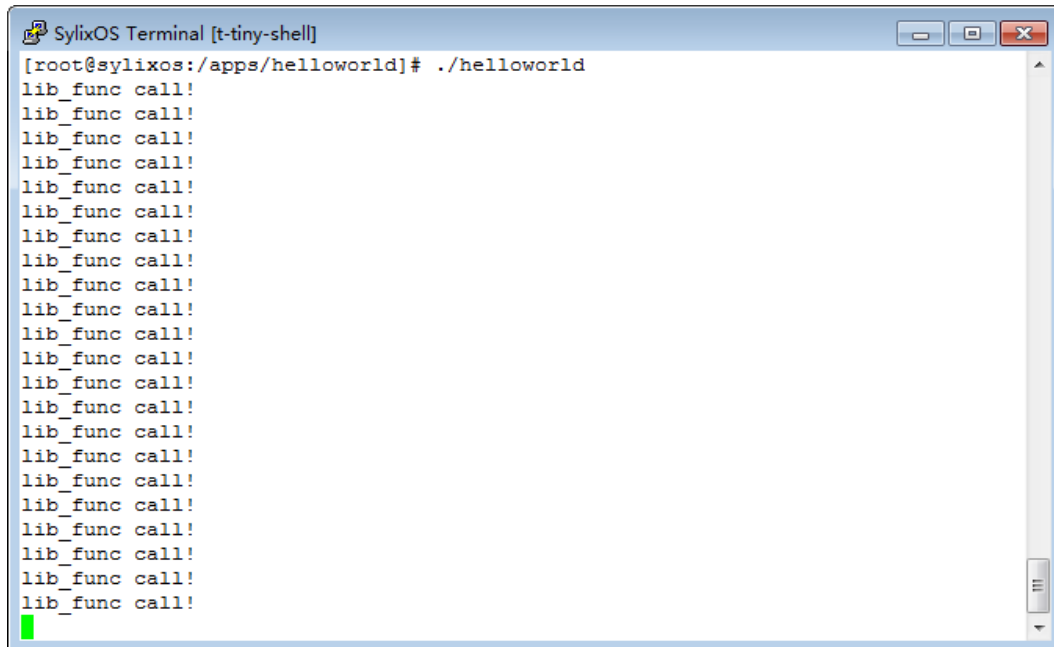
```
#include <unistd.h>
#include <stdio.h>

extern void lib_func_test(void);

int main (int argc, char *argv[])
{
    while (1) {
        lib_func_test();
        sleep(1);
    }
    return (0);
}
```

4.2.5 Run the Hello world Application

Recompile the code modified in the previous section, and then upload the result file generated to the SylixOS device and run it. Observe the running running, as shown in Figure 4.20.



```
SylixOS Terminal [t-tiny-shell]
[root@sylixos:/apps/helloworld]# ./helloworld
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
lib_func call!
```

Figure 4.20 Running results

4.2.6 Debug Hello world applications and dynamic library

In the debugging process, sometimes it is required to jump to the library function for further analysis of the program. At the moment, one must use RealEvo-IDE dynamic library debugging to complete it. In order to achieve dynamic library debugging, RealEvo-IDE shall be subject to following settings.

Select "helloworld" project, open "Debug Configurations" dialog box, open "Debugger" tab, select "Add library paths from project setting" at "Library Paths Setting", click "Apply" button to complete settings, as shown in Figure 4.21.

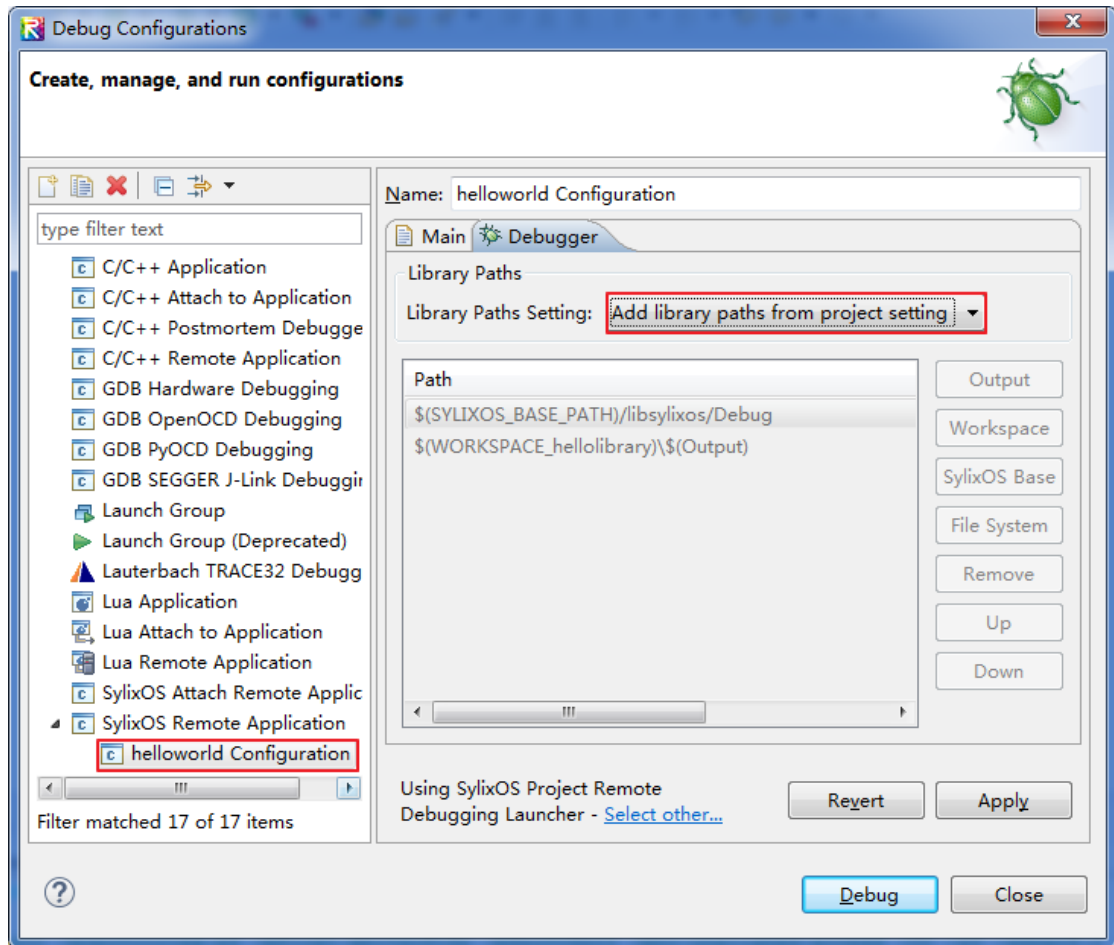


Figure 4.21 Settings of dynamic library debugging

Chapter 5 I/O System

5.1 I/O System

I/O system is also called as the input / output system. SylixOS is compatible with the POSIX standard input / output system. I/O concept of SylixOS inherits I/O concept of UNIX operating system, which considers everything as files. Same with UNIX operating system, files in SylixOS are also divided into different types.

5.1.1 File type

The regular files and directory files are the most common in SylixOS system, but there are other special file types, including the following:

- Regular file, it is the most common file type, and such files contain certain forms of data. Such data, regardless of plain text or binary, makes no difference to SylixOS. It shall be noted that the kernel must understand its format for a binary executable file. The binary executable file of SylixOS follows a standardized format, which allows SylixOS to determine the program code and data loading location (for details, see Chapter 19 Dynamic Loading);
- Directory files, which contain the names of other files and pointers pointed at the information related with these files;
- Block device files. The I/O interface standards provided by such files conform to SylixOS's definition of block devices.
- Character device file, which is a standard unbuffered device file. The most common device file in the system is the character device file;
- FIFO file. This type of file is used for inter-process communication, and it sometimes is also called as named pipe;
- Socket files, which can be used for inter-process network communication (for detailed introduction, see Chapter 15 Network I/O);
- Symbolic link, this type of file points to another file.

The information of file type is contained in the `st_mode` member of the `stat` structure. It can be judged through the macros shown in Table 5.1, and the parameters of these macros are the type values of the member `st_mode`.

Table 5.1 File type

Macro name	File type
------------	-----------

S_ISDIR(mode)	Directory file
S_ISCHR(mode)	Character device file
S_ISBLK(mode)	Block device file
S_ISREG(mode)	Regular file
S_ISLNK(mode)	Symbolic link file
S_ISFIFO(mode)	Pipe or a named pipe
S_ISSOCK(mode)	Socket file

Here is an example of the print file type:

Program List 5.1 Print file type

```
#include <stdio.h>
#include <sys/stat.h>

int main (int argc, char *argv[])
{
    struct stat  mystat;
    int          ret;

    if (argc < 2) {
        fprintf(stderr, "argc error.\n");
        return (-1);
    }

    ret = stat(argv[1], &mystat);
    if (ret < 0) {
        perror("stat");
        return (-1);
    }

    if (S_ISDIR(mystat.st_mode)) {
        fprintf(stdout, "file: %s is dir file.\n", argv[1]);
    }

    if (S_ISCHR(mystat.st_mode)) {
        fprintf(stdout, "file: %s is char file.\n", argv[1]);
    }

    if (S_ISBLK(mystat.st_mode)) {
        fprintf(stdout, "file: %s is block file.\n", argv[1]);
    }
}
```

```
    if (S_ISREG(mystat.st_mode)) {
        fprintf(stdout, "file: %s is general file.\n", argv[1]);
    }

    if (S_ISLNK(mystat.st_mode)) {
        fprintf(stdout, "file: %s is link file.\n", argv[1]);
    }

    if (S_ISSOCK(mystat.st_mode)) {
        fprintf(stdout, "file: %s is socket file.\n", argv[1]);
    }

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./type_test /dev/socket
file: /dev/socket is socket file.
# ./type_test /dev/null
file: /dev/null is char file.
```

Program List 5.1 uses the stat function. The detailed information of this function will be introduced in Subsection 5.2.2. It can be seen from the program results that the file type can be judged by the macros shown in Table 5.1. The program also displays how to use macros in Table 5.1.

5.1.2 File descriptor

For the kernel, all open files are referenced through the file descriptor. The file descriptor is a non-negative integer. When an existing file is opened or a new file is created, the kernel returns a file descriptor to the process. When reading or writing a file, use the file descriptor returned by the open function or the creat function to identify the file. This file descriptor can be transferred to the read function or the write function as a parameter.

SylixOS file descriptor is compatible with POSIX definition. It is an integer number (`_POSIX_OPEN_MAX`) starting from 0 up to a maximum value. Each open file has one or more (dup) file descriptors for matching. SylixOS, like the majority of operating systems, always uses a minimal and unused file descriptor as a newly allocated file descriptor when opening a file.

According to the habit, 0 file descriptor represents standard input, 1 file descriptor represents standard output, and 2 file descriptor represents standard error. It shall be noted here that each process of SylixOS has its own file descriptor table, and each

process does not conflict with each other. If a child process has a parent process, all file descriptors of the parent process will be inherited; if the child process is an orphan process, only 3 standard file descriptors of the system will be inherited. All threads within a process share the process file descriptor. There is a global file descriptor table in the kernel. This file descriptor table does not contain 0, 1 and 2 standard files. These three file descriptors are remapping flags in the kernel, that is to say, SylixOS allows each kernel task in the kernel has its own standard file.

In POSIX.1-compliant applications, 0, 1, and 2 have been standardized, but they shall be replaced with symbolic constants `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`, so as to improve readability. In SylixOS, one can use these constants by including `<unistd.h>` header file.

5.1.3 I/O System structure

The I/O system structure of SylixOS is divided into ORIG drive structure and NEW_1 drive structure for historical reasons. The NEW_1 drive structure is added with file access permission, file record lock and other functions based on ORIG drive structure.

Figure 5.1 shows the diagram of ORIG drive structure of SylixOS:

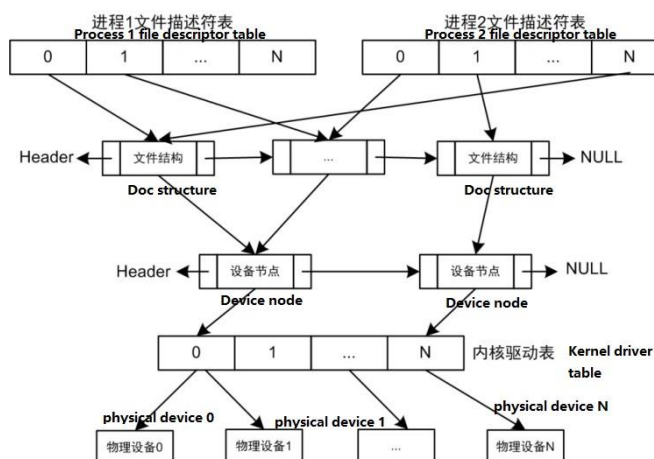


Figure 5.1 ORIG drive structure

Each file descriptor in SylixOS corresponds to a file structure. Different file descriptors may correspond to the same file structure. When all file descriptors corresponding to the same file structure are closed, the operating system will release the corresponding file structure, and call the corresponding driver. Different file structures can point to the same logical device. For example, a FAT file system device can be opened with many file structures. Different logical devices can also correspond to the same driver. For example, serial port 0 and serial port 1 with the same physical structure can correspond to a group of drivers for their services. The hardware device for specific service of each group of

drivers is determined by the bottom BSP.

Figure 5.2 shows the diagram of SylixOS NEW_1 driver structure:

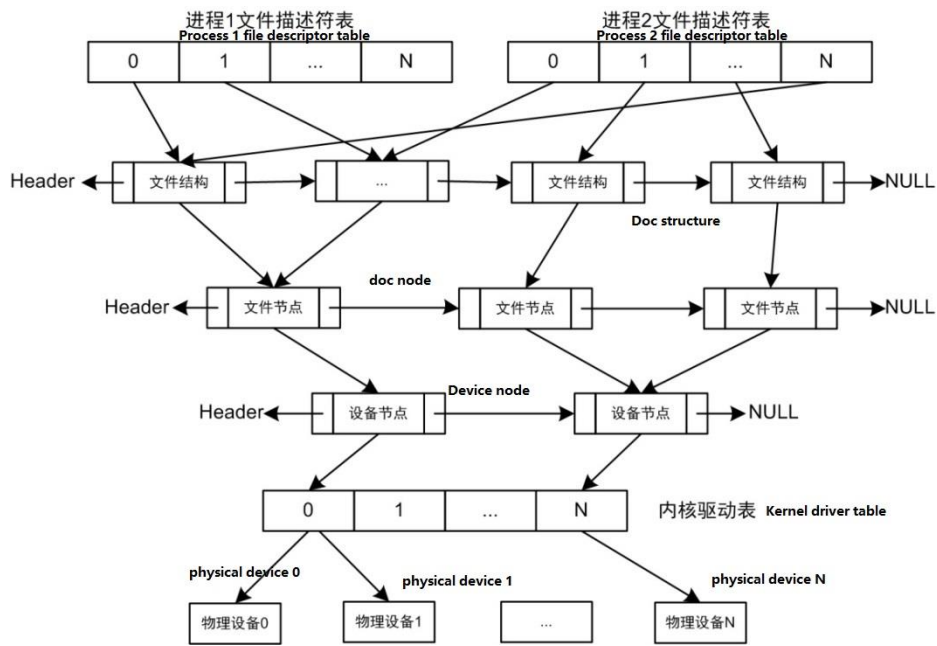


Figure 5.2 NEW_1 driver structure

The NEW_1 driver structure is added with file nodes based on ORIG. Therefore, file access permission, file user information, file record locks (described in detail in Section 5.4.4) are introduced.

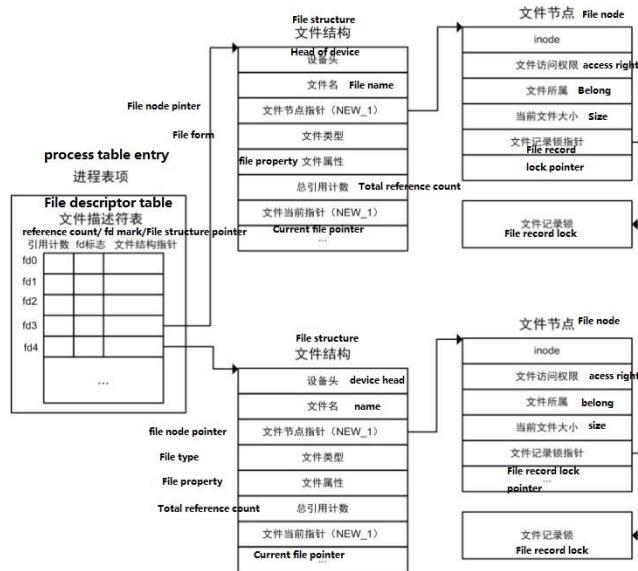


Figure 5.3 NEW_1 kernel data structure

It is found from Figure 5.2 that SylixOS supports sharing open files between different processes. It can be seen from Figure 5.3 NEW_1 kernel data structure that SylixOS kernel uses three data structures (file descriptor item, file structure and file node) to represent the open file, and their relationship determines possible influence of a process on another process in the aspect of file sharing.

- Each process maintains a file descriptor table of its own. Each file descriptor occupies one item, and those associated with each file descriptor are:
 - ◆ Pointer to the file structure;
 - ◆ File reference count;
 - ◆ File descriptor flag (FD_CLOEXEC).
- The kernel maintains a file structure table for all open files, and entries of each file structure table includes (partial):
 - ◆ Device head pointer (this pointer points to the device node);
 - ◆ File name;
 - ◆ File node pointer;
 - ◆ File attribute flags (read, write, etc. See Table 5.2 for more information);
 - ◆ File current pointer (indicating file offset).
- Each open file has a file node, and the file node includes (partial):
 - ◆ Device descriptor;
 - ◆ Inode^① (there is only one inode in the same file);
 - ◆ File permissions information (readable, writable and executable);
 - ◆ File user information;
 - ◆ Current file size;
 - ◆ File record lock pointer.

Figure 5.3 shows the relationship among the three data structures corresponding to a process. The process opens two different files, one opened from file descriptor 3 and the other opened from file descriptor 4.

If two independent processes open the same file respectively, the relation is shown in Figure 5.4.

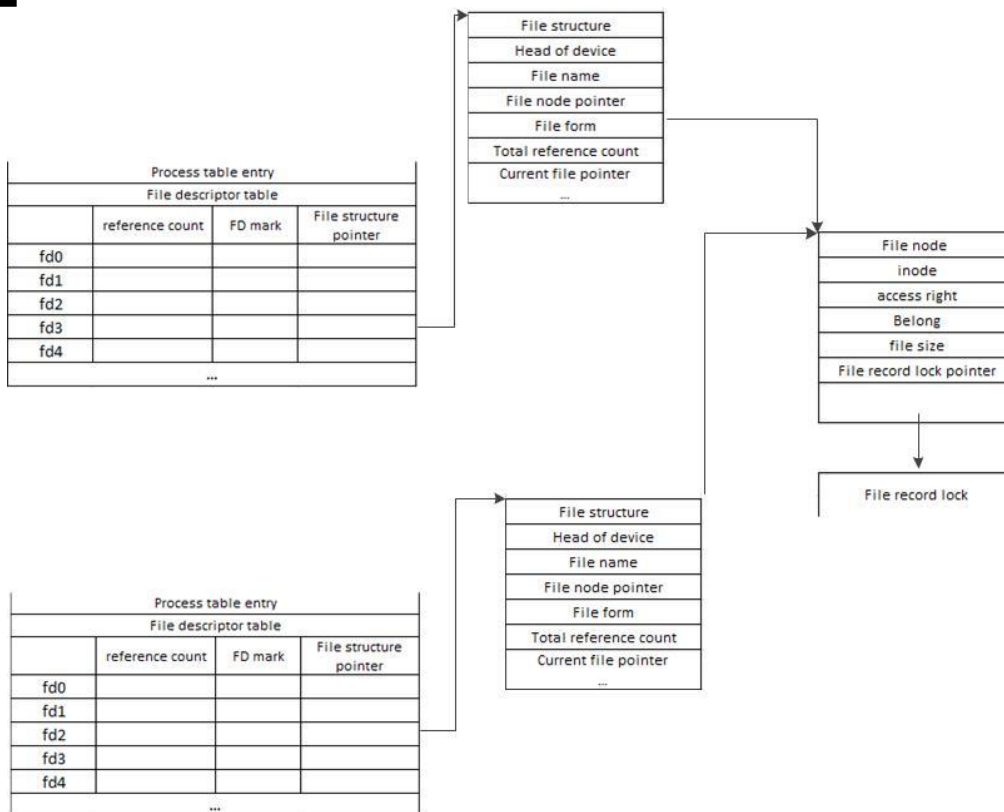


Figure 5.4 Two independent processes open the same file respectively

We assume that the first process opens the file on file descriptor 3, while another process opens the same file on file descriptor 4. Each process which opens the file gets its own file structure, but there is only one file node for a given file. Each process gets its own file structure because it allows each process to have its own current read / write pointer to the file (file operation offset).

The file descriptor flag and the file attribute flag are different in terms of action scope. The former only applies to one file descriptor in a process, while the latter applies to all file descriptors in any process which points to the given file structure. In Section 5.2.1, we will describe how to call the `fcntl` function to obtain and modify the file descriptor flag and the file attribute flag.

5.2 Standard I/O access

Standard I/O is also called as synchronous I/O, i.e., initiating transmission and controlling I/O are both user-active actions. The file or device must operate with user's intervention. The majority of applications currently use this type of I/O operation. SylixOS supports the majority of synchronous I/O operations specified by POSIX. We will describe in detail operation of files and directories in SylixOS as follows.

5.2.1 File I/O

1. Function open

```
#include <fcntl.h>
int open(const char *cpcName, int iFlag, ...);
```

Prototype analysis of Function open:

- For success of the function, return the file descriptor. For failure, return -1 and set the error number;
- Parameter **cpcName** is the name of the file to be opened^①;
- Parameter **iFlag** is the open file flag;
- Parameter... is a variable parameter.

Call the open function to open or create a file. The last parameter of the open function is written as ..., ISO C uses this method to represent that the number of remaining parameters and its type are variable. For open function, this parameter will only be used when the new file is created.

Parameter **iFlag** contains multiple options, as shown in Table 5.2. This parameter is usually composed by adding "or" between multiple options.

Table 5.2 iFlag options

Option name	Note
O_RDONLY	Open the file in a read-only manner
O_WRONLY	Open the file in a write-only manner
O_RDWR	Open the file in a readable or writable manner
O_CREAT	If the file does not exist, the file is created, and the third parameter of open function specifies the file permission mode
O_TRUNC	If the file exists and it is opened successfully in write-only or read-write mode, its length is truncated to 0
O_APPEND	Append the read-write pointer to the end of the file
O_EXCL	If O_CREAT is specified and the file exists, an error occurs. If the file does not exist, create it
O_NONBLOCK	Open the file in a non-blocking manner
O_SYNC	Enable each write to wait for physical I/O operation to complete, including file attribute updates caused by the write operation.
O_DSYNC	Make each write wait for physical I/O operation to complete. However, if the write operation does not affect reading the data just written, you must not wait for the file attributes to be updated
O_NOCTTY	If cpcName quotes the terminal device, the device will not be assigned as control terminal for this process
O_NOFOLLOW	If cpcName quotes a symbolic link, an error occurs

O_CLOEXEC	Set FD_CLOEXEC flag as the file descriptor flag
O_LARGEFILE	Open big file flag

The file descriptor returned by the open function must be the smallest and unused descriptor value in the system. This is used by some applications to open new files on standard input, standard output or standard error. For example, an application can first close standard output (file descriptor 1) and then open another file. Before executing the open operation, it can be understood that the file will be opened on file descriptor 1. When explaining dup2, you will learn that there is a better way to open a file on a given file descriptor.

I/O system of SylixOS supports up to 2TB file. However, limited by certain file system design, the FAT file system, for example, can only support 4GB file size at maximum. In an application, in order to explicitly specify to open a large file, the O_LARGEFILE flag shall be specified when the open function is called. SylixOS also provides the following functions to open large files.

```
#include <fcntl.h>
int open64(const char *cpcName, int iFlag, ...);
```

Function open64 prototype analysis:

- For success of the function, return the file descriptor. For failure, return -1 and set the error number;
- Parameter **cpcName** is the name of the file to be opened;
- Parameter **iFlag** is the open file flag;
- Parameter... is a variable parameter.

2. Function creat

```
#include <fcntl.h>
int creat(const char *cpcName, int iMode);
```

Function creat prototype analysis:

- For success of the function, return the file descriptor. For failure, return -1 and set the error number;
- Parameter **cpcName** is the name of the file to be created;
- Parameter **iMode** is the mode to create file.

A file can be created by calling creat function. The function is equivalent to the following function call:

```
open(cpcName, O_WRONLY | O_CREAT | O_TRUNC, iMode);
```

The file access permission mode (*iMode*) will be described in detail in Section 5.2.2.

One drawback of creat function is that it opens the created file in a write-only manner. If a file is created with creat function and then it is required to read the file, one must firstly call creat function to create the file, then call close to close the file, and then open the file in a read manner. However, this way can be achieved directly by calling open function:

```
open(cpName, O_RDWR | O_CREAT | O_TRUNC, iMode);
```

3. Function close

```
#include <unistd.h>
int close(int iFd);
```

Prototype analysis of function close:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor.

Calling of close function will reduce the reference count of file descriptors and the total reference count of files by one. When the reference count of file descriptors is zero, the file descriptor will be deleted (one will see the point at introduction to dup function). When the total reference count is reduced to zero, close the file, and release all record locks added by the current process on the file (Section 5.4.4 Introduction to Record Lock).

When a process is terminated, the kernel will automatically close all the files it opens. Many programs take advantage of the function not to explicitly call close function to close the open file.

4. Function read

```
#include <unistd.h>
ssize_t read(int iFd,
             void *pvBuffer,
             size_t stMaxBytes);
```

Prototype analysis of Function read:

- For success of the function, return the number of bytes read. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Output parameter *pvBuffer* is the receive buffer zone;
- Parameter *stMaxBytes* is the size of the receive buffer zone.

The data can be read from the open file by calling read function, and the number of bytes actually read is less than the number of bytes to be read in many cases:

- When the ordinary file is read, the file end has been reached before the requested number of bytes is read;
- For reading from a terminal device, a line is read at most at a time;
- For reading from the network, the buffer mechanism in the network may cause that the return value is less than the number of bytes to be read;
- When it is interrupted by the signal, and some data has been read.

Usually, we need to judge the quantity and correctness of the read data through the read return value.

5. Function write

```
#include <unistd.h>
ssize_t write(int          iFd,
              const void *pvBuffer,
              size_t       stNBytes);
```

Prototype analysis of Function write:

- For success of the function, return the number of bytes written. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *pvBuffer* is the address of the data buffer zone to be written to the file;
- Parameter *stNBytes* is the number of bytes written to the file.

Call Function write to write data to the open file, the return value is usually the same with Parameter *stNBytes* value. Otherwise, it means an error. A common cause for write error is that the disk is full or exceeds the file length limit of a process.

For ordinary files, the write operation is start at the current offset of the file. If the O_APPEND flag is specified when the file is opened, the file offset is set at the file end before each operation. After a successful write, the file offset increases the number of bytes actually written at the file end.

6. Function lseek

Each open file has a current file offset associated with it, which is usually an integer, which measures the number of bytes calculated from the file beginning. In general, both read and write operations are started from the current file offset, and the offset increases the number of bytes read and written. At default of SylixOS, when a file is opened, the current file offset is always set as 0 unless the O_APPEND flag is specified.

```
#include <fcntl.h>
```

```

off_t  lseek(int    iFd,
            off_t    oftOffset,
            int      iWhence);

```

Prototype analysis of Function lseek:

- For success of the function, return the new file offset. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *oftOffset* is the offset;
- Parameter *iWhence* is the location datum.

Calling the lseek function can explicitly set an offset for an open file. It shall be noted that the lseek call just adjusts the file offset records associated with the file descriptor in the kernel, but does not cause access to any physical device.

The meaning of Parameter *oftOffset* varies depending on Parameter *iWhence*, as shown in Table 5.3:

Table 5.3 *iWhence* value correlation

<i>iWhence</i> value	Instructions for <i>oftOffset</i>
SEEK_SET	Set the file offset with <i>oftOffset</i> bytes from the file beginning.
SEEK_CUR	Set the file offset as the current value plus <i>oftOffset</i> bytes, and <i>oftOffset</i> can be negative
SEEK_END	Set the file offset to file length plus <i>oftOffset</i> bytes, and <i>oftOffset</i> can be negative

The meaning of *iWhence* parameter is shown in Figure 5.5.

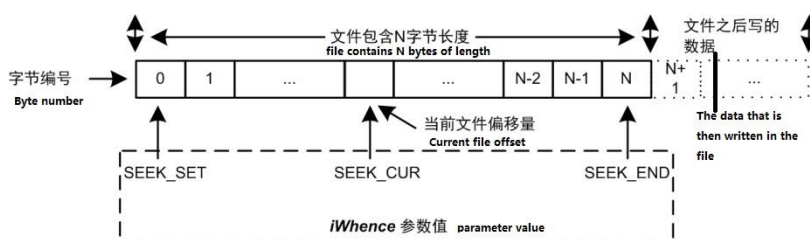


Figure 5.5 Meaning of *iWhence* parameter

Here are some instances of Function lseek call, and the note describes where to move the current file pointer.

```

lseek(fd, 0, SEEK_SET);          /* at the beginning of the file */
lseek(fd, 0, SEEK_END);         /* at the end of the file      */

```

```

lseek(fd, -1, SEEK_END);          /* The first byte of the file is counted
                                  down */
lseek(fd, -20, SEEK_CUR);        /* 20 bytes before the current location
of the file */
lseek(fd, 100, SEEK_END);        /* Extend 100 bytes at the end of the file
                                  */

```

If the file offset of the program has already spanned the file end, perform I/O operation, read function call will return 0, indicating the file end. However, write function can write data at any position behind the file end.

The space from the file end to the newly written data is called as file hole. From the perspective of programming, there is a byte in the file hole, and reading the hole will return a buffer zone filled with 0.

Existence of hole means that the nominal size of a file may be larger than the total disk storage it occupies (sometimes much larger). Of course, the specific processing method is related to implementation of the file system.

The following instance shows use of lseek function. This program creates a new file, generates a file hole by calling lseek function, and then writes some data at the file end, so that the program can read the file hole part, which are non-printable characters. Program print "\0" represents non-printable character.

Program List 5.2 Use instance of Function lseek

```

#include <stdio.h>
#include <fcntl.h>
#include <ctype.h>
#include <unistd.h>

#define FILE_MODE      (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)

int main (int argc, char *argv[])
{
    int      fd, i;
    off_t    ret;
    ssize_t  size;
    char     *buf1 = "sylixos";
    char     buf2[16];

    fd = open("./file", O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
    if (fd < 0) {
        fprintf(stderr, "Create file failed.\n");
        return -1;
    }
}

```



```
write(fd, buf1, 7);
ret = lseek(fd, 1000, SEEK_SET);
if (ret < 0) {
    fprintf(stderr, "lseek failed.\n");
    close(fd);
    return -1;
}

write(fd, buf1, 7);
lseek(fd, -7, SEEK_END);

size = read(fd, buf2, 7);
if (size < 0) {
    fprintf(stderr, "read error.\n");
    close(fd);
    return -1;
}

for (i = 0; i < size; i++) {
    if (!isprint(buf2[i])) {
        fprintf(stdout, "\\0");
    } else {
        fprintf(stdout, "%c", buf2[i]);
    }
}

fprintf(stdout, "\n");
lseek(fd, -14, SEEK_END);

size = read(fd, buf2, 7);
if (size < 0) {
    fprintf(stderr, "read error.\n");
    close(fd);
    return -1;
}

for (i = 0; i < size; i++) {
    if (!isprint(buf2[i])) {
        fprintf(stdout, "\\0");
    } else {
        fprintf(stdout, "%c", buf2[i]);
    }
}
```

```

    }
    fprintf(stdout, "\n");
    close(fd);
    return 0;
}

```

Run this program under the SylixOS Shell, and the program results show that the written data and contents of the file hole part are correctly read.

```

# ./lseek_test
sylixos
\0\0\0\0\0\0\0

```

7. Functions pread and pwrite

Section 5.1.3 introduces that multiple processes in SylixOS can read the same file. Each process has its own file structure, which also has its own current file offset. However, in non-NEW_1 file systems (without unique file node), unexpected results may be generated when multiple processes write the same file. To illustrate how to avoid this situation, one needs to understand concept of atomic operation.

Considering the following code, a file is opened in the process to append data to it.

```

.....
ret = lseek(fd, 0, SEEK_END);
if (ret < 0) {
    fprintf(stderr, "Lseek error.\n");
}

ret = write(fd, buf, 10);
if (ret != 10) {
    fprintf(stderr, "Write data error.\n")
}
.....

```

This code has no problem in the case of single process, and the fact also proves this. However, if there are multiple processes, problems will be caused by using this method to append data to the file.

If there are two independent processes 1 and 2 to simultaneously append the write operation to a file, each process does not use the O_APPEND flag when opening the file. At this time, the relation of each data structure is shown in Figure 5.4. Each process is has its own file structure and file current offset, but a file node is shared. If process 1 calls lseek function to set the current file offset to the file end, then process 2 runs, calls lseek function, and also sets the current file offset to the file end. Then process 2 calls write function to push the file offset of process 2 backward by 10 bytes. At this time, the file becomes longer, and the kernel also increases the file length in the file node by 10 bytes.

After that, the kernel switches process 1 for running, and calls the write function. At this point, process 1 starts to write from its own current offset, thus overwriting the data that process 2 just wrote.

As can be seen from the above process, the problem lies in the "position to the file end, and then write the file", this process is completed with two functions. Therefore, it causes a non-atomic operation, because the process may be switched between two functions. Therefore, we can conclude that if the process is completed in a function (forming an atomic operation), the problem can be solved.

SylixOS provides an atomic operation method for this operation. When the file is opened, the O_APPEND flag is set, so that the kernel will set the current offset to the file end for each write operation, and it is not required to call lseek function before each write.

SylixOS provides an atomic function to locate and perform I/O operations: pread, pwrite^①.

```
#include <unistd.h>
ssize_t pread(int      iFd,
              void     *pvBuffer,
              size_t   stMaxBytes,
              off_t    oftPos);
ssize_t pwrite(int    iFd,
               const void *pvBuffer,
               size_t    stNBytes,
               off_t     oftPos);
```

Prototype analysis of Function pread:

- For success of the function, return the number of bytes read. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Output parameter *pvBuffer* is the receive buffer zone;
- Parameter *stMaxBytes* is the size of the buffer zone;
- Parameter *oftPos* appoints the read position.

Prototype analysis of Function pwrite:

- For success of the function, return the number of bytes written. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *pvBuffer* is the data buffer zone;
- Parameter *stNBytes* is the number of bytes written;

• appoints the writing position.

Calling pread function is equivalent to calling lseek function before calling read function, but pread function has the following important differences from this sequence:

- When calling the pread function, its positioning and read operations cannot be interrupted (atomic operation process);
- Do not update the current file offset.

Calling pwrite function is equivalent to calling write function after calling lseek function first, but there are also similar differences from the above.

In general, atomic operation refers to one operation consisting of multiple steps. If the operation is performed atomically, either all steps are performed or no step is performed, it is not possible to perform only a subset of all steps.

To be able to read and write larger files (usually larger than 4GB), SylixOS provides the following group of functions.

```
#include <unistd.h>
ssize_t pread64(INT      iFd,
                PVOID    pvBuffer,
                size_t    stMaxBytes,
                off64_t   oftPos);
ssize_t pwrite64(INT     iFd,
                 CPVOID  pvBuffer,
                 size_t   stNBytes,
                 off_t64  oftPos);
```

Prototype analysis of Function pread64:

- For success of the function, return the number of bytes read. For failure, return -1 and set the error number;
- Parameter ***iFd*** is the file descriptor;
- Output parameter ***pvBuffer*** is the receive buffer zone;
- Parameter ***stMaxBytes*** is the size of the buffer zone;
- Parameter ***oftPos*** appoints the read position.

Prototype analysis of Function pwrite6:

- For success of the function, return the number of bytes written. For failure, return -1 and set the error number;
- Parameter ***iFd*** is the file descriptor;

- data buffer zone;

- Parameter ***stNBytes*** is the number of bytes written;
- Parameter ***oftPos*** appoints the writing position.

The following example shows how to use `pwrite` function and `pread` function. This program creates a new file, writes data to the specified offset of the file by calling `pwrite` function, and then calls `read` function to verify the current offset of the file. Calling `pread` function also verifies that the file hole is generated.

Program List 5.3 Use of `pwrite` and `pread` functions

```
#include <stdio.h>
#include <fcntl.h>
#include <ctype.h>
#include <unistd.h>

#define FILE_MODE      (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)

int main (int argc, char *argv[])
{
    int          fd, i;
    ssize_t      size;
    char         *buf1 = "sylixos";
    char         buf2[16];

    fd = open("./file", O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
    if (fd < 0) {
        fprintf(stderr, "Create file failed.\n");
        return (-1);
    }

    size = pwrite(fd, buf1, 7, 100);
    if (size != 7) {
        fprintf(stderr, "pwrite error.\n");
        close(fd);
        return -1;
    }

    size = read(fd, buf2, 7);
    if (size < 0) {
        fprintf(stderr, "read error.\n");
        close(fd);
    }
}
```

```
        return (-1);
    }

    for (i = 0; i < size; i++) {
        if (!isprint(buf2[i])) {
            fprintf(stdout, "\\0");
        } else {
            fprintf(stdout, "%c", buf2[i]);
        }
    }
    fprintf(stdout, "\n");

    size = pread(fd, buf2, 7, 100);
    if (size < 0) {
        fprintf(stderr, "pread error.\n");
        close(fd);
        return (-1);
    }

    for (i = 0; i < size; i++) {
        if (!isprint(buf2[i])) {
            fprintf(stdout, "\\0");
        } else {
            fprintf(stdout, "%c", buf2[i]);
        }
    }
    fprintf(stdout, "\n");
    close(fd);
    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./pwrite_test
\0\0\0\0\0\0\0
sylixos
```

The print results show that the file hole is generated after pwrite function was called, and the current file offset has not changed. This also confirms that the aforementioned pwrite function is equivalent to calling lseek function before function write, but the difference is that pwrite is an atomic operation.

8. Functions dup and dup2

```
#include <unistd.h>
int dup(int iFd);
int dup2(int iFd1, int iFd2);
```

Prototype analysis of Function dup:

- For success of the function, return the file descriptor. For failure, return -1 and set the error number;
- Parameter **iFd** is the original file descriptor.

Prototype analysis of Function dup2:

- For success of the function, return **iFd2** file descriptor. For failure, return -1 and set the error number;
- Parameter **iFd1** is the file descriptor 1;
- Parameter **iFd2** is the file descriptor 2;

Calling dup function and dup2 function can copy an existing file descriptor. The new file descriptor returned by the dup function must be the smallest value in the currently available file descriptor. For dup2 function, one can use Parameter **iFd2** to specify the value of the new file descriptor. If **iFd2** has been already opened, it shall be closed firstly, and FD_CLOEXEC file descriptor flag of **iFd2** will be cleared, so that **iFd2** is open when the process calls exec function (see Chapter 8 Process Management). Note that the SylixOS kernel does not currently support the situation that **iFd1** is equal to **iFd2**.

The file descriptor returned by dup function and Parameter **iFd** share the same file structure item (file table entry). Similarly, the file descriptors **iFd1** and **iFd2** of dup2 function also share the same file structure item, as shown in Figure 5.6.

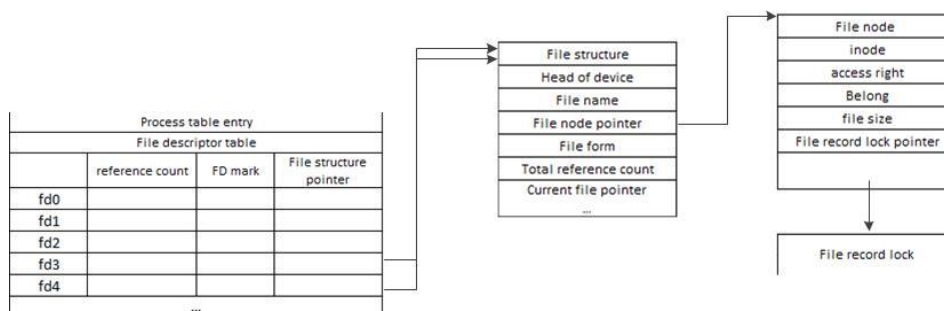


Figure 5.6 Kernel data structure behind dup(3)

In Figure 5.6, the process calls:

```
fd = dup(3);
```

Assume that file descriptor 3 has already been occupied (which is very likely). At the

moment, we call dup function to use file descriptor 4, because both file descriptors point to the same file structure (file table entry). Therefore, they share the same file attribute flag (read, write, append, etc.) and the same file current pointer (file offset).

Each file has its own set of file descriptor flags, and the new file descriptor flag (FD_CLOEXEC) is always cleared by dup function.

Another way to copy the descriptor is to use fcntl function. The following subsection introduces the function, in fact, calling

```
dup(fd);
```

Equivalent to

```
fcntl(fd, F_DUPFD, 0);
```

While, calling

```
dup2(fd, fd2);
```

Equivalent to

```
fcntl(fd, F_DUP2FD①, fd2);
```

Or

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

As mentioned above, each process of SylixOS has its own file descriptor table, and there is also a global file descriptor table in the kernel. Then if a file is opened in the process, the kernel cannot see the file descriptor, but there are some file descriptors to be opened by the kernel operating process (for example: write data to the file specified by the application in the log system). SylixOS provides the following functions to realize copying of the process file descriptor to the kernel space.

```
#include <unistd.h>
int dup2kernel(int fd);
```

Prototype analysis of Function dup2kernel:

- For success of the function, return the kernel file descriptor. For failure, return -1 and set the error number;
- Parameter *fd* is the process file descriptor.

The following program shows how to use the dup function. The program creates a new file, calls dup function to copy a new file descriptor, and then manipulates the created file on the new file descriptor.

① POSIX.1-2008 addition.

Program List 5.4 Use of dup function

```
#include <stdio.h>
#include <fcntl.h>

int main (int argc, char *argv[])
{
    int    fd, newfd;
    char   buf[64] = {0};

    fd = open("./file", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        fprintf(stderr, "open file: %s failed.\n", "./file");
        return -1;
    }

    newfd = dup(fd);
    if (newfd < 0) {
        fprintf(stderr, "dup error.\n");
        close(fd);
        return -1;
    }

    write(newfd, "sylixos", 7);
    lseek(newfd, 0, SEEK_SET);
    read(fd, buf, 7);

    fprintf(stdout, "buf: %s\n", buf);
    return 0;
}
```

Running this program in SylixOS Shell, and the operation results show that lseek operation newfd is equivalent to operation fd.

```
# ./dup_test
buf: sylixos
```

9. Functions sync, fsync and fdatasync

SylixOS system has a disk cache in the kernel and most disk I/O is conducted through the buffer zone. When we write data to the file, the kernel usually copies the data into the buffer zone buffer, then queues it, and then writes it to the disk (done by the thread “t_diskcache”) when appropriate. This is called as **delayed write**.

In general, when the kernel needs to re-use the buffer zone to store other disk block data, it will write all delayed write data blocks to the disk. In order to guarantee

consistency between the actual file system on the disk and contents in the buffer zone, SylixOS provides three functions of sync, fsync and fdatasync.

```
#include <fcntl.h>
void sync(void);
int fsync(int iFd);
int fdatasync(int iFd);
```

Prototype analysis of Function fsync:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor.

Prototype analysis of Function fdatasync:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor.

The sync function arranges all modified disk caches in the system to the write queue, and then waits to complete the actual write disk operation before return.

The fsync function only works for a file specified by the file descriptor *iFd*.

The fdatasync function is similar to fsync function, but it only affects the data portion of the file. In addition to the data, fsync also synchronously updates file attributes.

10. Function fcntl

```
#include <fcntl.h>
int fcntl(int iFd, int iCmd, ...);
```

Prototype analysis of Function fcntl:

- For success of the function, return different values according to different Parameter *iCmd*. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *iCmd* is the command;
- Parameter ... is the command parameter.

Calling fcntl function can change attributes of the opened file. In the instance of this section, the 3rd parameter is always an integer. However, when the record lock is described, the 3rd parameter is a pointer to a structure.

SylixOS fcntl function supports the following 4 functions:

- Copy an existing file descriptor (*iCmd* = F_DUPFD, F_DUPFD_CLOEXEC,

`F_DUP2FD, F_DUP2FD_CLOEXEC);`

- Get / set the file descriptor flag (*iCmd* = `F_GETFD, F_SETFD`);
- Get / set the file attribute flag (*iCmd* = `F_GETFL, F_SETFL`);
- Get / set the file record lock (*iCmd* = `F_GETLK, F_SETLK, F_SETLKW`).

Table 5.4 describes the functions of the first 3 commands, and the record lock function will be described in detail in Subsection 5.4.4 File Record Lock.

Table 5.4 Function description of `fcntl`

Command	Note
<code>F_DUPFD</code>	Copy the file descriptor, equivalent to <code>dup</code> and <code>dup2</code> functions
<code>F_DUPFD_CLOEXEC</code>	Copy the file descriptor, and set the file descriptor flag
<code>F_DUP2FD</code>	Copy the file descriptor, equivalent to <code>dup2</code> function
<code>F_DUP2FD_CLOEXEC</code>	Copy the file descriptor, and set the file descriptor flag
<code>F_GETFD</code>	Get the file descriptor flag (<code>FD_CLOEXEC</code>), and return it as the return value
<code>F_SETFD</code>	Set the file descriptor flag (<code>FD_CLOEXEC</code>)
<code>F_GETFL</code>	Get the file attribute flag, and return it as the return value
<code>F_SETFL</code>	Set the file attribute flag

The following program shows how to get the file attribute flag by calling `fcntl` function. The user enters different file open attribute flags to verify that the attribute flags got by `fcntl` function are also different.

Program List 5.5 `fcntl` modify file attribute flag

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[])
{
    int fd;
    int flags, inflags = 0;

    if (argc < 3) {
        fprintf(stderr, "Don't find parse files.\n");
        return (-1);
    }

    if (!strcmp(argv[2], "O_RDONLY")) {
```

```
        inflags = O_RDONLY;
    }

    if (!strcmp(argv[2], "O_WRONLY")) {
        inflags = O_WRONLY;
    }

    if (!strcmp(argv[2], "O_RDWR")) {
        inflags = O_RDWR;
    }

    fd = open(argv[1], inflags);
    if (fd < 0) {
        fprintf(stderr, "open file: %s failed.\n", argv[1]);
        return (-1);
    }

    flags = fcntl(fd, F_GETFL, 0);
    switch (flags & O_ACCMODE) {

    case O_RDONLY:
        fprintf(stdout, "file: %s read only!\n", argv[1]);
        break;

    case O_WRONLY:
        fprintf(stdout, "file: %s write only!\n", argv[1]);
        break;

    case O_RDWR:
        fprintf(stdout, "file: %s read write.\n", argv[1]);
        break;

    default:
        fprintf(stdout, "file: %s flags: %x\n", argv[1], flags);
        break;
    }

    close(fd);
    return (0);
}
```

Run the program under SylixOS Shell. Judging from the program running results, if the open file attribute flags are different, and the file attribute flags obtained by fcntl

function will also be changed.

```
# touch test.file
# ./fcntl_test test.file O_RDONLY
file: test.file read only!
# ./fcntl_test test.file O_RDWR
file: test.file read write.
```

11. Function ioctl

```
#include <fcntl.h>
int ioctl(int iFd, int iFunction, ...);
```

- Play the role of Function ioctl
- For success of the function, return 0. For failure, return -1;
- Parameter *iFd* is the file descriptor;
- Parameter *iFunction* is the function;
- Parameter... is the function parameter.

For I/O operation, ioctl function can be seen as a “treasure chest”. Those which cannot be done with I/O function can be done with ioctl function. A lot of ioctl operations are used in the terminal I/O.

Each device driver can define its own set of ioctl commands. The system provides the generic ioctl command for different types of devices.

5.2.2 Files and directories

In the previous section, we discussed the basic operations of the file: open file, read file, write file and so on. In this section, we introduce other features of the file and how to modify these features. Finally, we will introduce the symbolic links in SylixOS.

1. Functions stat, lstat and fstat

```
#include <sys/stat.h>
int stat(const char *pcName, struct stat *pstat);
int lstat(const char *pcName, struct stat *pstat);
int fstat(int iFd, struct stat *pstat);
```

Prototype analysis of Function stat:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *pcName* is the file name;
- Output parameter *pstat* returns the file status information.

Prototype analysis of Function lstat:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **pcName** is the file name;
- Output parameter **pstat** returns the file status information.

Prototype analysis of Function fstat:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **iFd** is the file descriptor;
- Output parameter **pstat** returns the file status information.

Calling stat function will return the status information of the **pcName** file through Parameter **pstat**, calling the fstat function will get information about the file which has been opened on the descriptor **iFd**. lstat function is similar to stat function. However, when the incoming file name is the symbolic link name, lstat function will get information about the symbolic link, but not information of the actual file which the symbolic link refers to (the subsections below will detail the symbolic link).

Parameter **pstat** is a status buffer zone to be provided by the user. The pointer points to the stat structure type buffer zone, and the structure is as shown below:

```
struct stat {
    dev_t      st_dev;          /* device */
    ino_t      st_ino;         /* inode */
    mode_t     st_mode;        /* protection */
    nlink_t    st_nlink;       /* number of hard links */
    uid_t      st_uid;         /* user ID of owner */
    gid_t      st_gid;         /* group ID of owner */
    dev_t      st_rdev;        /* device type (if inode device) */
    off_t      st_size;        /* total size, in bytes */
    time_t     st_atime;       /* time of last access */
    time_t     st_mtime;       /* time of last modification */
    time_t     st_ctime;       /* time of last create */
    blksize_t  st_blksize;     /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;      /* number of blocks allocated */

    .....
};
```

In the stat structure, there are basically the basic data types of the system. The //

command is mostly used in stat function of SylixOS, and this command can get the following file information:

```
# ll
-rw-r--r-- root    root    Tue Jul 07 10:22:28 2015    0 B, test.file
-rwxr-xr-x root    root    Tue Jul 07 10:18:51 2015   233KB, app
```

Next, we will focus on the file mode (st_mode information). We introduced the file types in Section 5.1.1. Table 5.5 shows the corresponding bit information of these types in st_mode.

Table 5.5 File type bit

st_mode bit	Note
S_IFIFO	FIFO file
S_IFCHR	Character file
S_IFDIR	Directory file
S_IFBLK	Block device file
S_IFREG	Ordinary file
S_IFLNK	Symbolic link file
S_IFSOCK	Socket file

All of these file types have access permission, and each file has 9 access permission bits, just like the first column output by ll command. These access permission bits can be divided into 3 categories, as shown in Table 5.6.

Table 5.6 Access permission bit

st_mode bit	Note
S_IRUSR	User read
S_IWUSR	User write
S_IXUSR	User execution
S_IRGRP	Group read
S_IWGRP	Group write
S_IXGRP	Group execution

S_IROTH	Other read
S_IWOTH	Other write
S_IXOTH	Other implementation

In each group shown in Table 5.6, the term "user" refers to the file owner, "group" refers to the group of the owner, and "other" refers to other users not belonging to this group. The 9 permission bits can be modified with the **chmod** command. It shall be noted that during permission modification with the **chmod** command in Linux and other systems, user can be represented by u, group can be represented by g, and others can be represented by o. SylixOS is directly represented with the number. For example: 755 represents -rwxr-xr-x.

When we open a file of any type with a name, we must have execution permission for each directory contained in that name, including the current working directory it may imply. For example, in order to open the file /apps/app/test.c, it is required to have execution permission bits for directory /apps, /apps/app.

The read permission bit for a file determines whether we can open an existing file for read operation. This is related to the O_RDONLY and O_RDWR flags of open function. Of course, the write situation is similar.

2. Function access

```
#include <unistd.h>
int access(const char *pcName, int iMode);
```

Prototype analysis of Function access

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **pcName** is the file name;
- Parameter **iMode** is the access mode.

access function performs test according to the file access permission of the file owner. The test mode of access function is shown in Table 5.7.

Table 5.7 *iMode* bit of access function

iMode bit	Note
R_OK	File readable

W_OK	File writable
X_OK	File executable
F_OK	File exist

The following program shows how to use access function. The program reads the file provided by the user from the Shell interface, and then determines whether the file is writable and can be successfully opened.

Program List 5.6 Use method of access function

```

#include <stdio.h>
#include <fcntl.h>

int main (int argc, char *argv[])
{
    int    fd;

    if (argc != 2) {
        fprintf(stderr, "%s [filename].\n", argv[0]);
        return -1;
    }

    if (access(argv[1], W_OK) < 0) {
        fprintf(stdout, "%s can't write.\n", argv[1]);
    } else {
        fprintf(stdout, "%s can write.\n", argv[1]);
    }

    if ((fd = open(argv[1], O_WRONLY)) < 0) {
        fprintf(stdout, "open file failed.\n");
    } else {
        fprintf(stdout, "open file success.\n");
        close(fd);
    }

    return 0;
}

```

Run the program under the SylixOS Shell:

```

# 11
-rw-r--r-- root    root    Tue Jul 07 13:49:01 2015    0 B, b.c
-r----- root    root    Tue Jul 07 13:48:33 2015    0 B, a.c
# ./access_test b.c
b.c can write.
open file success.
# ./access_test a.c
a.c can't write.
open file failed.

```

It can be seen from the above instance that open function cannot open this file normally by setting the file access permission.

3. Function umask

```
#include <sys/stat.h>
mode_t umask(mode_t modeMask);
```

Prototype analysis of Function umask:

- This function returns the previous mask word;
- Parameter *modeMask* is the new mask word.

The umask function creates a mask word for the current process setup file, and returns the previous value. It is a function without error value returned.

Among them, Parameter *modeMask* consists of a number of bitwise "or" from the constants listed in Table 5.6.

When the process creates a new file or a new directory, it will surely use the file mode to create mask words (during introduction of open function and creat function, both functions have a mode parameter, which specifies the access permissions of the new file). For the bit as 1 the file mask word, the corresponding access permission bits in the file must be closed. It shall be noted that the owner's read permission will not be masked when the Sylix OS kernel creates a new file (this guarantees that the file owner can read files normally).

The following program shows how to use umask function. The program creates two files. When the first one is created, the umask value is 0. That is to say, no permission bit will be masked. The file will be created according to the default permission mode of the kernel. When the second is created, the umask value disables group and other read and write permissions.

Program List 5.7 Use of umask function

```
#include <stdio.h>
#include <fcntl.h>

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)

int main (int argc, char *argv[])
{
    umask(0);

    if (creat("./a.c", FILE_MODE) < 0) {
        fprintf(stderr, "create file failed.\n");
        return -1;
    }

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

```

if (creat("./b.c", FILE_MODE) < 0) {
    fprintf(stderr, "create file failed.\n");
    return -1;
}
return 0;
}

```

Running this program under the SylixOS Shell, and it can be found from the running results that the file permission bit is affected by the process mask word.

```

# ./umask_test
# 11
-rw----- root    root    Tue Jul 07 15:10:25 2015    0 B, b.c
-rw-rw-rw- root    root    Tue Jul 07 15:10:25 2015    0 B, a.c

```

4. Functions fchmod and chmod

```

#include <sys/stat.h>
int fchmod(int iFd, int iMode);
int chmod(const char *pcName, int iMode);

```

Prototype analysis of Function fchmod:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *iMode* is the mode to be set.

Prototype analysis of Function chmod:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *pcName* is the file name;
- Parameter *iMode* is the mode to be set.

Calling chmod function and fchmod function can change the access permissions of existing files. chmod function operates on the specified file, and fchmod function operates on the opened file.

The following program shows use of chmod function. The program implements two operations. One operation is to remove a certain permission based on the original access permission of the file, and the other operation is to set certain access permission bits.

Program List 5.8 Use of Function chmod

```

#include <stdio.h>
#include <sys/stat.h>

int main (int argc, char *argv[])
{
    struct stat    newstat;

    if (stat("./a.c", &newstat) < 0) {
        fprintf(stderr, "stat error.\n");
        return -1;
    }

    if (chmod("./a.c", (newstat.st_mode & ~S_IRGRP)) < 0) {
        fprintf(stderr, "drop mode error.\n");
        return -1;
    }

    if (chmod("./b.c", S_IRUSR | S_IWGRP) < 0) {
        fprintf(stderr, "set mode error.\n");
    }

    return 0;
}

```

Run the program under SylixOS Shell, and it can be seen from program running results that chmod function correctly sets corresponding access permission bits.

```

# 11
-rw-rw-rw- root    root    Tue Jul 07 15:10:25 2015    0 B, b.c
-rw-rw-rw- root    root    Tue Jul 07 15:10:25 2015    0 B, a.c
# ./chmod_test
# 11
-r---w---- root    root    Tue Jul 07 15:10:25 2015    0 B, b.c
-rw--w-rw- root    root    Tue Jul 07 15:10:25 2015    0 B, a.c

```

5. Function unlink and remove

```

#include <unistd.h>
int unlink(const char *pcName);

```

Prototype analysis of Function unlink:

- For success of the function, return 0. For failure, return -1 and set the error number;

Parameter ***pcName*** is the

name of the file to be deleted.

One can delete a file via calling the unlink function. File deletion shall satisfy a certain condition. When the reference count of the file reaches 0, the file can be deleted. When the process opens the file, it cannot be deleted. When a file is deleted, the kernel first checks the number of processes opening the file. If the number reaches 0, the kernel rechecks the reference count. If it is also 0, then the file is deleted.

If Parameter ***pcName*** is the name of a symbolic link, the symbolic link will be deleted, instead of deleting the file referenced by the symbolic link.

You can also call the remove function in ANSI C to delete a file.

```
#include <stdio.h>
int remove(const char *file);
```

Prototype analysis of Function remove:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***file*** is the name of the file to be deleted.

6. Function rename

```
#include <stdio.h>
int rename(const char *pcOldName, const char *pcNewName);
```

Prototype analysis of Function rename:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcOldName*** is the old file name;
- Parameter ***pcNewName*** is the file name modified.

The file or directory can be renamed with the rename function. The following several situations shall be indicated according to different Parameter ***pcOldName***:

- If ***pcOldName*** points to a non-directory file, there are two following situations:
 - ◆ If ***pcNewName*** already exists, ***pcNewName*** cannot be a directory file;
 - ◆ If ***pcNewName*** already exists and is not a directory file, delete it firstly, and rename pcOldName.
- If ***pcOldName*** points to a directory file, there are two following situations:
 - ◆ If ***pcNewName*** already exists, pcNewName must be an empty directory;
 - ◆ If ***pcNewName*** already exists, delete it firstly, and rename pcOldName.

- If ***pcOldName*** and ***pcNewName*** are the same file, the function returns silently without any modification.

Note: In SylixOS, if ***pcOldName*** is a symbolic link name, calling the rename function will modify the name of the real file to which the symbolic link points, and special attention shall be paid for the point.

7. Function opendir and closedir

```
#include <dirent.h>
DIR *opendir(const char *pathname);
int closedir(DIR *dir);
```

Prototype analysis of Function opendir:

- For success of the function, return the directory pointer. For failure, return NULL and set the error number;
- Parameter ***pathname*** is the directory name.

Prototype analysis of Function closedir:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***dir*** is the directory pointer.

Calling the opendir function will open the directory to which the ***pathname*** points and returns the directory pointer of DIR type, which points to the starting position of the directory. It might also be noted that the opendir function will open the directory in a read-only manner. It means that the open directory must exist. Otherwise, it will return NULL and set the errno as ENOENT. If the ***pathname*** is not a valid directory file, NULL is returned and the errno is set as ENOTDIR.

Calling the closedir function will close the directory to which ***dir*** points (***dir*** is returned via the opendir function).

8. Function readdir and readdir_r

```
#include <dirent.h>
struct dirent *readdir(DIR *dir);
int readdir_r(DIR *pdir,
              struct dirent *pdirentEntry,
              struct dirent **ppdirentResult);
```

Prototype analysis of Function readdir:

- For success of the function, return the directory information pointer. For failure,
- Parameter ***dir*** is an open directory pointer.

Prototype analysis of Function readdir_r:

For success of the function, return 0. For failure, return -1 and set the error number;

- Parameter ***pdir*** is an open directory pointer.
- Output parameter ***pdirentEntry*** returns directory information;
- Output parameter ***ppdirentResult*** points to the ***pdirentEntry*** address or NULL.

Calling the `readdir` function will return the directory information of the specified directory, and the `readdir` function is not reentrant. The `readdir_r` function is reentrant implementation of the `readdir` function, and ***pdirentEntry*** points to the user buffer for storing directory information. If the end of the directory is read, ***ppdirentResult*** is equal to NULL.

The read directory information is stored in the `dirent` structure, as shown below:

```
struct dirent {
    char          d_name[NAME_MAX + 1]; /* file name */
    unsigned char d_type;              /* file type (maybe DT_UNKNOWN) */
    char          d_shortname[13];     /* fat short file name (maybe
                                     doesn't exist) */
    .....
};
```

The `d_name` member saves the name of the file in the directory, and the `d_type` indicates the type of the file as shown in Table 5.1. The following macros can realize mutual conversion of the file type and the file type mode bit (shown in Table 5.5).

```
#include <dirent.h>
unsigned char IFTODT①(mode_t mode);
mode_t DTTOIF(unsigned char dtype);
```

IFTODT macro converts the type mode to the file type, and **DTTOIF** macro converts the file type to the type mode.

The following program shows how to use operation directory function. The program opens a appointed directory (such as "/") and reads the directory information, and then displays the name and type of the directory file.

Program List 5.9 Display directory information

```
#include <stdio.h>
#include <dirent.h>
#include <string.h>
```

① Here is a kind of representation form of macro function, which hides details realized in the macro.


```
#define DIR_PATH "/"

char *file_type (char type, char *name, int len)
{
    if (!name) {
        return (NULL);
    }

    if (S_ISDIR(DTTOIF(type))) {
        strlcpy(name, "directory", len);
        return (name);
    }

    if (S_ISREG(DTTOIF(type))) {
        strlcpy(name, "regular", len);
        return (name);
    }

    if (S_ISSOCK(DTTOIF(type))) {
        strlcpy(name, "socket", len);
        return (name);
    }

    if (S_ISLNK(DTTOIF(type))) {
        strlcpy(name, "link", len);
        return (name);
    }

    return (NULL);
}

int main (int argc, char *argv[])
{
    DIR                *dir;
    struct dirent      dirinfo;
    struct dirent      *tempdir;
    int                ret;
    char               name[64];

    dir = opendir(DIR_PATH);
    if (dir == NULL) {
        return (-1);
    }
}
```

```

while (((ret = readdir_r(dir, &dirinfo, &tempdir)) == 0) && tempdir) {
    fprintf(stdout, "file: %s type is: %s file\n",
            dirinfo.d_name, file_type(dirinfo.d_type, name, sizeof(name)));
}

closedir(dir);
return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./readdir_test
file: tmp type is: directory file
file: var type is: link file
file: root type is: link file
file: home type is: link file
.....

```

9. Function mkdir and rmdir

```

#include <sys/stat.h>
int mkdir(const char *dirname, mode_t mode);
int rmdir(const char *pathname);

```

Prototype analysis of Function mkdir:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **dirname** is the name of the created directory;
- Parameter **mode** is the creation mode.

Prototype analysis of Function rmdir:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **pathname** is the directory name.

Calling the mkdir function can create an empty directory, the access authority of the directory appointed directory is appointed by the **mode**, and the **mode** will modify the mask work according to the file mode of the progress.

Calling Function rmdir can delete an empty directory, and the underlying can be realized by calling the unlink function.

10. Function chdir, fchdir and getcwd

Each process has a current working directory, which is the starting point for searching all relative path names (the path not starting with the oblique line is called the relative path). When the user logs in SylixOS, the current working directory is usually the sixth field of the user's login entry in the password file "/etc/passwd" - the user's home directory.

```
#include <unistd.h>
int  chdir(const char *pcName);
int  fchdir(int  iFd);
char *getcwd(char *pcBuffer, size_t stByteSize);
```

Prototype analysis of Function chdir:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **pcName** is the new default directory.

Prototype analysis of Function fchdir:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **iFd** is the file descriptor.

Prototype analysis of Function getcwd:

- For success of the function, return the first address in the buffer zone of the default directory. For failure, return NULL;
- Output parameter **pcBuffer** is the default directory buffer;
- Parameter **stMaxBytes** is the size of the buffer zone.

The process calls the chdir function or the fchdir function to change the current working directory, the chdir function appoints the current working directory with parameter **pcName**, and the fchdir function appoints the current working directory via the file descriptor **iFd**.

The current working directory is a property of the process. Therefore, it is worth noting that modification of the working directory of the process will not influence the working directory of other processes.

Calling the getcwd function can get the current default working path, and the function must have a large enough buffer zone to store the name of the absolute path returned plus a terminating null byte. Otherwise, it returns error.

The getcwd function is very useful when an application is required to return to the starting point of its work in the file system. Before change of the working directory, one call the getcwd function to save the current working directory firstly. After processing, one can

take the working directory previously saved as the parameter and transfer it to the `chdir` function, which returns to the starting point of the file system.

The `fchdir` function provides a more convenient method. Before change in different locations of the file system, call the `open` function to open the current working directory firstly, and save the file descriptor returned. When it is hoped to return the original working directory, it is required to take the saved file descriptor as the parameter and transfer it to the `fchdir` function.

11. Symbolic link

The symbolic link is the indirect pointer to a file, and any user can create the symbolic link to the directory. The symbolic link is generally used to direct a file or the whole directory `str`

When the file is opened with the `open` function, if the parameter of the file name transferred to the `open` function is a symbolic link, `open` will open the appointed file with the symbolic link. However, if the file does not exist, the `open` function will return error indicating that the file does not exist, and the point shall be noticed.

```
#include <unistd.h>
int symlink(const char *pcActualPath, const char *pcSymPath);
ssize_t readlink(const char *pcSymPath, char *pcBuffer, size_t iSize);
```

Prototype analysis of Function `symlink`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcActualPath*** is the target file of the actual link;
- Parameter ***pcSymPath*** is the symbolic link file newly created.

Prototype analysis of Function `readlink`:

- For success of the function, return the length of symbolic link contents read. For failure, return -1 and set the error number;
- Parameter ***pcSymPath*** is the symbolic link name to be read;
- Output parameter ***pcBuffer*** is the content buffer zone;
- Parameter ***iSize*** is the length of the buffer zone.

SylixOS can call `symlink` function to create the symbolic links, `symlink` function will create a symbolic link ***pcSymPath*** pointing to ***pcActualPath***, and ***pcActualPath*** and ***pcSymPath*** cannot be in the same file system. The `open` function mentioned above can only open the file to which the symbolic link points. Therefore, there needs to be a way to open the symbolic link itself, and read its contents. The `readlink` provides such function.

Note: SylixOS does not support the hard link at present.

12. File truncation

Sometimes we need to intercept some data at the end of the file to shorten the file, and it is a special case to truncate the length of a file to 0. One can do this with the `O_TRUNC` flag when you open a file.

```
#include <unistd.h>
int ftruncate(int iFd, off_t oftLength);
int truncate(const char *pcName, off_t oftLength);
```

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *oftLength* is the file length.

Prototype analysis of Function truncate:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *pcName* is the file name;
- Parameter *oftLength* is the file length.

Calling the truncate function and ftruncate function can reduce or extend the file length. If the previous file length is longer than the *oftLength*, the additional data will be lost. If the previous file length is smaller than the appointed length, the file length will be expanded, i.e., the file hole is generated. The ftruncate function operates the file has been opened by the user, and the file descriptor is transferred.

5.2.3 Standard I/O library

1. Standard input, standard output and standard error

All I/O functions introduced above surround the file descriptor. When a file is opened, a file descriptor is returned, and the file descriptor is used for subsequent I/O operation. For standard I/O libraries, their operation is performed around the stream. When a file is opened with the standard I/O library, the file is correspondingly associated with the file, and then a file pointer of the FILE type is returned.

Three streams are predefined for a process, and the three streams are automatically used by the process, including standard input, standard output and standard error. The files quoted by these streams are the same with those by the file descriptors of `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

The predefined file pointers of three stream are: `stdin`, `stdout` and `stderr`, which are used by our print function previously.

2. Buffer area

The standard I/O library provides buffering to call the read function and the write function as little as possible, and automatically perform buffer management for each I/O stream, so as to avoid the trouble from consideration of the point by the application. The standard I/O library provides 3 types of buffer zones.

- Full buffer. In this case, actual I/O operations can be performed after filling the standard I/O buffer zone. Full buffer is usually performed for the file stayed on the disk via the standard I/O library.
- Line buffer. Actual I/O operations, i.e., standard input and standard output, are performed when the line break is encountered at input and output, and line buffer is used usually.
- No buffer, the standard I/O library does not perform buffer storage for the character. Standard errors are usually unbuffered, so that the error information can be indicated as far as possible.

Calling the following functions can modify types of the buffer zone.

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

Prototype analysis of Function setbuf:

- Parameter **fp** is the file pointer.
- Parameter **buf** is the buffer zone.

Prototype analysis of Function setvbuf:

- For success of the function, return 0. For failure, return non-0 value and set the error number;
- Parameter **fp** is the file pointer;
- Parameter **buf** is the buffer zone;
- Parameter **mode** is the buffer type, as shown in Table 5.8;

Table 5.8 Standard I/O buffer type

Type of buffer zone	Note
_IOFBF	Full buffer
_IOLBF	Line buffer
_IONBF	No buffer

●
buffer size.

The above function requires that the appointed stream has been opened, and called before any operation. One can use the `setbuf` function to open or close the buffer zone. In order to set a buffer zone, the parameter **buf** points to the first address of the buffer zone, and `BUFSIZ` defines the size of the buffer zone (defined in `<stdio.h>`). If one wants to close the buffer zone, it is only required to point the **buf** parameter to `NULL`. Calling the `setvbuf` function can appoint the buffer type, as shown in Table 5.8.

Generally, standard I/O will automatically release the buffer zone when the stream is closed. Of course, calling the `fflush` function can flush a stream at any time.

```
#include <stdio.h>
int fflush(FILE *fp);
```

Prototype analysis of Function `fflush`:

- For success of the function, return 0. For failure, return `EOF`^①;
- Parameter **fp** is the file pointer.

The `fflush` function transmits all unwritten data on the appointed stream to the kernel, and SylixOS does not currently support the case where **fp** is `NULL`.

3. Open the stream

```
#include <stdio.h>
FILE *fopen(const char *file, const char *mode);
FILE *freopen(const char *file, const char *mode, FILE *fp);
FILE *fdopen(int fd, const char *mode);
```

Prototype analysis of Function `fopen`:

- For success of the function, return the file pointer. For failure, return `NULL` and set the error number;
- Parameter **file** is the name of file to be opened;
- Parameter **mode** is the open mode, as shown in Table 5.9.

Prototype analysis of Function `freopen`:

- For success of the function, return the file pointer. For failure, return `NULL` and set the error number;
- Parameter **file** is the name of file to be opened;
- Parameter **mode** is the open mode, as shown in Table 5.9;
- Parameter **fp** is the file pointer.

Prototype analysis of Function fdopen:

- For success of the function, return the file pointer. For failure, return NULL and set the error number;
- Parameter **fd** is the open file descriptor;
- Parameter **mode** is the open mode.

Above 3 functions can open a standard I/O stream, and the fopen function opens the file appointed by **file**. The freopen function opens an appointed file on an appointed stream. If the stream has been opened, it is closed first. If the stream has been orientated, the orientation will be cleared. The fdopen function takes an existing file descriptor, and combines a standard I/O stream with the descriptor.

Table 5.9 mode parameter of opening standard I/O stream

Operation type	Note	Symbol of the open function
r or rb	Open it in a read-only manner	O_RDONLY
w or wb	Truncate the file to 0 byte length, or create it in a write manner	O_WRONLY O_CREAT O_TRUNC
a or ab	Add, and open it in a write manner at the end of the file, or create it in a write manner	O_WRONLY O_CREAT O_APPEND
r+ or r+b or rb+	Open it in read and write manners	O_RDWR
w+ or w+b or wb+	Truncate the file to 0 byte length, or open it in read and write manners	O_RDWR O_CREAT O_TRUNC
a+ or a+b or ab+	Open or create it from the file end in read and write manners	O_RDWR O_CREAT O_APPEND

Use character b as a part of the **mode**. Therefore, the standard I/O system can differentiate the text file and the binary file. However, these two files are not differentiated in SylixOS, so that Character b is invalid in SylixOS.

For the fdopen function, the meaning of the **mode** parameter is slightly different. The file descriptor has been opened. Therefore, the file opened via the fdopen function for writing are not truncated. In addition, the append write manner of standard I/O cannot be used to create the file.

After opening a file with append write, the data will be written at the end of the file each time. If multiple processes open the same file via the append write manner of standard I/O, the data from each process will be correctly written in the file.

Calling the fclose function close an open stream.

```
#include <stdio.h>
```



```
int fclose(FILE *fp);
```

Prototype analysis of Function fclose:

- For success of the function, return 0. For failure, return EOF and set the error number;
- Parameter **fp** is the file pointer.

Before the file is closed, the output data in the buffer zone is flushed first, and the input data in the buffer is discarded. If the standard I/O library has automatically distributed the buffer zone, it is released.

When a process is terminated normally, all standard I/O streams with unwritten buffer data are flushed, and all open standard I/O streams are closed.

4. Read and write streams

Once the stream is opened, one can read and write different types of I/O, and read a character at a time when calling the following functions.

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

Prototype analysis of Function getc:

- For success of the function, return the next character. For failure, return EOF;
- Parameter **fp** is the file pointer.

Prototype analysis of Function fgetc:

- For success of the function, return the next character. For failure, return EOF;
- Parameter **fp** is the file pointer.

Prototype analysis of Function getchar:

- For success of the function, return the next character. For failure, return EOF.

The getchar function is equal to getc (stdin). The difference between the getc function and the getchar function is that the getc function can be implemented as the macro, and the fgetc function cannot be implemented as the macro.

Above 3 input functions correspond to the following 3 output functions.

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

Prototype analysis of Function `putc`:

- For success of the function, return the entered character. For failure, return EOF and set the error number;
- Parameter ***c*** is the character to be entered;
- Parameter ***fp*** is the file pointer.

Prototype analysis of Function `fputc`:

- For success of the function, return the entered character. For failure, return EOF and set the error number;
- Parameter ***c*** is the character to be entered;
- Parameter ***fp*** is the file pointer.

Prototype analysis of Function `putchar`:

- For success of the function, return the entered character. For failure, return EOF and set the error number;
- Parameter ***c*** is the character to be entered.

`Putchar(c)` is equal to `putc(c, stdout)`, and the `putc` function and the `fputc` function can output a character to the appointed stream. The difference is that the `putc` function can be implemented as the macro, and the `fputc` function cannot be implemented as the macro.

The following functions can read a line of characters from the appointed stream (line terminators are represented with `"\n"`).

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
```

Prototype analysis of Function `fgets`:

- For success of the function, return the first address of `buf`. For failure, return `NULL`;
- Parameter ***buf*** is the character buffer zone;
- Parameter ***n*** is the length of the buffer zone;
- Parameter ***fp*** is the file pointer.

Prototype analysis of Function `gets`:

- For success of the function, return the first address of ***buf***. For failure, return `NULL`;
- Parameter ***buf*** is the character buffer zone,

Two functions appoint the address of the buffer zone to which the lines read will be sent. The gets function is read from standard input, and the fgets function is read from the appointed stream.

The gets function must appoint the length of the buffer zone, and the function is always read until the next break line character, not exceeding n-1 characters. The read character is sent to the buffer zone ending with a null character. If the number of characters in the final line include the line is more than n-1, the fgets function returns only an incomplete line. The buffer always ends with the null byte, and the next call to the fgets function will continue read the line.

The gets function is not recommended, because the caller does not appoint the length of the buffer, which may cause buffer area overflow^①.

The following function can output one line of characters to the appointed stream.

```
#include <stdio.h>
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

Prototype analysis of Function fputs:

- For success of the function, return the non-negative value. For failure, return EOF and set the error number;
- Parameter **str** is character string to be written;
- Parameter **fp** is the file pointer.

Prototype analysis of Function puts:

- For success of the function, return the non-negative value. For failure, return EOF and set the error number;
- Parameter **str** is character string to be written.

The fputs function writes a null-terminated string to the appointed stream, and the stop character null at the end is not written. The puts function writes a null-terminated string to the standard output, and the stop character null is not written. Unlike the fputs function, the puts function will write a break line character to the standard output afterwards. The puts function is not recommended generally.

The following program shows how to use the standard I/O read and write functions. The program writes a string to the open file, then call the rewind function (usage of the function will be introduced later) to move the file's current pointer to beginning of the file, and call the fgets function to read the character string in the file for printing.

Program List 5.10 Use of standard I/O function read and write

```
#include <stdio.h>
```

```

int main (int argc, char *argv[])
{
    FILE    *fp;
    int     ret;
    char    *str = "This is test sylixos string functions example.";
    char    buf[64] = {0};

    fp = fopen("file", "r+");
    if (fp == NULL) {
        fprintf(stderr, "fopen error.\n");
        return (-1);
    }

    ret = fputs(str, fp);
    if (ret < 0) {
        fprintf(stderr, "fputs write error.\n");
        fclose(fp);
        return (-1);
    }

    rewind(fp);                                /* currently offset to the
beginning of the file                        */

    if (fgets(buf, sizeof(buf), fp) == NULL) {
        fprintf(stderr, "fgets read fp error.\n");
        fclose(fp);
        return (-1);
    }

    fprintf(stdout, "buf: %s\n", buf);
    fclose(fp);
    return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./iorw_test
buf: This is test sylixos string functions example.

```

5. Positioning stream

```

#include <stdio.h>
long  ftell(FILE *fp);
int   fseek(FILE *fp, long offset, int whence);

```

```
void rewind(FILE *fp);
```

Prototype analysis of Function ftell:

- For success of the function, return the current file offset. For failure, return -1 and set the error number;
- Parameter **fp** is the file pointer.

Prototype analysis of Function fseek:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **fp** is the file pointer;
- Parameter **offset** is the set offset;
- Parameter **whence**, as shown in Table 5.3.

Prototype analysis of Function rewind:

- Parameter **fp** is the file pointer.

For a binary file, its file location indicator is measured in byte from the file start location. When ftell function is used for the binary file, the return value is this byte location. In order to use fseek to locate a binary file, one must appoint a byte **offset** and **whence**. ISO C does not require an implementation to support SEEK_END for binary files. The reason is that some systems require that the length of the binary file is the integral multiple of a certain magic number. However, SEEK_END is supported in SylixOS.

For the text file, the current location of their files may not be measured by the simple byte offset. This is also mainly in non-UNIX systems, and the text files may be stored in different formats. In order to locate a text file, **whence** must be SEEK_SET, and **offset** can only have two values: 0 (rewind to the file start location), or the value returned by ftell function of the file. A stream can also be set to the file start location by using rewind function.

The fgetpos function and the fsetpos function are introduced by ISO C standard.

```
#include <stdio.h>
int fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

Prototype analysis of Function fgetpos:

- For success of the function, return 0. For failure, return non-0 value and set the error number;
- Parameter **fp** is the file pointer.
- The output parameter **pos** is the file offset location.

Prototype analysis of Function fsetpos:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **fp** is the file pointer.
- Parameter pos is the file offset.

The fgetpos function stores the current value of the file position indicator in the object pointed to by pos. You can use this value to relocate the stream to that position when the fsetpos function is called later.

6. I/O formatting

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *fp, const char *format, ...);
int fdprintf(int fd, const char *format, ...);
int sprintf(char *buf, const char *format, ...);
int snprintf(char *buf, size_t n, const char *format, ...);
```

Prototype analysis of Function printf:

- For success of the function, return the number of characters output. For failure, return the negative value;
- Parameter **format** is the format character string;
- Parameter... is a variable parameter.

Prototype analysis of Function fprintf:

- For success of the function, return the number of characters output. For failure, return the negative value;
- Parameter **fp** is the file pointer;
- Parameter **format** is the format character string;
- Parameter... is a variable parameter.

Prototype analysis of Function fdprintf:

- For success of the function, return the number of characters output. For failure, return the negative value;
- Parameter **fd** is the file descriptor;
- Parameter **format** is the format character string;
- Parameter... is a variable parameter.

Prototype analysis of Function sprintf:

- For success of the function, return the number of characters output. For failure, return the negative value;
- Parameter **buf** is the pointer of the character buffer zone;
- Parameter **format** is the format character string;
- Parameter... is a variable parameter.

Prototype analysis of Function snprintf:

- For success of the function, return the number of characters output. For failure, return the negative value;
- Parameter **buf** is the pointer of the character buffer zone;
- Parameter **n** is the length of the buffer zone;
- Parameter **format** is the format character string;
- Parameter... is a variable parameter.

The printf function prints characters to the standard output stream in the format specified by **format**. The fprintf function prints characters to the stream specified by **fp** in the format specified by **format**. Parameter **fd** of the fdprintf function is the descriptor of the opened file. This function prints the format character to the file specified by **fd**. The sprintf function and The snprintf function print the format characters to the buffer zone specified by **buf**. The difference between two functions is that snprintf function specifies the length of the buffer zone, so as to guarantee memory security. It is not suggested to use sprintf function.

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
int sscanf(const char *buf, const char *format, ...);
```

Prototype analysis of Function scanf:

- For success of the function, return the number of matching characters. Otherwise, return EOF;
- Parameter **format** is the format character string;
- Parameter... is a variable parameter.

Prototype analysis of Function fscanf:

- For success of the function, return the number of matching characters. Otherwise, return EOF;

Parameter ***fp*** is the file

- pointer;

- Parameter ***format*** is the format character string;
- Parameter... is a variable parameter.

Prototype analysis of Function `sscanf`:

- For success of the function, return the number of matching characters. Otherwise, return EOF;
- Parameter ***buf*** is the pointer of the buffer zone;
- Parameter ***format*** is the format character string;
- Parameter... is a variable parameter.

The `scanf` function scans the standard input and stores the value in the ***format*** format to the appropriate memory. The memory address will be given in the variable parameter. Functions of the `fscanf` function are similar to those of the `sscanf` function. The difference is that the `fscanf` function scans characters from the stream specified by `fp`, while the `sscanf` function scans characters from the buffer zone specified by `buf`.

The following program shows how to use the formatting function. The program uses the `fdopen` function to get the file pointer from the opened file descriptor.

Program List 5.11 Use of the formatting function

```
#include <stdio.h>
#include <fcntl.h>

int main (int argc, char *argv[])
{
    FILE    *fp;
    char    buf[64] = {0};
    char    *content = "Format print function test.";
    char    *temp;
    int     fd;

    fd = open("file", O_RDWR | O_CREAT, 0644);
    if (fd < 0) {
        fprintf(stderr, "open failed test done.\n");
        return (-1);
    }

    fp = fdopen(fd, "r+");
    if (fp == NULL) {
```



```
        fprintf(stderr, "fdopen failed test done.\n");
    close(fd);
    return (-1);
}

fprintf(fp, "%s", content);
rewind(fp);

temp = fgets(buf, sizeof(buf), fp);
if (temp == NULL) {
    fprintf(stderr, "fgets read error or file end.\n");
    fclose(fp);
    return (-1);
}

fprintf(stdout, "buf: %s\n", buf);
fclose(fp);
return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./format_test
buf: Format print function test.
```

5.3 Asynchronous I/O access

The signaling mechanism (see Chapter 10 Signaling System) provides a way to asynchronously notify that certain events have occurred. However, such asynchronous I/O is limited. They cannot be used on all file types, and only a signal can be used. If asynchronous I/O is performed on more than one file descriptors, then the process does not know which file descriptor the signal corresponds to when receiving the signal.

SylixOS supports the standard asynchronous I/O interfaces defined by the POSIX 1003.1b real-time extension protocol, namely `aio_read` function, `aio_write` function, `aio_fsync` function, `aio_cancel` function, `aio_error` function, `aio_return` function, `aio_suspend` function and `lio_listio` function. This group of APIs is used to operate asynchronous I/O. Asynchronous I/O is a concept proposed as per synchronous I/O. It does not require the thread to wait for I/O results, but only requests transmission. Then the system automatically completes the I/O transmission. At the end or when an error occurs, the corresponding I/O signal will be generated. The user program only needs to set the corresponding signal sink function to handle an asynchronous I/O event.

5.3.1 POSIX asynchronous I/O

The POSIX asynchronous I/O interface provides a set of consistent method for asynchronous I/O for different types of files. These interfaces come from the real-time draft standard, and these asynchronous I/O interfaces adopt AIO control block to describe I/O operation. The `aiocb` structure defines AIO control block, and SylixOS implements this structure as follows:

```
struct aiocb {
    int          aio_fildes;    /* file descriptor          */
    off_t        aio_offset;    /* file offset              */
    volatile void *aio_buf;     /* Location of buffer.      */
    size_t       aio_nbytes;    /* Length of transfer.      */
    int          aio_reqprio;   /* Request priority offset. */
    struct sigevent aio_sigevent; /* Signal number and value. */
    int          aio_lio_opcode; /* Operation to be performed. */
    .....
};
```

The following describes the structure members in detail:

- The `aio_fildes` is the file descriptor to be operated, i.e., the file descriptor returned by the `open` function;
- The `aio_offset` is the file offset at the start of read and write;
- The `aio_buf` is a pointer of the data buffer zone. For read, the data is read from the buffer zone. For write, the data is written to the buffer zone;
- The `aio_nbytes` is the number of bytes read and written;
- The `aio_reqprio` is the priority of asynchronous I/O request. This priority determines the read and write sequence, which also means that the higher the priority is, the sooner it is read or written.
- The `aio_sigevent` is the signal to be sent. When a read or write is completed, the the signal specified by the application will be sent.
- The `aio_lio_opcode` is the type of asynchronous I/O operation, as shown in Table 5.10;

Table 5.10 Type of asynchronous I/O operation

Operation type	Note
LIO_NOP	No transmission request
LIO_READ	Request a read operation

LIO_WRITE	Request a write operation
LIO_SYNC	Request asynchronous I/O synchronization

Before asynchronous I/O operation, we need to initialize the AIO control block firstly, and then request asynchronous write operation by calling the `aioread` function, and request an asynchronous read operation by calling the `aiowrite` function.

```
#include <aio.h>
int aioread(struct aiocb *paiocb);
int aiowrite(struct aiocb *paiocb);
```

Prototype analysis of Function `aioread`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***paiocb*** is AIO control block.

Prototype analysis of Function `aiowrite`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***paiocb*** is AIO control block.

After the `aioread` function and the `aiowrite` function have been called successfully, the asynchronous I/O request has already been put into the waiting queue by the operating system. These return values have nothing to do with the results of actual I/O operation. If it is required to view the return status of the function, you can call `aierror` function.

To force all waiting asynchronous operations to write to memory without waiting, one can create an AIO control block, and call the `aiofsyc` function.

```
#include <aio.h>
int aiosync(int op, struct aiocb *paiocb);
```

Prototype analysis of Function `aiofsyc`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***op*** is the operation option;
- Parameter ***paiocb*** is AIO control block.

The `aiocb` member in the structure `struct aiocb` is the file to be synchronized. If the ***op*** parameter is set to `O_SYNC`, the `aiofsyc` call is similar to `fsync`. If the ***op*** parameter is set as `O_DSYNC`, the `aiofsyc` call is similar to `fdatasync` (Currently

SylixOS does not make a specific distinction between the two situations).

Same with the `aioread` function and the `aiowrite` function, `aioread` can be called by viewing the `aioread` function.

```
#include <aioread.h>
int aioread(const struct aiocb *paiocb);
```

Prototype analysis of Function `aioread`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter `paiocb` is AIO control block.

The return value shall be one of the following 4 situations:

- 0: asynchronous operation completed successfully, call `aioread` function to get the return code;
- -1: failure to call `aioread`;
- EINPROGRESS: waiting for asynchronous operation;
- Other cases: any other return value is the error code returned by the associated asynchronous operation failure.

If asynchronous operation is successful, one can call the `aioread` function to get the return value of asynchronous operation.

```
#include <aioread.h>
ssize_t aioread(struct aiocb *paiocb);
```

Prototype analysis of Function `aioread`:

- For success of the function, return the return value after asynchronous operation completion. For failure, return -1 and set the error number;
- Parameter `paiocb` is AIO control block.

When performing I/O operations, one can use asynchronous I/O if there are other transactions to be processed without being blocked by I/O operation. However, if all transactions are completed but asynchronous operation is not completed, one can call the `aioread` function to block the process until operation is completed.

```
#include <aioread.h>
int aioread(const struct aiocb * const list[], int nent,
            const struct timespec *timeout);
```

Prototype analysis of Function `aioread`:

For success of the function, return 0. For failure, return -1 and set the error number;

- Parameter **list** is the array of AIO control blocks;
- Parameter **nent** is the number of array elements;
- Parameter **timeout** is the set timeout period.

The `aiosuspend` may return one of the following 3 situations:

- In case of block timeout, then the `aiosuspend` function will return -1;
- If any I/O operation is completed, the `aiosuspend` function will return 0;
- If all asynchronous I/O operations have been completed when the `aiosuspend` function is called, the `aiosuspend` function will return when not blocked.

Parameter **list** will automatically ignore the null pointer, and the non-null element is the initialized AIO control block.

When there are still waiting asynchronous I/O operations not to be completed, one can call the `aio_cancel` function to cancel them.

```
#include <aio.h>
int aio_cancel(int filides, struct aiocb *paiocb);
```

Prototype analysis of Function `aio_cancel`:

- This function return value is shown in Table 5.11;

Table 5.11 Type of `aio_cancel` return value

Type of return value	Note
AIO_CANCELED	All requested operations have been canceled
AIO_NOTCANCELED	At least one requested operation is not canceled
AIO_ALLDONE	Completed before the request is canceled
-1	Failure to call the <code>aio_cancel</code> function

- Parameter **filides** is the file descriptor to be operated;
- Parameter **paiocb** is AIO control block.

If **paiocb** is NULL, the system will try to cancel all unfinished asynchronous I/O operations on this file. In other cases, the system will attempt to cancel the single asynchronous I/O operation described by the AIO control block specified by **paiocb**.

If the asynchronous I/O operation is successfully canceled, the corresponding AIO

control block will call the `aio_error` function to return error ECANCE LED. If the operation cannot be canceled, the corresponding AIO control block will not be modified due to call of the `aio_cancel` function.

The following program shows how to use the above function. The program constructs an AIO control block for read operation, and then calls `aio_read` to request read operation. After the operation is completed, the `signal_handler` function will be notified via the signal (see the signal section) for further processing.

Program List 5.12 Use of asynchronous I/O function[®]

```
#include <aio.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#define BUFSIZE (64)

void signal_handler (union sigval val)
{
    struct aiocb    *paio = (struct aiocb *)val.sival_ptr;
    ssize_t        ret;

    ret = aio_return(paio);
    if (ret < 0) {
        fprintf(stderr, "aio_return error.\n");
        return;
    }

    fprintf(stdout, "len: %ld\ncontent: %s\n", ret, (char *) (paio->aio_buf));
}

int main (int argc, char *argv[])
{
    int            fd;
    int            ret;
    static struct aiocb    aio;
    const struct aiocb *const list[] = {&aio};

    fd = open("file", O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "open file failed.\n");
    }
}
```

```

        return (-1);
    }

    memset(&aio, 0, sizeof(struct aiocb));
    /*
     * 设置 AIO 控制块
     */
    aio.aio_fildes      = fd;
    aio.aio_buf         = malloc(BUFSIZE + 1);
    aio.aio_nbytes      = BUFSIZE;
    aio.aio_offset      = 0;
    aio.aio_reqprio     = 1;
    aio.aio_lio_opcode  = LIO_READ;
    aio.aio_sigevent.sigev_notify      = SIGEV_THREAD | SIGEV_SIGNAL;
    aio.aio_sigevent.sigev_value.sival_ptr = (void *)&aio;
    aio.aio_sigevent.sigev_signo       = SIGUSR1;
    aio.aio_sigevent.sigev_notify_function = signal_handler;

    ret = aio_read(&aio);
    if (ret < 0) {
        perror("aio_read");
        close(fd);
        return (-1);
    }

    aio_suspend(list, 1, NULL);
    close(fd);
    return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./aio_read_test
len: 64
content: This is a sylixos test file

```

Calling the `lio_listio` function can submit a series of I/O requests described by a list of AIO control blocks.

```

#include <aio.h>
int lio_listio(int mode, struct aiocb * const list[], int nent,
               struct sigevent *sig);

```

Prototype analysis of Function `lio_listio`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **mode** is the transfer mode (LIO_WAIT, LIO_NOWAIT);
- Parameter **list** is the array of request AIO control blocks;
- Parameter **nent** is the array of AIO control blocks;
- Parameter **sig** is the signal method generated after all I/O operations are completed.

Parameter **mode** determines whether I/O is really asynchronous. If this parameter is set as LIO_WAIT, the `lio_listio` function will return after all I/O operations specified by the list are completed. In this case, Parameter **sig** will be ignored. If the **mode** parameter is set as LIO_NOWAIT, the `lio_listio` function will return immediately after the I/O request is enqueued. The process will be notified asynchronously as specified by Parameter **sig** after all I/O operations are completed. If notice is not required, one can set Parameter **sig** as NULL. It might also be noted that each asynchronous I/O operation can be set with its own notification mode. However, the notification mode specified by Parameter **sig** is an additional notification mode, and it will be notified only after all operations are completed.

In each AIO control block, the `aio_lio_opcode` field specifies whether the operation is a read operation (LIO_READ), a write operation (LIO_WRITE), or a null operation (LIO_NOP) which will be ignored.

The following program shows how to use the `lio_listio` function. It can be seen from the code that a single `lio_listio` function call initiates multiple transmissions. From this point, it can be concluded that performance of the `lio_listio` function is worth studying.

Program List 5.13 Use of `lio_listio` function

```
#include <aio.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#define BUFSIZE (64)

void signal_handler (union sigval val)
{
    fprintf(stdout, "async operator complete.\n");
}

int main (int argc, char *argv[])
```



```
{
int          fd;
int          ret;
struct aiocb aio[2];
struct aiocb *const list[] = {&aio[0], &aio[1]};
static struct sigevent notify;

fd = open("file", O_RDONLY);
if (fd < 0) {
    fprintf(stderr, "open file failed.\n");
    return (-1);
}

memset(&aio[0], 0, sizeof(struct aiocb));
memset(&aio[1], 0, sizeof(struct aiocb));
/*
 * Set up the first AIO control block
 */
aio[0].aio_fildes = fd;
aio[0].aio_buf    = malloc(BUFSIZE + 1);
aio[0].aio_nbytes = BUFSIZE;
aio[0].aio_offset = 0;
aio[0].aio_reqprio = 1;
aio[0].aio_lio_opcode = LIO_READ;
/*
 * Set up second AIO control block
 */
aio[1].aio_fildes = fd;
aio[1].aio_buf    = malloc(BUFSIZE + 1);
aio[1].aio_nbytes = BUFSIZE;
aio[1].aio_offset = BUFSIZE;
aio[1].aio_reqprio = 2;
aio[1].aio_lio_opcode = LIO_READ;

notify.sigev_signo      = SIGUSR1;
notify.sigev_notify_function = signal_handler;
notify.sigev_notify     = SIGEV_THREAD | SIGEV_SIGNAL;

ret = lio_listio(LIO_NOWAIT, list, 2, &notify);
if (ret < 0) {
    perror("lio_listio");
    close(fd);
    return (-1);
}
```

```

    }

    sleep(60);
    close(fd);
    return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./lio_listio_test
signal handler.

```

During implementation of asynchronous I/O of SylixOS, I/O operation will be performed via an additional thread (proxy thread). In order to be able to set or get the stack information of the proxy thread, SylixOS adds the following set of functions. This set of functions are not defined in the standard. It might also be noted that that the set stack information is only valid for future start threads (this usually occurs in future I/O requests).

```

#include <aio.h>
int aio_setstacksize(size_t newsize);
size_t aio_getstacksize(void);

```

Prototype analysis of Function `aio_setstacksize`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *newsize* is the new stack size.

Prototype analysis of Function `aio_getstacksize`:

- This function always returns the size of the current thread stack.

Call the `aio_setstacksize` function to set the stack size for future start threads (proxy thread), and call the `aio_getstacksize` function to get the size of the current thread (proxy thread) stack.

5.4 Advanced I/O access

5.4.1 Decentralized aggregation operation

```

#include <sys/uio.h>
ssize_t readv(int iFd, struct iovec *piovec, int iIovcnt);
ssize_t writev(int iFd, const struct iovec *piovec,int iIovcnt);

```

Prototype analysis of Function `readv`:

- For success of the function, return the number of bytes read. For failure, return -1 and set the error number;
- Parameter *ifd* is the file descriptor;
- The output parameter *popovec* is a pointer of the decentralized buffer zone array;
- Parameter *ilovcnt* is the number of the buffer zones.

Prototype analysis of Function *writv*:

- For success of the function, return the number of bytes written. For failure, return -1 and set the error number;
- Parameter *ifd* is the file descriptor;
- Parameter *popovec* is an pointer of the aggregation buffer zone (data to be sent) array;
- Parameter *ilovcnt* is the number of the buffer zones.

The *readv* function and the *writv* function are used to read and write multiple non-contiguous buffer zones in a single function call. Sometimes, these two functions are called scatter read and gather write.

The second parameter of these two functions is a pointer to the *iovec* structure array:

```
struct iovec {
    PVOID    iov_base;           /* base address          */
    size_t   iov_len;           /* length                */
};
```

The member *iov_base* in the structure points to the first address of a buffer zone, and the member *iov_len* is the length of the buffer zone. The relation between the parameters of these two functions and the *iovec* structure is shown in Figure 5.7.

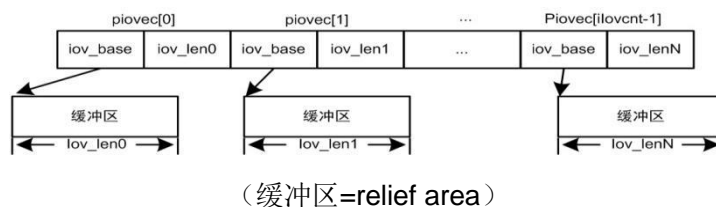


Figure 5.7 Relation between *readv* and *writv* and *iovec* structures

The order to read *readv* function from the buffer zone is *piovec*[0], *piovec*[1] until *piovec* [*ilovcnt*-1], and the total number of bytes read is successfully returned. The write order of the *writv* function is the same with that of *readv*, and the total number of bytes

written is successfully returned.

The following program shows how to use the `readv` function and the `writev` function. The program defines two `iovec` elements, two buffer zones with different lengths, calls the `readv` function to read data from the file `a.test`, and then calls the `writev` function to write data to the file `b.test`.

Program List 5.14 Use of `readv` and `writev`

```
#include <stdio.h>
#include <sys/uio.h>

int main (int argc, char *argv[])
{
    int          fd;
    char         buf1[15], buf2[34];
    struct iovec iov[2];
    ssize_t      ret;

    iov[0].iov_base = buf1;
    iov[0].iov_len  = sizeof(buf1);
    iov[1].iov_base = buf2;
    iov[1].iov_len  = sizeof(buf2);

    fd = open("a.test", O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "open file: %s failed.", "a.test");
        return (-1);
    }

    ret = readv(fd, iov, 2);
    if (ret < 0) {
        fprintf(stderr, "readv error.\n");
        close(fd);
        return (-1);
    }
    fprintf(stdout, "readv read bytes %ld\n", ret);
    close(fd);

    fd = open("b.test", O_WRONLY);
    if (fd < 0) {
        fprintf(stderr, "open file: %s failed.\n", "b.test");
        return (-1);
    }
}
```

```

ret = writev(fd, iov, 2);
if (ret < 0) {
    fprintf(stderr, "writev error.\n");
    close(fd);
    return (-1);
}
fprintf(stdout, "writev write bytes %ld\n", ret);
close(fd);
return (0);
}

```

Run the program under the SylixOS Shell. It can be seen from the program operation results that the readv function will read two buffer zones in turn, while the writev function will write the data in the two buffer zones to the specified file.

```

# cat a.test
This is a sylixos test readv and writev example.
# ./rwy_test
readv read bytes 49
writev write bytes 49
# cat b.test
This is a sylixos test readv and writev example.

```

5.4.2 Non-blocking I/O

Some "low-speed" system functions^① may block the process forever, such as certain inter-process communication functions, certain ioctl operations and so on.

Non-blocking I/O makes open, read and write operations unblocked forever (requiring support by the device driver). If these operations cannot be completed, the call immediately returns with an error, indicating that the operation will be blocked if it continues.

There are two ways to get a non-blocking I/O:

- Call the open function to get a file descriptor, and one can specify the O_NONBLOCK flag;
- If the file is already open, one can call ioctl to specify the FIONBIO command, or call fcntl to specify the F_SETFL option.

5.4.3 I/O multiplexing

The I/O multiplexing technique is to construct a list of file descriptors which we are

interested in, and then call a function. The function will not return until one of these descriptors is ready for I/O. 4 functions of select, pselect, poll and ppoll can implement I/O multiplexing functions. When these functions return, the process or thread is notified which file descriptors have been ready for I/O operation.

The details of the select function, pselect function, poll function, and ppoll function are described as below.

1. select function group

```
#include <sys/select.h>
int select(int          iWidth,
           fd_set      *pfdsetRead,
           fd_set      *pfdsetWrite,
           fd_set      *pfdsetExcept,
           struct timeval *ptmvalTO);
int pselect(int          iWidth,
            fd_set      *pfdsetRead,
            fd_set      *pfdsetWrite,
            fd_set      *pfdsetExcept,
            const struct timespec *ptmspecTO,
            const sigset_t  *sigsetMask);
```

Prototype analysis of Function select:

- For success of the function, return the quantity of descriptor waited. For failure, return -1 and set the error number;
- Parameter ***iWidth*** is the maximum descriptor in the list of file descriptors plus 1;
- Parameter ***pfdsetRead*** is the set of read descriptors;
- Parameter ***pfdsetWrite*** is the set of write descriptors;
- Parameter ***pfdsetExcept*** is the set of exception descriptors;
- Parameter ***ptmvalTo*** is the waiting timeout.

Prototype analysis of Function pselect:

- For success of the function, return the quantity of descriptor waited. For failure, return -1 and set the error number;
- Parameter ***iWidth*** is the maximum descriptor in the list of file descriptors plus 1;
- Parameter ***pfdsetRead*** is the set of read descriptors;
- Parameter ***pfdsetWrite*** is the set of write descriptors;
- Parameter ***pfdsetExcept*** is the set of exception descriptors;
- Parameter ***ptmspecTo*** is the waiting timeout;

the signal blocked at waiting.

Judging from parameters of the select function and the parameters transferred to the kernel, the following points can be seen:

- Tell the kernel which file descriptors we care about;
- Conditions which we care about for each file descriptor (read, write and exception);
- How long you are willing to wait (you can wait forever, you may not wait, you can wait for the specified time);

When select returns, we can know how many file descriptors are ready and which file descriptors are ready. Read, write and other operations can be performed with these ready file descriptors.

- Parameter **ptmvalTo** can be divided into 3 kinds of situations:
- When **ptmvalTo** == NULL, it represents waiting forever,
- When **ptmvalTo->tv_sec** == 0 && **ptmvalTo->tv_usec** == 0, it represents noy wait,
- When **ptmvalTo->tv_sec** != 0 || **ptmvalTo->tv_usec** != 0, it represents waiting for the satisfy number of seconds and microseconds.

Parameters **pdfsetRead**, **pdfsetWrite** and **pdfsetExcept** are pointers to the file descriptor set. Each file descriptor set is stored in a variable of the fd_set data type. For this type, different systems may have different implementations. Here we can consider it as a very large byte array, and each file descriptor occupies a bit. In SylixOS, the following group of macros is provided for variables of fd_set type:

Table 5.12 Variable operation macro of fd_set type

Macro name	Note
FD_SET(n, p)	Set the file descriptor n in the file descriptor set p
FD_CLR(n, p)	Clear the file descriptor n from file descriptor set p
FD_ISSET(n, p)	Determine whether the file descriptor n belongs to the file descriptor set p
FD_ZERO(p)	Clear the file descriptor set p

After declaring a file descriptor set of fd_set type, firstly use FD_ZERO to clear the file descriptor set, and then use FD_SET to put the file descriptor we care about into the set. When select returns successfully, use FD_ISSET to determine it is the file descriptor we

care about.

The select function has 3 possible return values:

- The return value -1 indicates an error. For example, if a signal is captured when none of the specified file descriptors is ready, return -1;
- Return value 0 represents no file descriptor is ready, because no file descriptor is ready within the specified time, i.e., call timeout;
- The return value is an integer larger than zero. This value is the sum of all ready file descriptors in the 3 file descriptor sets.

The select function returns the sum of the ready file descriptors, and here "ready" has the following meanings:

- For a file descriptor in the read set, read operation will not block;
- For a file descriptor in the write set, write operation will not block;
- For a file descriptor in the exception set, there is a pending exception condition.

It might also be noted that if the file end is encountered on a file descriptor, the select function will consider the file descriptor as readable, and then call the read function to return 0.

In the above definition, we see that except for the last two parameters and select function, other parameters of the pselect function are the same. Let's introduce these two different parameters.

The type of select function timeout value is struct timeval, while timeout value type of the pselect function is struct timespec (see Chapter 11 Time Management). The timespec structure expresses the timeout value in seconds and nanoseconds, that is to say, the pselect function provides more accurate timeout value than the select function^①.

The pselect function can use signal mask words. If sigmask is NULL, the operation conditions of the pselect function and the select function are the same. Otherwise, sigmask points to a signal mask word, which is installed through atomic operation when the pselect function is called. On return, the previous signal mask word is recovered.

The following program shows how to the select function. The program waits for the standard input (STDIN_FILENO) descriptor to be readable. If select returns within the timeout period, the standard input is read, and the read character is printed.

Program List 5.15 Use of select function

```
#include <stdio.h>
#include <sys/select.h>

int main (int argc, char *argv[])
```



```
{
fd_set      fdset;
int         ret;
struct timeval timeout;
char        ch;

timeout.tv_sec = 10;
timeout.tv_usec = 0;

for (;;) {
    FD_ZERO(&fdset);
    FD_SET(STDIN_FILENO, &fdset);

    ret = select(STDIN_FILENO + 1, &fdset, NULL, NULL, &timeout);
    if (ret <= 0) {
        break;
    } else if (FD_ISSET(STDIN_FILENO, &fdset)){
        read(STDIN_FILENO, &ch, 1);
        if (ch == '\n') {
            continue;
        }
        fprintf(stdout, "input char: %c\n", ch);

        if (ch == 'q') {
            break;
        }
    }
}

return (0);
}
```

Run the program under the SylixOS Shell, and enter the characters to see the corresponding print.

```
# ./select_test
h
input char: h
```

2. poll function group

Functions of the poll function are similar to those of the select function, but the function interface are different.

```
#include <poll.h>
```

```

int poll(struct pollfd fds[], nfd_t nfds, int timeout);
int ppoll(struct pollfd          fds[],
          nfd_t                  nfds,
          const struct timespec  *timeout_ts,
          const sigset_t         *sigmask);

```

Prototype analysis of Function poll:

- For success of the function, return the quantity of descriptor waited. For failure, return -1 and set the error number;
- Parameter *fds* is the array of poll file descriptors;
- Parameter *nfds* is the number of elements in *fds* array;
- Parameter *timeout* is the timeout value.

Prototype analysis of Function ppoll:

- For success of the function, return the quantity of descriptor waited. For failure, return -1 and set the error number;
- Parameter *nfds* is the number of elements in *fds* array;
- Parameter *timeout_ts* is the timeout value;
- Parameter *sigmask* is a signal blocked while waiting.

Unlike the select function, the poll function does not construct a file descriptor set for each condition. Instead, it constructs an array of pollfd structure. Each array element specifies a file descriptor number and the conditions for which we are interested in the file descriptor.

```

struct pollfd {
    int      fd;          /* file descriptor being polled */
    short int events;    /* the input event flags */
    short int revents;   /* the output event flags */
};

```

When the poll function is called, events in each *fds* element shall be set as one or more values in Table 5.13. These values tell the kernel which events of the each file descriptor we care about. At return, the revents member is set by the kernel to indicate which events occur in each file descriptor.

3 lines in Table 5.13 represents the readable, writable and exception from top to bottom.

Table 5.13 pollfd event value

Macro name	Note
POLLIN	Can read data beyond high priority without blocking (Equivalent to POLLRDNORM POLLRDBAND)
POLLRDNORM	Can read ordinary data without blocking
POLLRDBAND	Can read priority data without blocking
POLLPRI	Can read data with high priority without blocking
POLLOUT	Can write the ordinary data without blocking
POLLWRNORM	Same with POLLOUT
POLLWRBAND	Can write priority data without blocking
POLLERR	There has been error
POLLHUP	Hanged off

The timeout wait parameters of the poll function are similar to those of the select function which are also divided into three cases. We shall notice that the poll function and the select function will not affect the blocking condition because a file descriptor is blocked.

The behaviors of the ppoll function are similar to those of poll, except that the ppoll function can specify the signal mask word.

5.4.4 File record lock

When a process is reading or modifying a portion of a file, the file record lock can be used to prevent other processes from modifying the same area of the same file. It can be used to lock an area of a file or the entire file. SylixOS supports multiple file record lock API.

Previously, we said that SylixOS supports multiple device driver models. However, only NEW_1 device driver supports file record lock function currently. Such driver file nodes are similar to vnode of the UNIX system.

Firstly, we introduce the fcntl lock with flexible functions. For the function prototype, refer to the fcntl part of Section 5.2.1. When introducing the fcntl function earlier, we mentioned that the fcntl function contains the function to operate the file record lock. This function contains 3 commands: F_GETLK, F_SETLK and F_SETLKW. The 3rd parameter of fcntl record lock is a flock structure pointer.

```
struct flock {
    short    l_type;          /* F_RDLCK, F_WRLCK, or F_UNLCK */
    short    l_whence;       /* flag to choose starting offset */
    off_t    l_start;        /* relative offset, in bytes */
    off_t    l_len;          /* length, in bytes; 0 means
                             /* lock to EOF
    pid_t    l_pid;          /* returned with F_GETLK
    .....
};
```

The meaning of the struct flock member is as follows:

- l_type is the lock type: F_RDLCK (shared read lock), F_WRLCK (exclusive write lock) and F_UNLCK (unlock);
- l_whence value, as shown in Table 5.3;
- l_start is relative to the starting position of l_whence offset (note that it cannot be locked from the beginning of the file);
- l_len is the length of the lock area. If it is 0, the end of file (EOF) is locked. If data is added to the file, it will also be locked.
- l_pid is the process ID preventing the current process from locking (returned by the command F_GETLK).

For the shared read locks and the exclusive write lock mentioned above, the basic rules are: any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Further, if there has already been one or more read locks on a given byte, you cannot add a write lock on the byte; if there is a write lock on a byte, you cannot add any more locks.

The above rules apply to lock requests made by different processes, but do not apply to lock requests proposed by the single process. That is to say, if a process already has a lock in a file range, and the process attempts to add a lock in the same range, it is also OK, and the new lock will replace the existing lock at the moment. Therefore, if a process adds a write lock to a file, and then attempts to add a read lock to the file, it will be executed successfully, and the original write lock will be replaced by the read lock. We will verify this view via Program List 5.16.

In addition, when the read lock is added, the file descriptor must be read open. When the write lock is added, the file descriptor must be write open.

Program List 5.16 Locking in the single process

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    int          fd;
    struct flock fl;
    short        lockt = F_WRLCK;

    fd = open("file", O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "open file failed.\n");
        return -1;
    }

    fl.l_type  = lockt;
    fl.l_whence = SEEK_SET;
    fl.l_start  = 0;
    fl.l_len    = 0;

    if (fcntl(fd, F_SETLK, &fl) < 0) {
        perror("fcntl");
        fprintf(stderr, "add write lock failed.\n");
        close(fd);
        return -1;
    }
    fprintf(stdout, "add write lock success.\n");

    lockt      = F_RDLCK;
    fl.l_type  = lockt;
    fl.l_whence = SEEK_SET;
    fl.l_start  = 0;
    fl.l_len    = 0;

    if (fcntl(fd, F_SETLK, &fl) < 0) {
        perror("fcntl");
        fprintf(stderr, "add read lock failed.\n");
        close(fd);
        return -1;
    }
    fprintf(stdout, "add read lock success.\n");
}
```

```

return 0;
}

```

Run the program under the SylixOS Shell:

```

# ./lock_test
add write lock success.
add read lock success.

```

SylixOS supports lock of the whole file via the traditional BSD function flock to lock the entire file. This function is an old API.

```

#include <sys/file.h>
int flock(int iFd, int iOperation);

```

Prototype analysis of Function flock:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the file descriptor;
- Parameter *iOperation* is the lock type (as shown in Table 5.14).

Calling the flock function locks an open file, and the types of locks supported by this function are shown in Table 5.14. When the process uses the flock function to try to lock the file, if the file has already been locked by other processes, the process will be blocked until the lock is released, or when the flock function is called, the LOCK_NB parameter is used. When the file has been locked by other processes when attempts to lock, an error will return.

The flock function can call the LOCK_UN parameter to release the file lock, or release the file lock by closing fd, which means that the flock lock will be released as the process exits.

Table 5.14 Type of flock lock

Type of flock lock	Note
LOCK_SH	Shared lock, multiple processes can use a lock, commonly used as read lock
LOCK_EX	Exclusive lock, only allow to be occupied by a process, commonly used as write lock
LOCK_NB	Implement non-blocking, blocking at default
LOCK_UN	Unlock

SylixOS supports the lockf function, which is a POSIX lock and can be seen as an package of the fcntl interface.

```
#include <unistd.h>
int lockf(int iFd, int iCmd, off_t oftLen);
```

Prototype analysis of Function lockf:

- For success of the function, return 0. For failure, return -1;
- Parameter *iFd* is the file descriptor;
- Parameter *iCmd* is the lock command;
- Parameter *oftLen* is the locked resource length, started from the current offset.

The behavior of calling the lockf function is basically the same with that of the fcntl function. The file descriptor is input by the lockf function. The lock commands supported by this function are shown in Table 5.15.

Table 5.15 Lockf lock command

Lockf lock command	Note
F_ULOCK	Unlock
F_LOCK	Request the exclusive lock in the blocking way, i.e., write lock
F_TLOCK	Request the exclusive lock in a non-blocking way, i.e., write lock
F_TEST	Get the lock status of the specified file area, and test whether it is locked

5.4.5 File memory mapping

File memory mapping can map a disk file to a memory area in memory space. When the data is fetched from the buffer zone, it is equivalent to read the corresponding byte in the file. Correspondingly, when data is stored in the buffer zone, the corresponding byte is automatically written to the file. This makes it possible to perform I/O operations without using the read function and the write function.

To use this feature, one shall firstly tell the kernel to map a given file into a memory area. This is realized via the mmap function. These technical details will be introduced in Chapter 12 Memory Management.

Chapter 6 Thread management

6.1 Thread

The thread is also called as the task, the instruction stream of a certain single sequence, and it is the smallest unit scheduling of the operating system. A standard thread consists of thread handle (or ID), current instruction pointer (PC), CPU register set and thread stack. Each thread is the scheduling unit of the operating system.

The thread itself has only limited and indispensable resources during operation, such as CPU register, stack and so on. The kernel thread shares all kernel resources, such as the kernel file descriptor table, while the in-process thread shares all resources in the process, such as the process file descriptor table.

A CPU can only run one thread at a moment (multi-CPU system can run multiple threads at the same time). If there are multiple threads in the system, the CPU requires running switch between several threads, equivalent to concurrent execution of multiple threads macroscopically. The correspondence between time and thread of CPU is determined via the scheduling algorithm of the operating system, For example, the time sharing operating system divides time into smaller fragments, called as time slices. After each thread runs for a period, the operating system will command the CPU to switch to another thread for execution. Each thread in the real-time operation system has its own priority. When it is required to execute the thread with high priority, the operating system will immediately switch the current CPU to execute the thread with higher priority, and such dispatching algorithm satisfy demands of the system for real-time signal response.

6.2 Thread state machine

Multiple threads in the same process or kernel can be executed concurrently. However, threads has discontinuity during operation due to mutual restraint between threads. The threads also have three basic states of block, ready and run. Meanings of three states are as follows:

- **Block:** the thread lacks the conditions or resources to make it run, and can enter the ready state after conditions are satisfied.
- **Ready:** The thread already has all resources to make it run, waiting for scheduling of the operating system;
- **Run:** the thread has been scheduled by the operating system (the operating system distributes a CPU to the thread for execution of the thread code).

The thread created in SylixOS system is always at any of these three states. Where, the blockage state is divided into the following due to different reasons: wait for semaphores, wait for messages, sleep and so on. State switching between threads is shown in Figure 6.1.

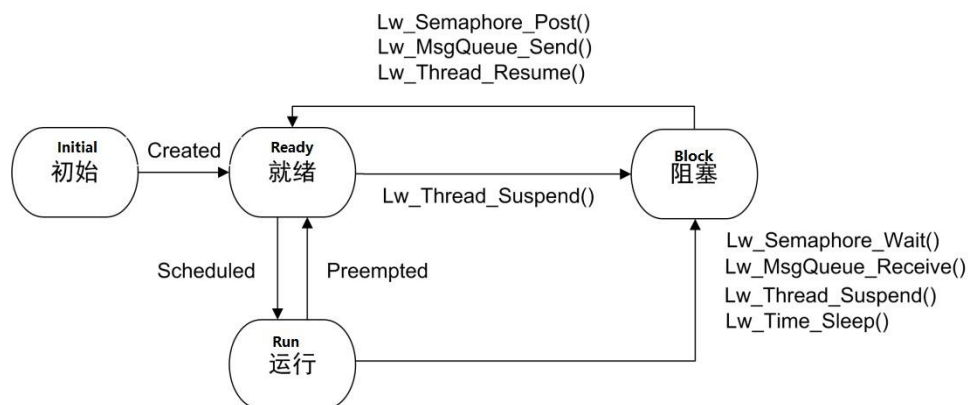


Figure 6.1 State conversion diagram of the thread

In Figure 6.1, the initial state is just a state before the thread was created. Let's see switching relationship between other three states:

- Ready → Run: the thread at the ready state is scheduled by the system, and the right to use CPU is gotten;
- Run → Ready: the thread at the running state is occupied by other threads, or the right to use CPU is abandoned;
- Ready → Block: other threads actively suspend it (not recommended for SylixOS);
- Block → Ready: the waiting resource becomes available;
- Run → Block: waiting for semaphores, receiving messages and sleeping make it blocked.

It is required to indicate that SylixOS does not recommend using the thread suspend function to suspend other threads (a kind of blockage). Therefore, the structured design of the program will be damaged, so that design of the application is complicated and unpredictable, and it is easy to cause design error.

After transition relationships of these states are analyzed, let's see the application interface function in SylixOS causing these changes in state:

Table 6.1 State change function

Function name	State changes
Lw_Thread_Create	Create a thread, and the thread will enter the ready state

Lw_Thread_Init	Initialize a thread, and the thread enters the initial state
Lw_Thread_Start	Enable a thread at the initial state to enter the ready state
Lw_Thread_Suspend	Enable the thread to enter the blocked state (not recommended)
Lw_Thread_Resume	Enable the thread from the blocked state to the ready state (not recommended)
Lw_Thread_ForceResume	Force the thread to enter the ready state (not recommended)
Lw_Thread_Yield	Enable the thread actively abandon CPU to enter the ready state
Lw_Thread_Wakeup	The thread is awakened from the sleep mode into the ready state
Lw_Semaphore_Wait	Block thread
Lw_Semaphore_Post	Recover the thread from the blocked state to the ready state
Lw_SemaphoreC_Wait	Block thread
Lw_SemaphoreC_Post	Recover the thread from the blocked state to the ready state
Lw_SemaphoreB_Wait	Block thread
Lw_SemaphoreB_Post	Recover the thread from the blocked state to the ready state
Lw_SemaphoreM_Wait	Block thread
Lw_SemaphoreM_Post	Recover the thread from the blocked state to the ready state
Lw_MsgQueue_Receive	Block thread
Lw_MsgQueue_Send	Recover the thread from the blocked state to the ready state
Lw_Time_Sleep	Thread sleep enters the blocked state
Lw_Time_SSleep	Thread sleep enters the blocked state
Lw_Time_MSsleep	Thread sleep enters the blocked state

6.3 SylixOS thread

SylixOS is the multi-thread operating system, which can create multiple threads at the same time. The specific maximum number of threads depends on the size of the system memory and related configuration when compilation the SylixOS operating system^①.

6.3.1 Thread creation

1. Thread attribute creation

Each SylixOS thread has its own attribute, including priority of the thread, stack information, thread parameters and so on.

```
#include <SylixOS.h>
ULONG Lw_ThreadAttr_Build(PLW_CLASS_THREADATTR pthreadattr,
                          size_t stStackSize,
                          UINT8 ucPriority,
                          ULONG ulOption,
                          PVOID pvArg);
```

Prototype analysis of Function `Lw_ThreadAttr_Build`:

- For success of the function, return `ERROR_NONE`. For failure, return the error number;
- Output parameter ***pthreadattr*** returns the generated attribute block;

Each thread attribute block consists of the structure `LW_CLASS_THREADATTR`. The structure members are as follows:

```
typedef struct {
    PLW_STACK      THREADATTR_pstkLowAddr;    /* Stack low memory start
address          */
    size_t         THREADATTR_stGuardSize;    /* Stack alert area size */
    size_t         THREADATTR_stStackSize; /* Total stack size (bytes)*/
    UINT8         THREADATTR_ucPriority;    /* Thread priority */
    ULONG         THREADATTR_ulOption;     /* Task options */
    PVOID         THREADATTR_pvArg;       /* Thread parameters */
    PVOID         THREADATTR_pvExt;      /* Extended data segment
pointer          */
} LW_CLASS_THREADATTR;
```

For users of SylixOS application development, it is only required to care about the bold part, and the operating system will set by default for other members.

- Parameter ***stStackSize*** is the stack size (byte);
- Parameter ***ucPriority*** is the thread priority;

In order to use thread priority reasonably, SylixOS sets some common priority values (the smaller the value, the higher the priority of the thread, and the system has priority scheduling), as follows^①.

Table 6.2 Thread priority macro (part)

Macro name	Value
<code>LW_PRIO_HIGHEST</code>	0
<code>LW_PRIO_LOWEST</code>	255
<code>LW_PRIO_EXTREME</code>	<code>LW_PRIO_HIGHEST</code>
<code>LW_PRIO_CRITICAL</code>	50
<code>LW_PRIO_REALTIME</code>	100
<code>LW_PRIO_HIGH</code>	150
<code>LW_PRIO_NORMAL</code>	200
<code>LW_PRIO_LOW</code>	250
<code>LW_PRIO_IDLE</code>	<code>LW_PRIO_LOWEST</code>

In application development, the task priority shall be between LW_PRIO_HIGH and LW_PRIO_LOW generally, so as not to influence the system kernel threads as much as possible.

- Parameter *ulOption* is the thread option;
- Parameter *pvArg* is the thread parameter;

Table 6.3 Thread option

Macro name	Explanation
LW_OPTION_THREAD_STK_CHK	Inspect the thread stack during running
LW_OPTION_THREAD_STK_CLR	Zero the data during thread creation
LW_OPTION_THREAD_USED_FP	Save the floating-point arithmetic unit
LW_OPTION_THREAD_SUSPEND	Create thread rear blockage
LW_OPTION_THREAD_INIT	Initialize the thread
LW_OPTION_THREAD_SAFE	The thread created is the safe mode
LW_OPTION_THREAD_DETACHED	The thread shall be free of merger
LW_OPTION_THREAD_UNSELECT	The thread does not use the select function
LW_OPTION_THREAD_NO_MONITOR	The kernel tracker does not work on the thread
LW_OPTION_THREAD_UNPREEMPTIVE	The task cannot be occupied (not supported at present)
LW_OPTION_THREAD_SCOPE_PROCESS	Competition in the progress (not supported at present)

Note: SylixOS provides a quick function to get the default attribute block of the system, `LwStud _ ThreadAttrStup _ GetDefault`, and the return value of the function is the thread attribute block. For the attribute block, the default setting size of the thread stack is 4K, the priority is LW_PRIO_NORMAL, and the option is LW_OPTION_THREAD_STK_CHK. The reader can appropriately modify the returned attribute block, and modify the thread option (value assignment between options in the form of "or") and parameter usually.

SylixOS provides the following set of functions to modify the thread's attribute blocks:

```
#include <SylixOS.h>
ULONG Lw_ThreadAttr_SetGuardSize(PLW_CLASS_THREADATTR pthreadattr,
                                size_t stGuardSize);
ULONG Lw_ThreadAttr_SetStackSize(PLW_CLASS_THREADATTR pthreadattr,
                                size_t stStackByteSize);
ULONG Lw_ThreadAttr_SetArg(PLW_CLASS_THREADATTR pthreadattr,
                           PVOID pvArg);
```

The `Lw_ThreadAttr_SetGuardSize` function modifies the size of the stack alerting area of the thread attribute block, and the parameter `stGuardSize` appoints the size of the new stack alerting area. The `Lw_ThreadAttr_SetStackSize` function modifies the stack

size of the thread attribute block, and the parameter **stStackSize** appoints the new stack size. The `Lw_ThreadAttr_SetArg` function can set the thread's startup parameter **pvArg**.

2. Thread stack

Each thread has its own stack area. These areas are used for thread's function call, distribution of automatic variable, function return value and so on. Each thread control block saves the initial position, terminal position and stack warning point (for stack overflow check) of the stack area. The thread is the basic unit of SylixOS scheduling. When task scheduling occurs, the thread stack area will save the thread's current environment (for recovery in the context). Therefore, setting of the thread stack must be reasonable. If it is too big, waste of memory space will be caused. If it is too small, stack overflow will be caused. All threads in SylixOS run in the same address space. For real-time requirements, there is no address protection mechanism between threads. Therefore, stack overflow will cause unpredictable errors.

There is no formula applied mechanically for setting of the stack size. Generally, one can set a large value according to experience, store space to replace reliability, and use the **ss** command to view usage of each task stack in the shell environment.

3. Thread creation

```
#include <SylixOS.h>
LW_HANDLE Lw_Thread_Create(CPCHAR          pcName,
                          PTHREAD_START_ROUTINE pfuncThread,
                          PLW_CLASS_THREADATTR pthreadattr,
                          LW_OBJECT_ID        *pulId);
```

Prototype analysis of Function `Lw_Thread_Create`:

- For success of the function, return a thread ID created successfully (LW_HANDLE type). For failure, return the error number;
- Parameter **pcName** is the thread name;
- The parameter **pfuncThread** is the thread entry function, i.e., the starting address of the thread code segment;
- Parameter **pthreadattr** is the attribute block pointer of the thread (when NULL, the default property block will be used);
- The parameter **pulId** is the pointer of the thread ID, and the content is the same with the return value. It can be NULL.

The function can create an SylixOS thread. Let's take a look at how SylixOS creates a thread via the following example.

Program List 6.1 Thread creation instance

```

#include <SylixOS.h>

PVOID tTest (PVOID pvArg)
{
    while (1) {
        printf("thread running...\n");
        sleep(1);
    }
}

int main (int argc, char *argv[])
{
    LW_CLASS_THREADATTR    threadattr;
    LW_HANDLE               hThreadId;

    Lw_ThreadAttr_Build(&threadattr,
                       4 * LW_CFG_KB_SIZE,
                       LW_PRIO_NORMAL,
                       LW_OPTION_THREAD_STK_CHK,
                       LW_NULL);

    hThreadId = Lw_Thread_Create("t_test", tTest, &threadattr, LW_NULL);
    if (hThreadId == LW_OBJECT_HANDLE_INVALID) {
        return (PX_ERROR);
    }

    return (ERROR_NONE);
}

```

Run the program under the SylixOS Shell:

```

#./thread_test
thread running...
thread running...
#ts
thread show >>

      NAME          TID    PID  PRI  STAT  ERRNO    DELAY  PAGEFAILS  FPU  CPU
-----
.....
thread_test      4010033  10   200  SEM        0         0         1     0
t_test           4010034  10   200  SLP       71        88         0     0
.....

```

Output results of the program are printed every 1 second as expected. From output results of the `ts` command, it can be seen that the "t_test" thread has been created (ID: 4010034 priority: 200), indicating that our thread is created successfully.

This program uses two functions above mentioned and creation of thread attributes, and distributes 4 * LW_CFG_KB_SIZE (LW_CFG_KB_SIZE is the built-in macro of SylixOS system, and the value is 1024) stack size, thread priority of LW_PRIO_NORMAL and thread option of LW_OPTION_THREAD_STK_CHK, and there is no thread parameters. The name of the created thread is "t_test". This thread does not do any substantial thing, but is only a simple print. However, it is enough to explain the process and method of creating the SylixOS thread.

4. Thread initialization

```
#include <SylixOS.h>
LW_HANDLE Lw_Thread_Init(CPCHAR          pcName,
                        PTHREAD_START_ROUTINE pfuncThread,
                        PLW_CLASS_THREADATTR pthreadattr,
                        LW_OBJECT_ID         *pulId);
ULONG Lw_Thread_Start(LW_OBJECT_HANDLE ulId);
```

Prototype analysis of Function Lw_Thread_Init:

- For success of the function, return the threadID. For failure, return the error number;
- Parameter *pcName* is the thread name;
- Parameter *pfuncThread* is the entry function of the thread;
- Parameter *pthreadattr* is the thread attribute (when NULL, the default property block will be used);
- Parameter *pulId* is the ID pointer, and can be NULL;

Prototype analysis of Function Lw_Thread_Start:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulId* is the thread ID;

The Lw_Thread_Init function will create a thread like the Lw_Thread_Create function. However, there is an essential difference between both functions, i.e., the thread created by the Lw_Thread_Init function is only at an initial state. The thread is not ready. The scheduler will not allocate CPU use right to the thread. Only when Lw_Thread_Start function is called, the thread can be at the ready state, and can be scheduled by the scheduler.

The following program shows how to use the thread initialize function.

Program List 6.2 Thread initialization instance

```

#include <SylixOS.h>

PVOID tTest (PVOID pvArg)
{
    while (1) {
        printf("Thread running...\n");
        sleep(1);
    }
    return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_CLASS_THREADATTR    threadattr;
    LW_HANDLE               hThreadId;
    INT                     iRet;

    Lw_ThreadAttr_Build(&threadattr,
                       4 * LW_CFG_KB_SIZE,
                       LW_PRIO_NORMAL,
                       LW_OPTION_THREAD_STK_CHK,
                       LW_NULL);

    hThreadId = Lw_Thread_Init("t_test", tTest, &threadattr, LW_NULL);
    if (hThreadId == LW_OBJECT_HANDLE_INVALID) {
        return (PX_ERROR);
    }

    iRet = Lw_Thread_Start(hThreadId);
    if (iRet) {
        return (PX_ERROR);
    }
    return (ERROR_NONE);
}

```

Run the program under the SylixOS Shell:

```

#./thread_init
Thread running...
Thread running...
#ts
thread show >>

      NAME          TID      PID  PRI  STAT  ERRNO      DELAY  PAGEFAILS  FPU  CPU
-----
.....

```


SpaceChain OS								
thread_init	4010014	1	200	SEM	0	0	1	
0								
t_test	4010015	1	200	SLP	71	68	0	0
.....								

Output results of the program are printed every 1 second as expected. From output results of the `ts` command, it can be seen that the "t_test" thread has been created (ID: 4010015 priority: 200), indicating that our thread is created successfully.

Comparing with Program List 6.1, we find that operation results of the two programs are the same, and properties of the thread created are also the same. Therefore, from the perspective of running behavior, effect of Program List 6.2 is the same with that of Program List 6.1. We will use the method in Program List 6.2 to actively control our threads to enter the ready state, and the method is very useful for some situations.

Note: it might also be noted that all created functions introduced above cannot be called in the interrupt context.

6.3.2 Thread control

1. Thread suspend and resume

The thread suspend is to make the specified thread at the non-ready state. The thread at the suspend state is ignored by the scheduler and is relatively "quiet" for debugging until the suspend is released.

```
#include <SylixOS.h>
ULONG  Lw_Thread_Suspend(LW_OBJECT_HANDLE  uId);
ULONG  Lw_Thread_Resume(LW_OBJECT_HANDLE  uId);
```

Prototype analysis of Function `Lw_Thread_Suspend`:

- For success of the function, return `ERROR_NONE`. For failure, return the error number;
- Parameter *uId* is the thread ID;

Prototype analysis of Function `Lw_Thread_Resume`:

- For success of the function, return `ERROR_NONE`. For failure, return the error number;
- Parameter *uId* is the thread ID;

Suspend is a binary memoryless state of a task. It will not check whether the task was suspended or released before prior to suspend, so:

- Effects of repeated suspend of a certain task are the same with those of one-time suspend;

- If suspend and release suspend are completed by different tasks, one must ensure that they are conducted in the correct order;

Let's see the following scenario, thread T1 sends a message to a certain thread T2 before suspend, and T2 releases the suspend state of T1 thread after receiving this message.

```
.....
T1: Send message to T2 thread
T1: 调用 Lw_Thread_Suspend 挂起自己
.....
T2: Received T1 message
T2: Call Lw_Thread_Resume to release T1 pending state
.....
```

We will find that there is a competitive risk after careful analysis of above scenario. T1 just sends the message. At this time, T2 with high priority receives the message to release thread suspend. Release has no effect at the moment, and T1 starts to suspend, thus entering the unlimited suspend state. We shall be careful about this situation.

In addition, we must also pay attention to avoid deadlock of the system at suspend, which usually requires suspend of the thread after obtaining a certain system resource with mutex access. One shall especially avoid such situation at asynchronous suspend.

To avoid the competition risk, we can attach the suspend state to the delayed state and the blocked state, causing the thread to enter the "delayed suspend" or "blocked suspend" state. The additional suspend state does not influence the original delay and block of the thread, which means that the suspend state and delayed or blocked state can coexist.

- During suspension, the delay thread still calculates the delay. If the delay expires, the task enters the suspend-only state.
- If the waiting condition is satisfied during suspension, the blocked thread will be unblocked, and enter the suspend-only state.
- If the thread is de-suspended when the delay expires or when the waiting condition occurs, the thread will returns to the original delay / block state.

Note: Lw_Thread_Suspend function and Lw_Thread_Resume function can be called during interrupt, and thread suspend is unconditional. To check the thread state at the moment in SylixOS, one can view "STAT" column via Shell command `ts`. As shown below, thread "t_test" is at the "SLP" state.

```
# ts
thread show >>

      NAME          TID    PID  PRI  STAT  ERRNO    DELAY    PAGEFAILS  FPU  CPU
-----
```

```

.....
t_test          .....          SLP          .....
.....

```

2. Thread delay

The thread delay is to let the thread sleep, so that the scheduler can schedule other threads, and resume operation after thread sleep.

```

#include <SylixOS.h>
VOID Lw_Time_Sleep(ULONG ulTick);
VOID Lw_Time_SSleep(ULONG ulSeconds);
VOID Lw_Time_MSsleep(ULONG ulMSeconds);

```

Function prototype analysis^①:

- Lw_Time_Sleep delay unit is Tick^②;
- Lw_Time_SSleep delay unit is s
- Lw_Time_MSsleep delay unit is ms;

The minimum time delay which can be obtained by using Lw_Time_Sleep series function in SylixOS is 1Tick, If you specify *ulTick* to be 0, SylixOS will not delay, it will not affect the current thread's behavior. If you want a shorter time delay (for example: 1ms). The nanosleep function can be called.

```

#include <time.h>
int nanosleep(const struct timespec *rqtp,
              struct timespec *rmtp);

```

Prototype analysis of Function nanosleep:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *rqtp* is the sleep time;
- Parameter *rmtp* saves the remaining time;

The function belongs to POSIX standard. The required wait time is represented by the pointer *rqtp* of the structure timespec. The structure represents time in seconds and nanoseconds. Unlike the above three functions, if *rmtp* is not NULL, the remaining time is returned via it. It might also be noted that this function can be awakened by the signal. If so, the error number errno is EINTR. For a detailed explanation of the signal, see Chapter 10 Signal System.

Note: If no signal is interrupted during nanosleep sleep, rmtp always returns 0. Otherwise, it will return the time interval from the signal interrupt time point to delay completion.

3. Thread mutex

Mutex access is a classical theoretical problem in the operating system. It is used to implement consistent access to shared resources. SylixOS implements different functions to provide multiple mutex mechanisms.

- Thread lock: `Lw_Thread_Lock`.
- Interrupt lock: `Lw_Interrupt_Lock`.
- Semaphore: `Lw_Semaphore_Wait`.

When the time for shared resource accessed is very long, semaphore method is very effective. For example, when a thread wants to apply for a shared area locked by semaphore, then this thread will be blocked at the moment, so as to save CPU cycle. For detailed use of semaphore, see Chapter 7 Inter-thread Communication.

Using the lock interrupt method will increase delay in interrupt response of the system. For general threads, lock interrupt is not a good method, and it is not suggested to adopt lock interrupt for general application development SylixOS.

Let's take a look at the thread lock. In SylixOS, we can call the thread lock function to disable the scheduler. When the thread calls the thread lock function, the scheduler temporarily fails. Even if a high-priority thread is ready, the thread will not be called out of the processor until the thread calls the thread release function. This mutex scheme introduces a priority delay (scheduling delay) for the system; the high priority thread which requires high real-time performance must wait until the thread is unlocked before being scheduled. Therefore, superior real-time performance of SylixOS is sacrificed to some extent. Therefore, this method is also not recommended.

Based on multiple considerations, the semaphore method is generally adopted for SylixOS application development to achieve mutex access.

Note: the thread lock function only locks the current CPU's schedule, but does not affect the other CPU's schedule. The blocking function cannot be used during thread locking. Thread locking does not lock interrupt. When interrupt occurs, the interrupt service routine is called as usual.

6.3.3 End of thread

End of thread means the end of the thread life cycle. End of thread includes 3 cases of operation end and exit, thread exit and thread delete.

1. Thread delete

Thread delete is to return the thread resources to the operating system, and the deleted thread can no longer be scheduled.

```
#include <SylixOS.h>
ULONG Lw_Thread_Delete(LW_OBJECT_HANDLE *pulId, PVOID pvRetVal);
ULONG Lw_Thread_ForceDelete(LW_OBJECT_HANDLE *pulId, PVOID pvRetVal);
```

Function prototype analysis:

- For success of two functions, return ERROR_NONE. For failure, return the error number;
- Parameter *pullId* is the handle of the thread to be deleted;
- Parameter *pvRetVal* is the value returned to the JOIN function;

Calling above two functions can make the thread end, and release thread resources. Because SylixOS supports the process, the delete thread can only be a thread in the same process, and the main thread can only be deleted by itself.

Active deletion of other executing threads may cause that unlocked resources cannot be released or atomic operations are interrupted. Therefore, it is not recommended to directly use thread delete function for call in SylixOS and any other operating system unless security is guaranteed. During application design, "request" delete mode shall be considered. When the thread finds that it has nothing to do or is requested to delete, the thread deletes itself (thread exit).

2. Thread exit

```
#include <SylixOS.h>
ULONG Lw_Thread_Exit(PVOID pvRetVal);
```

Prototype analysis of Function Lw_Thread_Exit:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *pvRetVal* is the value returned to the JOIN function.

3. Thread cancel

```
#include <SylixOS.h>
ULONG Lw_Thread_Cancel(LW_OBJECT_HANDLE *pullId);
```

Prototype analysis of Function Lw_Thread_Cancel:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *pullId* is the canceled thread ID.

The method to cancel the thread is to send the Cancel signal to the target thread, but how to deal with the Cancel signal is determined by the target thread itself. It might also be noted that the thread cancel is a complicated process, and consistency of the resource shall be considered. For detailed information of thread cancel, see 6.4.4 Thread Cancel.

6.3.4 Multi-thread security

The inherent advantages of the multi-thread model make SMP multi-core processor to realize true concurrent execution. However, multi-thread brings convenience and introduces some problems, such as the mutex access of global resources. For safe access to these resources, the competition conditions and deadlock shall be considered during program design.

Multi-thread security is a mechanism where a resource can be safely used by multiple threads in the case of concurrent execution of multiple threads. Multi-thread security includes the protection and reentrancy of critical section of code. The critical section of code refers to code which is indivisible when processed. Once execution of this part of the code is started, no interruption is allowed. To ensure that execution of the critical section code is not interrupted, interrupt must be closed before entering the critical section, and interrupt must be immediately opened after the critical section code is executed.

"Reentrability"^① means that the function can be called by multiple threads without destroying the data. Regardless of how many threads are used, the reentrant function always gets the expected results for each thread. The function which allows reentry is called as "reentrant function". Otherwise, it is the "non-reentrant function".

The problem of code reentrability is caused by parallel operation of multiple threads. Therefore, code reentrability is called as "multi-thread security".

In SylixOS, occasions which may cause code reentry include: multi-thread scheduling, interrupt service program scheduling and signal processing function scheduling.

The following two functions are considered:

```
char *ctime (const time_t *time)
{
    static char cTimeBuffer[sizeof(ASCBUF)];
    .....                               /* Write a string to the
buffer */
    return (cTimeBuffer);
}
char *ctime_r (const time_t *time, char *buffer)
{
    .....                               /* Write a string to the
buffer */
    return (buffer);
}
```

Two functions will convert the time represented by Parameter time to a character string for return. The ctime function defines the character string as a local static buffer zone. Considering that multiple threads call ctime "simultaneously", call of different threads will cause the ctime function to modify the same character string buffer area apparently. Therefore, the ctime function is the non-reentrant function. In contrast, the

character string buffer area of the `ctime_r` function is allocated by the caller, and different threads run to modify their respective buffer areas. Therefore, it is safe for multi-thread call.

Some functions will inevitably use global or static variables, such as the `malloc` function. In order to protect global variables from being destroyed, mutex is an option.

In addition to the method introduced above to implement function reentrancy, SylixOS also provides a "thread private data" mechanism to implement function reentry. Such protection sacrifices real-time performance of the system, and is only effective for single CPU system. It is not recommended to adopt the method in SylixOS unless necessary.

Thread private data is an unsigned long value of the thread context record (which can be seen as a pointer to a global variable) and a temporary variable which saves the global variable value (used to restore the value of the global variable in the thread context). Every time the thread is called into the processor, the system automatically loads the value of the global variable from the thread context according to the pointer. Correspondingly, when the task is called out of the processor, the system automatically saves the value of the global variable to the thread context according to the pointer.

The following group of functions implement operation of thread private data.

```
#include <SylixOS.h>
ULONG Lw_Thread_VarAdd(LW_HANDLE uId, ULONG *pulAddr);
ULONG Lw_Thread_VarDelete(LW_HANDLE uId, ULONG *pulAddr);
ULONG Lw_Thread_VarSet(LW_HANDLE uId, ULONG *pulAddr, ULONG ulValue);
ULONG Lw_Thread_VarGet(LW_HANDLE uId, ULONG *pulAddr);
ULONG Lw_Thread_VarInfo(LW_HANDLE uId, ULONG *pulAddr[], INT iMaxCounter);
```

Function prototype analysis:

- Parameter *uId* is the thread handle;
- Parameter *pulAddr* is the private data address;
- Parameter *ulValue* is the set value;
- Parameter *iMaxCounter* is the size of address list.

Calling `Lw_Thread_VarAdd` function can declare a thread private data. For success of the function, return `ERROR_NONE`. For failure, return the error number; Calling `Lw_Thread_Delete` function can declare a thread private data declared. For success of the function, return `ERROR_NONE`. For failure, return the error number; Calling `Lw_Thread_VarSet` function can set the value of the thread private data; calling `Lw_Thread_VarGet` function can get the value of the thread private data. The function returns the value of private data or 0; Calling `Lw_Thread_VarInfo` function will get information of the thread private data. The function returns the number of private data.

The process for the thread private data to implement reentrant is as follows:

```

INT  _G_iGlobal;
VOID func (VOID)
{
    .....

    if (Lw_Thread_VarAdd(threadId, (ULONG *)&_G_iGlobal) != 0) {
        .....
        /* Error handling
    */
    }
    _G_iGlobal++;
    /* Global variable
processing */
    .....
}

```

Obviously, for each new thread generated based on the func function, the global variable iGlobal will be added. After the thread private data is declared, the system will save and load the duplicate of global variable iGlobal owned by each thread when the thread context switches, so that modification of global variable iGlobal by different threads will not affect each other, as shown in Figure 6.2.

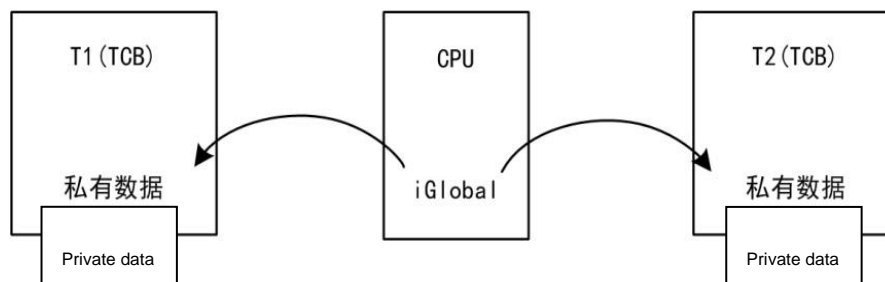


Figure 6.2 Thread private data

It might also be noted that the thread private data must be declared before any value assignment.

The Lw_Thread_VarSet function and the Lw_Thread_VarGet function are usually used to allow multiple cooperative threads participating in a job to obtain the thread private data values of other threads. It has the function of inter-thread communication. Program List 6.3 shows an instance of implementation of inter-thread communication by the private data.

Program List 6.3 Private data implements inter-thread communication

```

#include <SylixOS.h>
#include <stdio.h>

INT  _G_iGlobal = 0;

PVOID tTest (PVOID pvArg)

```



```
{
while (1) {
    fprintf(stdout, "tTest global value: %d\n", _G_iGlobal);
    sleep(1);
}

return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_HANDLE hId;

    hId = Lw_Thread_Create("t_test", tTest, NULL, NULL);
    if (hId == LW_HANDLE_INVALID) {
        return (PX_ERROR);
    }

    if (Lw_Thread_VarAdd(hId, (ULONG *)&_G_iGlobal) != 0){
        return (PX_ERROR);
    }

    while (1) {
        Lw_Thread_VarSet(hId, (ULONG *)&_G_iGlobal, 55);
        sleep(1);
    }

    Lw_Thread_Join(hId, NULL);

    return (ERROR_NONE);
}
```

Run the program under the SylixOS Shell, and the results are as follows:

```
# ./var_test
tTest global value: 55
tTest global value: 55
```

Each thread private data added will cause thread context switch, increasing unsigned long memory copy overhead, which is the flaw of thread private data. Compared to the previous method based on local dynamic variables, there are also limitations. When it is required to span the function scope, the global variables must be used. However, the commonly used method of "global variables + mutex" is more complicated than the method of thread private data, and mutex also requires a certain running overhead.

6.4 POSIX thread

6.4.1 Thread attribute

All POSIX thread attributes are represented via an attribute object defined as the structure `pthread_attr_t`. POSIX defines a series of function settings and read thread attribute values. Therefore, application does not need to know details of `pthread_attr_t` definition.

Management of thread attributes shall comply with the same mode usually:

- Each object is associated with the attribute object of its own type (thread and thread properties, mutex amount and mutex attribute and so on, and one attribute object contains multiple attributes (`pthread_attr_t`). Encapsulation of the attribute object makes application easier to transplant;
- The attribute object has an initialization function which sets the attribute as the default value;
- There is a destroy function corresponding to initialization for deinitialization;
- Each attribute object has a function which gets the attribute value from the attribute object. For success of the function, return 0. For failure, return the error number. Therefore, it can be returned to the caller via storing the attribute value in the memory unit appointed by a certain parameter of the function;
- Each attribute object has a function which sets the attribute value, so that the attribute value is transferred via the parameter.

Calling the `pthread_attr_init` function can initialize a thread attribute object, and calling the `pthread_attr_destroy` function can destroy a thread attribute object.

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *pattr);
int pthread_attr_destroy(pthread_attr_t *pattr);
```

Prototype analysis of Function `pthread_attr_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer requiring initialization;

Prototype analysis of Function `pthread_attr_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer requiring destroy.

Initialization of the attribute object assigns default values to various attributes, and POSIX does not restrict the default value. After the attribute object is initialized, the program usually needs to set the suitable value for single attribute.

Destroy attribute is an inverse process of attribute initialization, which usually sets the attribute value as the invalid value. If some system resources is dynamically distributed during initialization, these resources will be released during destroy. Only the thread option value is set as the invalid value in SylixOS. The attribute object cannot be used for thread creation after destroy unless reinitialization.

The `pthread_getattr_np` function can be called to get a attribute object of a appointed thread. The function is not a part of the POSIX standard, but a kind of extension of the Linux and SylixOS systems. SylixOS also supports the FreeBSD extension function `pthread_attr_get_np`.

```
#include <pthread.h>
int pthread_attr_get_np(pthread_t thread, pthread_attr_t *pattr);
int pthread_getattr_np(pthread_t thread, pthread_attr_t *pattr);
```

Prototype analysis of Function `pthread_attr_get_np`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *thread* is the thread handle;
- The output parameter *pattr* is the attribute object of the return thread.

The `pthread_attr_get_np` and `pthread_getattr_np` functions have the same functions and parameter types, and the `pthread_getattr_np` function calls the `pthread_attr_get_np` function on underlying implementation.

Calling the following functions can get or set the name of the POSIX thread.

```
#include <pthread.h>
int pthread_attr_setname(pthread_attr_t *pattr, const char *pcName);
int pthread_attr_getname(const pthread_attr_t *pattr, char **ppcName);
```

Prototype analysis of Function `pthread_attr_setname`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer;
- Parameter *pcName* is the thread name to be set.

Prototype analysis of Function `pthread_attr_getname`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer;
- The output parameter *ppcName* is the name of the return thread.

Calling the `pthread_attr_setname` function can set the name of the thread, and calling the `pthread_attr_getname` function can get the name of the thread. The default thread name for initializing the thread attribute object in SylixOS is "pthread".

Thread attributes specified by POSIX are as follows.

1. Stack size

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *pattr, size_t stSize);
int pthread_attr_getstacksize(const pthread_attr_t *pattr, size_t *pstSize);
```

Prototype analysis of Function `pthread_attr_setstacksize`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer;
- Parameter *stSize* is the stack size.

Prototype analysis of Function `pthread_attr_getstacksize`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer;
- The output parameter *pstSize* returns the stack size.

Calling the `pthread_attr_setstacksize` function can set the size of the thread stack, and calling the `pthread_attr_getstacksize` function will get the stack size of the appointed thread. The default stack value for initializing the thread attribute object in SylixOS is 0, which means that the stack size will inherit that of the creator.

Note: The stack size set shall not be less than 128 bytes^①. Otherwise, EINVAL error value will be returned.

2. Stack address

```
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t *pattr, void *pvStackAddr);
int pthread_attr_getstackaddr(const pthread_attr_t *pattr,
                             void **ppvStackAddr);
```

Prototype analysis of Function `pthread_attr_setstackaddr`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pattr* is the thread attribute object pointer;
- Parameter *pvStackAddr* is the stack address.

Prototype analysis of Function `pthread_attr_getstackaddr`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute object pointer;
- The output parameter ***ppvStackAddr*** is the the return address of the stack.

Calling the `pthread_attr_setstackaddr` function can set the new starting address of the stack, and calling the `pthread_attr_getstackaddr` function will get the starting address of the stack. The default stack address for initializing the thread attribute object in SylixOS is `LW_NULL`, which means that the system will automatically distribute the stack space.

```
#include <pthread.h>
int pthread_attr_setstack(pthread_attr_t *pattr,
                          void          *pvStackAddr,
                          size_t        stSize);
int pthread_attr_getstack(const pthread_attr_t *pattr,
                          void              **ppvStackAddr,
                          size_t            *pstSize);
```

Prototype analysis of Function `pthread_attr_setstack`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- Parameter ***pvStackAddr*** is the stack address;
- Parameter ***stSize*** is the stack size.

Prototype analysis of Function `pthread_attr_getstack`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- The output parameter ***ppvStackAddr*** is the the return address of the stack;
- The output parameter ***pstSize*** is the size of the return stack.

Calling the `pthread_attr_setstack` function can set the starting address and the size of the stack simultaneously, and calling the `pthread_attr_getstack` function will get the starting address and the size of the stack simultaneously.

3. Guard area of the thread stack

```
#include <pthread.h>
int pthread_attr_setguardsize(pthread_attr_t *pattr, size_t stGuard);
int pthread_attr_getguardsize(pthread_attr_t *pattr, size_t *pstGuard);
```

Prototype analysis of Function `pthread_attr_setguradsize`:

- For success of the function, return 0. For failure, return the error number;

- Parameter ***pattr*** is the thread attribute pointer;
- Parameter ***stGuard*** is the size of the stack guard area.

Prototype analysis of Function `pthread_attr_getguardsize`:

- For success of the function, return 0. For failure, return the error number;
- Output parameter ***pstGuard*** is the size of the return stack guard area.

Calling the `pthread_attr_setguardsize` function can set the size of the stack guard area, and calling the `pthread_attr_getguardsize` function will get the size of the stack guard area. The size of the default stack guard area for initializing the thread attribute object in SylixOS is `LW_CFG_THREAD_DEFAULT_GUARD_SIZE`.

4. Detach status

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *pattr, int iDetachState);
int pthread_attr_getdetachstate(const pthread_attr_t *pattr,
                               int *piDetachState);
int pthread_detach(pthread_t thread);
```

Prototype analysis of Function `pthread_attr_setdetachstate`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- Parameter ***iDetachState*** is the detach state value.

Prototype analysis of Function `pthread_attr_getdetachstate`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- The output parameter ***piDetachState*** is the return detach state value.

Prototype analysis of Function `pthread_detach`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***thread*** is the thread handle to be detached.

The thread detach states is divided into the join state and the separate state. At the join state, the thread creates a new thread to block itself until the new thread exits.

Calling the `pthread_attr_setdetachstate` function can set the detach state of the thread attribute object, and calling the `pthread_attr_getname` function will get the detach state of the thread attribute object. The default detach state for initializing the thread attribute object in SylixOS is the join state (`PTHREAD_CREATE_JOINABLE`).

If the thread is created in the join state, the thread can call the `pthread_detach` function to make it into a separate state. Otherwise, it is infeasible.

5. Inherit scheduling

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *pattr, int iInherit);
int pthread_attr_getinheritsched(const pthread_attr_t *pattr,
                                int *piInherit);
```

Prototype analysis of Function `pthread_attr_setinheritsched`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- Parameter ***iInherit*** is the inherit attribute.

Prototype analysis of Function `pthread_attr_getinheritsched`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;

Inherit scheduling determines whether to inherit the scheduling parameter from the parent thread (`PTHREAD_INHERIT_SCHED`) or be appointed explicitly (`PTHREAD_EXPLICIT_SCHED`) when creating the thread.

Calling the `pthread_attr_setinheritsched` function can set inheritance of the thread attribute object, and calling the `pthread_attr_getinheritsched` function will get the scheduling strategy of the thread attribute object (inheritance). The default scheduling strategy for initializing the thread attribute object is explicitly appointed.

6. Dispatching strategy

```
#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *pattr, int iPolicy);
int pthread_attr_getschedpolicy(const pthread_attr_t *pattr, int *piPolicy);
```

Prototype analysis of Function `pthread_attr_setschedpolicy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- Parameter ***iPolicy*** is the scheduling strategy.

Prototype analysis of Function `pthread_attr_getschedpolicy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;

- The output parameter ***piPolicy*** returns the scheduling strategy.

The item appoints the scheduling strategy to create the new thread, including SCHED_FIFO, SCHED_RR and SCHED_OTHER (SCHED_OTHER is the custom scheduling strategy specified by POSIX, and SCHED_OTHER is equal to SCHED_RR in SylixOS.). For detailed contents of these two scheduling strategies, see Section 6.6.

Calling the pthread_attr_setschedpolicy function can set the scheduling strategy of the thread attribute object, and calling the pthread_attr_getschedpolicy function will get the scheduling strategy of the thread attribute object. The default scheduling strategy for initializing the thread attribute object in SylixOS is SCHED_RR.

7. Scheduling parameter

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t          *pattr,
                               const struct sched_param *pschedparam);
int pthread_attr_getschedparam(const pthread_attr_t    *pattr,
                               struct sched_param      *pschedparam);
```

Prototype analysis of Function pthread_attr_setschedparam:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- The output parameter ***pschedparam*** is the scheduling parameter;

Prototype analysis of Function pthread_attr_getschedparam:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pattr*** is the thread attribute pointer;
- The output parameter ***pschedparam*** returns the scheduling parameter.

After the thread is created successfully, the scheduling parameter is allowed to be modified dynamically. For detailed contents of the scheduling parameter, see Section 6.7.

Calling the pthread_attr_setschedparam function can set the scheduling parameter of the thread attribute object, and calling the pthread_attr_getschedpolicy function will get the scheduling parameter of the thread attribute object. The default scheduling parameter for initializing the thread attribute object in SylixOS only sets the thread priority as LW_PRIO_NORMAL.

6.4.2 Thread creation

```
#include <pthread.h>
int pthread_create(pthread_t          *pthread,
```



```

        const pthread_attr_t  *pattr,
void      (*start_routine)(void *),
void      *arg ;

```

Prototype analysis of Function `pthread_create`:

- For success of the function, return 0. For failure, return the error number;
- Output parameter ***pthread*** is the return thread handle;
- Parameter ***pattr*** is the thread attribute object pointer;
- Parameter ***start_routine*** is the thread function;
- Parameter ***arg*** is the entry function parameter.

Calling the `pthread_create` function can create a POSIX thread, and the thread attribute object ***pattr*** can be created or dynamically set via the `pthread_attr_*` series of functions. If the ***pattr*** is NULL, the system will set a default thread attribute object. When the thread function appointed by the ***start_routine*** returns, the new thread ends. It might also be noted that the thread function ***start_routine*** has only one pointer parameter ***arg***, which means that multiple parameters shall be packaged as a structure for transfer.

The POSIX thread handle is defined as the `pthread_t` type variable. POSIXPOSIXPOSIXThe thread creator gets the created thread handle when the thread is created, the thread can get its own thread handle via calling the `pthread_self` function, and POSIX also defines the `pthread_equal` function to compare two threads for equality.

```

#include <pthread.h>
pthread_t  pthread_self(void);
int  pthread_equal(pthread_t  thread1, pthread_t  thread2);

```

Prototype analysis of Function `pthread_self`:

- For success of the function, return the current thread handle. For failure, return 0.

Prototype analysis of Function `pthread_equal`:

- The function returns comparative results;
- Parameter ***thread1*** is the thread handle;
- Parameter ***thread2*** is the thread handle;

The following program shows how to create the POSIX thread. The following program calls the function `pthread_join` to thread join, which will cause the main thread to wait for the child thread until exit (the specific usage of the `pthread_join` function is introduced in the next section).

Program List 6.4 POSIX thread creation

```

#include <stdio.h>

```

```
#include <pthread.h>
#include <time.h>

void *routine (void *arg)
{
    fprintf(stdout, "pthread running...\n");
    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t      tid;
    pthread_attr_t attr;
    int            ret;

    ret = pthread_attr_init(&attr);
    if (ret != 0) {
        fprintf(stderr, "pthread attr init failed.\n");
        return (-1);
    }

    ret = pthread_create(&tid, &attr, routine, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }
    pthread_join(tid, NULL);
    pthread_attr_destroy(&attr);
    return (0);
}
```

The above program calls the function `pthread_attr_init` to initialize the thread attribute object, and correspondingly calls the function `pthread_attr_destroy` to deinitialize the attribute object. Actually, we are accustomed to setting the second parameter of the `pthread_create` function as `NULL` during programming, so as to tell the operating system to set the thread attribute object as the default value.

Run the program under the SylixOS Shell, and the results are as follows:

```
# ./pthread_test
pthread running...
```

6.4.3 Thread exit

```
#include <pthread.h>
void pthread_exit(void *status);
```

Prototype analysis of Function pthread_exit:

- The function has no return value;
- Parameter **status** is the thread exit status code.

Thread end, i.e., the thread explicitly or implicitly calls the pthread_exit function. The **status** usually represents an integer, and can also point to a more complex data structure. The thread end code will be got by another thread which has joined with the thread.

The single thread can exit in 3 ways:

- The thread can simply return from the thread entry function, and the return value is the thread exit code;
- Threads can be canceled by other threads in the same process (see 6.4.4 Thread cancel);
- The thread explicitly calls the pthread_exit function.

The thread can wait for another thread to exit synchronously and get its exit code. It might also be noted that the synchronously waiting target thread must be at the join state, as shown in Figure 6.3.

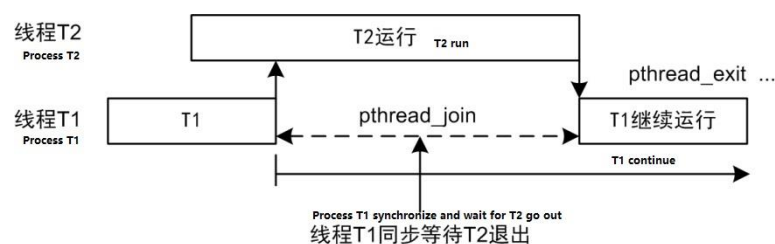


Figure 6.3 Thread synchronization wait

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **ppstatus);
```

Prototype analysis of Function pthread_join:

- For success of the function, return 0. For failure, return the error number;
- Parameter **thread** is the thread handle requiring join;
- Output parameter **ppstatus** is the thread exit status.

Calling the pthread_join function can merge the appointed thread. Calling thread will always block wait until it returns. After the thread returns, the pthread_join function will get the exit code of the thread via the **ppstatus** parameter.

One can set the *ppstatus* as NULL if not interested in the return value of the thread. In this case, the thread calling the `pthread_join` function can wait for the appointed thread to terminate, but it does not get the termination status of the thread.

The following program shows how to get the exit code of the thread via `pthread_join`.

Program List 6.5 Get the exit code of the thread

```
#include <stdio.h>
#include <pthread.h>

void *routine (void *arg)
{
    fprintf(stdout, "thread 1 return.\n");
    return ((void *)1);
}

void *routine1 (void *arg)
{
    fprintf(stdout, "thread 2 exit.\n");
    pthread_exit((void *)2);
}

int main (int argc, char *argv[])
{
    pthread_t    tid, tid1;
    int         ret;
    void        *retval;

    ret = pthread_create(&tid, NULL, routine, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_join(tid, &retval);
    if (ret != 0) {
        fprintf(stderr, "pthread join thread 1 failed.\n");
        return (-1);
    }
    fprintf(stdout, "thread 1 return code: %ld\n", (long)retval);

    ret = pthread_create(&tid1, NULL, routine1, NULL);
    if (ret != 0) {
```

```
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_join(tid1, &retval);
    if (ret != 0) {
        fprintf(stderr, "pthread join thread 2 failed.\n");
        perror("pthread_join");
        return (-1);
    }

    fprintf(stdout, "thread 2 exit code: %ld\n", (long)retval);

    return (0);
}
```

The above program uses two methods for thread exit: return thread and call the `pthread_exit` function to exit.

Run the program under the SylixOS Shell, and the results are as follows:

```
# ./join_test
thread 1 return.
thread 1 return code: 1
thread 2 exit.
thread 2 exit code: 2
```

It can be seen from execution results that when a thread exits by calling the `pthread_exit` function or simply returns from the thread function, other threads in the process can get the exit code of the thread by calling the `pthread_join` function.

As mentioned earlier, the untyped pointer parameters of the `pthread_create` function and the `pthread_exit` function can transfer a more complex structure. However, it shall be noted that the memory used by the structure must remain valid after the caller completes the call. For example, if the structure is allocated on the stack calling the thread, the memory contents of other threads may have changed when using the structure. For another example, the thread allocates a structure on its own stack and then transfers the pointer pointing to the structure to the `pthread_exit` function. When the thread calling the `pthread_join` function tries to use the structure, the stack may have been revoked, and the memory has been used for other purposes.

6.4.4 Thread cancel

Canceling a thread shall guarantee that the thread can release any locked and allocated memory it holds, so as to maintain consistency in the whole the system. It has

cert

A simple thread cancel: the cancel thread calls a cancel thread function, and the canceled thread dies. In this case, the resources held by the canceled thread are not released. The cancel thread shall guarantee that the canceled one is at the safe cancellation state. In a system requiring high reliability, such guarantee is very difficult or cannot be realized. This cancel is called as unrestricted asynchronous cancel.

Asynchronous cancel security is related with the asynchronous cancel, i.e., a piece of code can be canceled at any point during execution without causing inconsistency. The function satisfying the condition is called as the asynchronous cancel security function. Apparently, the asynchronous cancel security function does not involve use of mutex and so on. The POSIX standard requires that these functions are asynchronous cancel security functions: `pthread_cancel`, `pthread_setcancelstate` and `pthread_setcanceltype`.

The POSIX standard defines a more secure thread cancel mechanism. A thread can sent the cancel request to other threads of the process in a reliable and controlled manner, and the target thread can suspend the request and enable the actual cancel action to take place later, called as the **delay cancel**. The target thread can also define the thread clearing function automatically called by the system after canceled. A concept related with the delay cancel is the cancel point.

POSIX cancels the point function and a series of cancel control functions to realize the delay cancel via defining a cancelable state for a thread.

The thread can call the `pthread_cancel` function to request cancellation of a thread.

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Prototype analysis of Function `pthread_cancel`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***thread*** is the thread handle.

The `pthread_cancel` function is asynchronous with the cancel action of the target thread. According to different settings of the target thread, the cancel request may be ignored, executed immediately or delayed. In order to clarify these actions, here we need to understand the cancel state, cancel type and cancel point concept of the thread.

1. Cancellation status

The thread cancel state determines whether the appointed thread can be canceled. The cancel state is divided into allow cancel and prohibit cancel, as shown in Table 6.4. Setting a thread as prohibit cancel means that the thread can only return from itself or call the `pthread_exit` function to exit.

```
#include <pthread.h>
```

```
int pthread_setcancelstate(int newstate, int *poldstate);
```

Prototype analysis of Function pthread_setcancelstate:

- For success of the function, return 0. For failure, return the error number;
- The parameter **newstate** is the new state, as shown in Table 6.4;
- The output parameter **poldstate** returns to the previous state.

Table 6.4 Cancel state

Cancellation status	Note
LW_THREAD_CANCEL_ENABLE	Allow cancel
LW_THREAD_CANCEL_DISABLE	Prohibit cancel

Calling the pthread_setcancelstate function appoints **newstate** parameter value as LW_THREAD_CANCEL_ENABLE to allow cancel, and LW_THREAD_CANCEL_DISABLE to prohibit cancel. If the parameter **poldstate** is not NULL, it returns the previous cancel state.

2. Cancellation type

The cancel type is the thread cancel mode, which is divided into asynchronous cancel and delay cancel, as shown in Table 6.5.

```
#include <pthread.h>
int pthread_setcanceltype(int newtype, int *poldtype);
```

Prototype analysis of Function pthread_setcancelstate:

- For success of the function, return 0. For failure, return non-0 value;
- Parameter **newtype** is the new type, as shown in Table 6.5;
- Output parameter **poldtype** returns the previous type.

Table 6.5 Cancel type

Cancellation type	Note
LW_THREAD_CANCEL_ASYNCHRONOUS	Asynchronous cancel
LW_THREAD_CANCEL_DEFERRED	Delay cancel

Calling the pthread_setcanceltype function can set the cancellation type. When the **poldtype** is non-NULL, it returns the previous cancel type.

When the cancel state is set to prohibit, the cancel request for the thread will be

ignored. When the cancel state is set to allow, if the cancel request is received, the system action is determined by the selected cancel type.

- Asynchronous cancel, the cancel request is executed immediately;
- Delay cancel, the cancellation request is suspended, and executed until running to the next cancel point.

3. Cancel point

When the delay cancel mechanism is canceled, a thread defines the cancel point where it can be canceled. When the cancel request is received, the canceled thread exits when it reaches the cancel point or when a cancel point call is blocked. By the delay cancel, the program does not require prohibit/ allow cancel operation when entering the critical section.

It can be canceled at the cancel point during the delay cancel, and the restriction may enable the cancel request to be suspended for any length of time. Therefore, if a certain call may enable the thread to be blocked or enter a certain process for a long term, POSIX requests that these calls belong to a cancel point, or call these calls as cancel point calls, so as to prevent the cancel request from falling into long-term wait. Functions owning the cancel point in SylixOS are as shown in Table 6.6.

Table 6.6 Functions owning the cancel point

Function name	Function name	Function name
Lw_Thread_Join	send	open
Lw_Thread_Start	sendmsg	close
sleep	aio_suspend	read
Lw_Thread_Cond_Wait	mq_send	pread
system	mq_timedsend	write
wait	mq_reltimedsend_np	pwrite
waitid	mq_receive	readv
waitpid	mq_timedreceive	writew
wait3	mq_reltimedreceive_np	lockf
wait4	pthread_barrier_wait	fsync
reclaimchild	sem_wait	fdatasync
accept4	sem_timedwait	pselect
connect	sem_reltimedwait_np	select
recv	tcdrain	pause
recvfrom	fcntl	sigsuspend
recvmsg	creat	sigwait
sigwaitinfo	sigtimedwait	msgrcv
msgsnd	pthread_join	pthread_testcancel

The following program is an example of thread delay cancel. The program creates the thread thread0, and sets the cancel type as the delay cancel in the thread. The sleep function is a function which owns a cancel point. Therefore, the program will check whether the thread has the cancel request when running to the sleep function. When the cancel request is checked, the thread will be canceled at the next cancel point.

Program List 6.6 Thread delay cancel

```
#include <pthread.h>
#include <stdio.h>

void *thread0 (void *arg)
{
    int oldstate;
    int oldtype;

    pthread_setcancelstate(LW_THREAD_CANCEL_ENABLE, &oldstate);
    pthread_setcanceltype(LW_THREAD_CANCEL_DEFERRED, &oldtype);

    while (1) {
        fprintf(stdout, "thread0 running...\n");
        sleep(1);
    }

    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    ret = pthread_create(&tid, NULL, thread0, NULL);
    if (ret != 0) {
        return (-1);
    }
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, NULL);
    fprintf(stdout, "thread0 cancel.\n");
    return (0);
}
```

Run the program under the SylixOS Shell, and the cancel situation of the thread can

be seen from the results.

```
# ./cancel_test
thread0 running...
thread0 running...
thread0 running...
thread0 cancel.
```

The functions shown in Table 6.6 are realized via directly or indirectly calling the `pthread_testcancel` function. Therefore, the target thread can also call the `pthread_testcancel` function to check the cancel request. The difference between `pthread_testcancel` and other cancellation point calls is that the function does nothing except for creating a cancel point, i.e., specifically responds to cancel request.

```
#include <pthread.h>
void pthread_testcancel(void);
```

If the `pthread_testcancel` function does not check the cancel request, it is directly returned to the caller. When the cancel request is checked, the thread will be deleted and no longer returned to the caller.

The thread can arrange the function where it needs to call when it exits, such function is called as the thread cleanup processing program, and a thread can create multiple cleanup processing programs. The processing programs are recorded in the stack, i.e., the sequence of execution is opposite with that of registration, as shown in Figure 6.4.

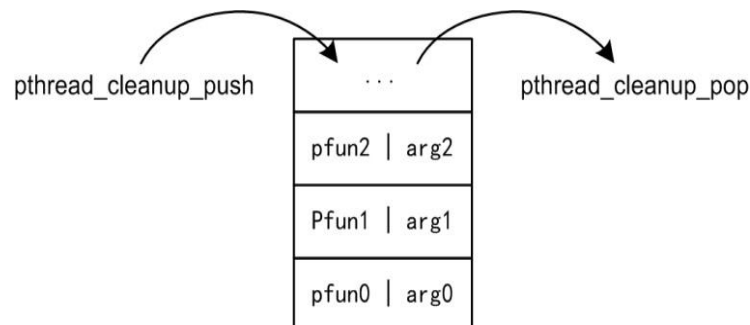


Figure 6.4 Cleanup function stack

```
#include <pthread.h>
void pthread_cleanup_pop(int iNeedRun);
void pthread_cleanup_push(void (*pfunc)(void *), void *arg);
```

Prototype analysis of Function `pthread_cleanup_pop`:

- Parameter *iNeedRun* indicates whether to execute.

Prototype analysis of the function `pthread_cleanup_push`:

- Parameter ***pfunc*** is the cleanup function to be executed;
- Parameter ***arg*** is the cleanup function parameter.

If ***iNeedRun*** is 0, the cleanup function will not be called, the `pthread_cleanup_pop` function will delete the cleanup processing program created via the last `pthread_cleanup_push` call.

These functions must be used in the paired form in the action scope same with the thread.

The following program shows how to use these two functions.

Program List 6.7 Thread cleanup

```
#include <stdio.h>
#include <pthread.h>

void cleanup (void *arg)
{
    fprintf(stdout, "cleanup: %s.\n", (char *)arg);
}

void *routine (void *arg)
{
    fprintf(stdout, "thread 1 running...\n");

    pthread_cleanup_push(cleanup, "thread1 first");
    pthread_cleanup_push(cleanup, "thread1 second");

    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);

    return ((void *)1);
}

void *routine1 (void *arg)
{
    fprintf(stdout, "thread 2 running...\n");

    pthread_cleanup_push(cleanup, "thread2 first");
    pthread_cleanup_push(cleanup, "thread2 second");

    if (arg) {
        return ((void *)2);
    }
}
```

```
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
return ((void *)2);
}

int main (int argc, char *argv[])
{
    pthread_t      tid, tid1;
    int           ret;
    void          *retval;

    ret = pthread_create(&tid, NULL, routine, (void *)1);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_join(tid, &retval);
    if (ret != 0) {
        fprintf(stderr, "pthread join thread 1 failed.\n");
        return (-1);
    }
    fprintf(stdout, "thread 1 return code: %ld\n", (long)retval);

    ret = pthread_create(&tid1, NULL, routine1, (void *)2);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_join(tid1, &retval);
    if (ret != 0) {
        fprintf(stderr, "pthread join thread 2 failed.\n");
        perror("pthread_join");
        return (-1);
    }
    fprintf(stdout, "thread 2 exit code: %ld\n", (long)retval);
    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./push_test
```

```

thread 1 running...
thread 1 return code: 1
thread 2 running...
cleanup: thread2 second.
cleanup: thread2 first.
thread 2 exit code: 2

```

From running results of the program, it can be seen that both threads run and exit normally, while the thread 1 does not call the cleanup function. As we mentioned before, the `pthread_cleanup_pop` function parameter ***iNeedRun*** is 0, and it will not call the clean function. However, the `pthread_cleanup_pop` function parameter ***iNeedRun*** of the thread2 is also 0, but it calls the cleanup function. This shows that the clean function is called after the thread function exits, and the call sequence is opposite with that during installation.

6.5 POSIX thread key value

The POSIX thread key value is also called as the thread private data. Intrinsicly, the thread key values has same names and different values for multiple threads, as shown in Figure 6.5. In Section 6.3.4 we discussed the role of thread private data in multi-threaded security. The following describes API of the POSIX thread key and how to use it.

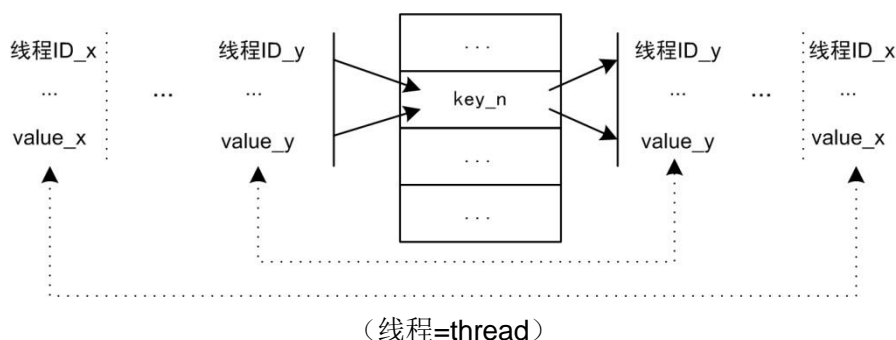


Figure 6.5 Thread key value

Note:global chain tables manages all keys in SylixOS. Therefore, keys in SylixOS are visual in the whole system.

```

#include <pthread.h>
int pthread_key_create(pthread_key_t *pkey, void (*destructor)(void *));
int pthread_key_delete(pthread_key_t key);

```

Prototype analysis of Function `pthread_key_create`:

- For success of the function, return 0. For failure, return the error number;
- Output parameter ***pkey*** returns the thread key created;

- Parameter ***destructor*** is the delete function.

Prototype analysis of Function pthread_key_delete:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***key*** is the key to be deleted.

Calling the pthread_key_create function can create a key, which can be used by all threads in the same process. Therefore, the key is usually defined as a global variable. If Parameter ***destructor*** is not NULL, it is automatically called when the key is deleted. If it is not required to release any memory, Parameter ***destructor*** can be set as NULL. Calling the pthread_key_delete function will delete a key. It might also be noted that, the key cannot be deleted in the parameter ***destructor*** function, because the key belongs to the whole system, i.e., all threads is visual. If the key in the ***destructor*** function is deleted, other threads may access it again, which will cause unpredictable errors.

A ***key*** corresponds to the only ***destructor*** function shared by all threads. The ***destructor*** function must be appointed when the ***key*** is created, and cannot be changed. If the ***destructor*** function is appointed, SylixOS will automatically call the cleanup function when the thread exits.

The key returned by the pthread_key_create function is the pthread_key_t type, which may represent different types in different system realization. The key value is just the array index in some systems, while the key value represents an address value in SylixOS. Therefore, we will not get the preconceive results when trying to print a key value.

Calling the pthread_setspecific function can associate the thread private data with the key.(the key is returned via the previous pthread_key_create call) The function of the pthread_getspecific function is opposite with that of the pthread_setspecific function, and the private data (***pvalue***) associated with the ***key*** in the thread is returned.

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *pvalue);
void *pthread_getspecific(pthread_key_t key);
```

Prototype analysis of Function pthread_setspecific:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***key*** is the key;
- Parameter ***pvalue*** is the value to be set.

Prototype analysis of Function pthread_getspecific:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***key*** is the key;

Parameter *pvalue* of the `pthread_setspecific` function is an untyped pointer, which can point to any data type, including complex data structure. When the thread terminates, the pointer will be transferred as a parameter to the *destructor* function (`pthread_key_create` parameter) corresponding to the key.

The following program shows how to use POSIX thread key.

Program List 6.8 Use of POSIX thread key

```
#include <pthread.h>
#include <stdio.h>

static char          *str = "this is pthread key.";
static pthread_key_t  key;

void *thread (void *arg)
{
    void          *keyval;

    pthread_setspecific(key, str);
    sleep(2);

    keyval = pthread_getspecific(key);
    fprintf(stdout, "keyval child thread is: %s\n", (char *)keyval);

    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t      tid;
    int            ret;
    void          *keyval;

    ret = pthread_key_create(&key, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread key create failed.\n");
        return (-1);
    }

    ret = pthread_create(&tid, NULL, thread, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }
}
```

```
    }  
  
    keyval = pthread_getspecific(key);  
    fprintf(stdout, "keyval main thread is: %s\n", (char *)keyval);  
  
    pthread_join(tid, NULL);  
    pthread_key_delete(key);  
  
    return (0);  
}
```

Run the program in SylixOS Shell, and it can be seen from program running results that two threads in the program share a key, but the key value set in the child thread is not visible in the main thread. As shown in Figure 6.5, although the thread ID_x and the thread ID_y share the key key_n, their values value_x and value_y are different.

```
# ./key_test  
keyval main thread is: (null).  
keyval child thread is: this is pthread key.
```

6.6 SylixOS thread scheduling

A significant difference between real-time system and time-sharing system is reflected in the scheduling strategy. Real-time system scheduling concerns delay in response to real-time events. However, the traditional time-sharing system scheduling considers multiple objectives: fairness, efficiency, utilization and throughput. Therefore, the real-time system usually adopts priority scheduling, that is to say, the operating system always selects the highest priority from the ready task queue for operation.

6.6.1 Priority scheduling

A thread will monopolize processor running once it gets the processor, and the system will schedule other threads unless it decides to abandon the processor for a certain reason, the scheduling mode is called as "non-preemptive scheduling", i.e., the system reselects the thread according to priority scheduling after the thread actively makes way for the processor. The scheduling mode will cause that the ready thread with high priority cannot be timely responded. Therefore, the real-time response speed will be reduced, When a thread is running, the operating system can deprive the processor allocated to it and allocate it to other threads according to a certain principle, the scheduling mode is called as "preemptive scheduling", and the scheduler principle includes: priority principle, time slice principle and so on. The preemptive priority scheduling principle is adopted between different priorities in SylixOS.

It can be seen from the above that the "preemptive scheduling" mode shall be adopted when focusing on real-time response. As long as the higher-priority thread is ready, the system immediately interrupts the current thread to schedule the high-priority thread, so as to guarantee that the high-priority thread can get the processor at any time, which is basic requirements for the real-time system.

The SylixOS kernel supports 256 priorities^①: 0 to 255. Priority 0 is the highest and Priority 255 is the lowest. The priority is determined when the thread is created, and dynamic modification is allowed during program execution. For the kernel, the priority is determined when a thread is selected from the ready queue, i.e., the kernel will not dynamically calculate the priority of each thread. Therefore, the scheduling strategy belongs to the static scheduling strategy. Compared with scheduling of the dynamic scheduling strategy, the thread priority shall be dynamically determined and scheduled according to a certain target. The efficiency of the static scheduling strategy is higher than that of the dynamic scheduling strategy.

```
#include <SylixOS.h>
ULONG Lw_Thread_SetPriority(LW_OBJECT_HANDLE uId, UINT8 ucPriority);
ULONG Lw_Thread_GetPriority(LW_OBJECT_HANDLE uId, UINT8 *pucPriority);
```

Prototype analysis of Function `Lw_Thread_SetPriority`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *uId* is the thread handle to be set;
- Parameter *ucPriority* is the new priority value.

Prototype analysis of Function `Lw_Thread_GetPriority`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *uId* is the thread handle;
- Output parameter *pucPriority* returns the thread priority.

Calling the `Lw_Thread_SetPriority` function can set the priority of a appointed thread, and calling the `Lw_Thread_GetPriority` function can get the priority of the appointed thread.

Note: we can use the Shell command *sprio* to dynamically modify the priority of the running thread.

[Command format]

```
sprio [priority thread_id]
```

[Common option]

None

[Instructions for parameters]

```

priority : Priority value
thread_id : Target thread ID

```

6.6.2 RR (Round-Robin) scheduling

The above-mentioned priority-based scheduling strategy has such problems: if it is not preempted by the higher-priority thread, or makes way for the processor due to blockage or other reasons, the thread will keep on running. In the circumstance, the thread with the same priority will not run. RR scheduling is based on such philosophy: on the premise of higher-priority thread scheduling still runs preferentially, fairness in a certain significance is pursued during scheduling between threads with the same priority.

RR scheduling divides thread running into time slices. After the thread runs for a time slice, the kernel calls it out of the processor and places it at the queue tail of the ready thread with same priority, and reselects thread running conforming to conditions. The effect of RR scheduling is to "make way for" the processor to the next thread after each thread runs a time slice, like rotation, so it is also called rotation scheduling. It can be seen that RP scheduling does not change characteristics of two real-time scheduling of "priority-based" and "preemptible".

If the RR scheduling strategy is adopted, one issue worth considering is determination of the time slice size. The small time slice facilitates the thread with same priority to fairly share the processor, but increases the scheduling overhead. If the time slice is increased, the scheduling overhead decreases, but the scheduling effect will tend to priority scheduling. The reasonable time slice will be compromised between fairness and efficiency.

Calling the following function can dynamically change and get the time slice of the thread.

```

#include <SylixOS.h>
ULONG Lw_Thread_SetSlice(LW_OBJECT_HANDLE ulId, UINT16 usSlice);
ULONG Lw_Thread_GetSlice(LW_OBJECT_HANDLE ulId, UINT16 *pusSliceTemp);
ULONG Lw_Thread_GetSliceEx(LW_OBJECT_HANDLE ulId, UINT16 *pusSliceTemp,
                           UINT16 *pusCounter);

```

Prototype analysis of Function `Lw_Thread_SetSlice`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *ulId* is the thread handle;
- Parameter *usSlice* is the new time slice of the thread.

Prototype analysis of Function `Lw_Thread_GetSlice`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***uId*** is the thread handle;
- Output parameter ***pusSliceTemp*** returns the time slice.

Prototype analysis of Function Lw_Thread_GetSliceEx:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***uId*** is the thread handle;
- Output parameter ***pusSliceTemp*** returns the time slice;
- Output parameter ***pusCounter*** returns the remaining time slice.

Calling the Lw_Thread_SetSlice function can set the running slice of the thread, calling the Lw_Thread_GetSlice function can get the time slice of the thread and calling the Lw_Thread_GetSliceEx function will also get the remaining time slice of the thread.

Calling the following functions can modify the thread scheduling strategy in SylixOS.

```
#include <SylixOS.h>
ULONG Lw_Thread_SetSchedParam(LW_OBJECT_HANDLE    uId,
                              UINT8              ucPolicy,
                              UINT8              ucActivatedMode);
ULONG Lw_Thread_GetSchedParam(LW_OBJECT_HANDLE    uId,
                              UINT8              *pucPolicy,
                              UINT8              *pucActivatedMode);
```

Prototype analysis of Function Lw_Thread_SetSchedParam:

- For success of the function, return 0. For failure, return non-0 value;
- Parameter ***uId*** is the thread handle;
- Parameter ***ucPolicy*** is the scheduling strategy;
- Parameter ***ucActivatedMode*** is the response model, as shown in Table 6.7.

Table 6.7 Thread response mode

Response mode	Note
LW_OPTION_RESPOND_IMMEDIA	High-speed response thread (only for test)
LW_OPTION_RESPOND_STANDARD	Common response thread
LW_OPTION_RESPOND_AUTO	Automatic

Prototype analysis of Function Lw_Thread_GetSchedParam:

- For success of the function, return 0. For failure, return non-0 value;

- Output parameter ***pucPolicy*** returns the scheduling strategy;
- Output parameter ***pucActivatedMode*** returns the thread response mode, as shown in Table 6.7.

Calling the `Lw_Thread_SetSchedParam` function can set the thread scheduling strategy, and calling the `Lw_Thread_GetSchedParam` function can get the thread scheduling strategy.

6.7 POSIX thread scheduling

The scheduling behavior is affected by two factors: scheduling strategy and task priority. Each task has a priority. The system maintains a list of ready tasks for each allowed priority. The list has a certain sequence, list head task and list tail task. If there is a new task ready, it will be put at the suitable place in the list (it is usually required to select this according to the scheduling strategy).

For the real-time system, the response speed is the most important. Therefore, as real-time extension of the basic definition of POSIX, the scheduling strategy defined by POSIX 1003.1b is based on the priority. Other evaluation indicators such as fairness and throughput capacity are secondary.

POSIX standard specifies that the high-priority thread has the large priority digital. The priority in SylixOS is opposite. Therefore, SylixOS performs priority conversion, and definition is as follows:

```
#include <posix/include/px_sched_param.h>
#define PX_PRIORITY_CONVERT(prio) (LW_PRIO_LOWEST - (prio))
```

The macro does not concern application development, but understanding the macro can clarify relationship between POSIX priority and SylixOS priority.

POSIX defines the structure `sched_param` to represent the scheduling-related parameter, and implementation in SylixOS is shown below:

```
struct sched_param {
    int          sched_priority;          /* POSIX scheduling priority */
                                           /* SCHED_SPORADIC parameter */
    int          sched_ss_low_priority; /* Low scheduling priority for */
                                           /* sporadic server. */
    struct timespec sched_ss_repl_period; /* Replenishment period for */
                                           /* sporadic server. */
    struct timespec sched_ss_init_budget; /* Initial budget for sporadic */
                                           /* server. */
    int          sched_ss_max_repl;     /* Max pending replenishments */
                                           /* for sporadic server. */
}
```

```
};
```

Note: SylixOS only supports the priority setting of the structure at present, while others are reserved items.

POSIX defines the following functions to dynamically select thread scheduling strategy (SCHED_FIFO and SCHED_RR).

```
#include <sched.h>
int sched_setscheduler(pid_t          pid①,
                      int            iPolicy,
                      const struct sched_param pschedparam);
int sched_getscheduler(pid_t pid);
```

Prototype analysis of Function sched_setscheduler:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *pid* is the process ID;
- Parameter *iPolicy* is the scheduling strategy.
- Output parameter *pschedparam* is the scheduler parameter.

Prototype analysis of Function sched_getscheduler:

- For success of the function, return the scheduler strategy (SCHED_FIFO and SCHED_RR). For failure, return -1 and set the error number;
- Parameter *pid* is the process ID;

In Section 6.4.1, when the scheduling strategy is introduced, we learned that the thread scheduling policy can be set via calling the pthread_attr_setschedpolicy function, the method is set before the thread is created, i.e., a static change method. However, a method to dynamically change the thread priority is provided by calling the sched_setscheduler function. Calling the sched_getscheduler function can get the scheduling policy of the appointed thread.

It might also be noted that the sched_setscheduler function will set the thread priority while setting the scheduling strategy, and later we will introduce what range values the POSIX thread priority shall satisfy.

Calling the following two functions can get the maximum and minimum values of the POSIX thread priority, and the priority set by application shall be within the range of these two values (excluded).

① Process operation usually refers to operation of the main thread of the process in SylixOS. Therefore, the process scheduling strategy here usually refers to the thread scheduling strategy.

```
#include <sched.h>
int sched_get_priority_max(int iPolicy);
int sched_get_priority_min(int iPolicy);
```

Prototype analysis of Function `sched_get_priority_max`:

- The function returns the maximum priority value of POSIX;
- Parameter ***iPolicy*** is the scheduling strategy.

Prototype analysis of Function `sched_get_priority_min`:

- The function returns the minimum priority value of POSIX;
- Parameter ***iPolicy*** is the scheduling strategy.

POSIX allows different priority ranges to be defined when different scheduling strategies are adopted. For implementation of current SylixOS, all scheduling strategies have the same priority range.

Calling the `sched_setscheduler` function sets the process priority while setting the scheduling priority. Actually, calling the `sched_setparam` function can set the process priority, and calling the `sched_getparam` function can get the priority of the appointed process.

```
#include <sched.h>
int sched_setparam(pid_t pid, const struct sched_param *pschedparam);
int sched_getparam(pid_t pid, struct sched_param *pschedparam);
```

Prototype analysis of Function `sched_setparam`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pid*** is the process ID;
- Output parameter ***pschedparam*** is the scheduling parameter.

Prototype analysis of Function `sched_getparam`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pid*** is the process ID;
- The output parameter ***pschedparam*** returns the scheduling parameter.

If `pid` equals 0, the current thread priority is set. It might also be noted that if the set priority is the same with the current priority of the appointed thread, nothing is done and 0 is returned.

```
#include <sched.h>
```

```
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
int sched_yield(void);
```

Prototype analysis of Function `sched_rr_get_interval`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *pid* is the process ID;
- Output parameter *interval* returns the time remaining for the process or thread.

Calling the `sched_rr_get_interval` function will return the time slice size (timespec type time value) of the appointed thread. The function is valid only when the scheduling strategy is SCHED_RR. Otherwise, it returns -1 and sets `errno` as EINVAL. Calling the `sched_yield` function will actively abandon the processor once.

The following program shows how to use the POSIX thread scheduling function.

Program List 6.9 Use of the POSIX thread scheduling function

```
#include <sched.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    struct sched_param param;
    struct sched_param newparam;
    struct timespec time;
    int ret;

    fprintf(stdout, "Max prio: %d, Min: %d\n",
            sched_get_priority_max(SCHED_RR),
            sched_get_priority_min(SCHED_RR));

    ret = sched_getparam(0, &param);
    if (ret != 0) {
        perror("sched_getparam");
        return (-1);
    }

    fprintf(stdout, "old prio: %d\n", param.sched_priority);
    param.sched_priority = 40;
    ret = sched_setscheduler(0, SCHED_RR, &param);
    if (ret != 0) {
        return (-1);
    }
}
```

```

ret = sched_getparam(0, &newparam);
if (ret != 0) {
    perror("sched_getparam");
    return (-1);
}

fprintf(stdout, "new prio: %d\n", newparam.sched_priority);
sched_rr_get_interval(0, &time);
fprintf(stdout, "time slice %lld(s):%ld(ns)\n", time.tv_sec, time.tv_nsec);

return (0);
}

```

Run the program in the SylixOS Shell: Running results show that the range of available priority of the POSIX thread in SylixOS is from 1 to 254, indicating that priority 0 and priority 255 are not recommended by the Sylix OS kernel.

```

# ./sched_test
Max prio: 254, Min prio: 1
old prio: 55
new prio: 40
time slice 0(s):70000000(ns)

```

6.8 SylixOS RMS scheduling

A very good method to solve multi-task scheduling conflicts for the periodic task is rate monotonic scheduling (Rate Monotonic Scheduling RMS), and RMS appoints the priority based on the task cycle.

In RMS, the task with the shortest cycle has the highest priority, the task with the next shortest cycle has the next highest priority, and so forth. When multiple tasks can be executed at the same time, the task with the shortest cycle is executed preferentially. If the function deems the task priority as the rate, then this is a monotonically increasing function.

```

#include <SylixOS.h>
LW_HANDLE Lw_Rms_Create(CPCHAR      pcName,
                       ULONG        ulOption,
                       LW_OBJECT_ID *pulId);
ULONG Lw_Rms_Delete(LW_HANDLE      *pulId);
ULONG Lw_Rms_DeleteEx(LW_HANDLE    *pulId,
                      BOOL         bForce);
ULONG Lw_Rms_Cancel(LW_HANDLE      ulId);

```


Prototype analysis of Function Lw_Rms_Create:

- For function success, return RMS handle. For failure, return LW_HANDLE_INVALID, and set the error number;
- Parameter **pcName** is RMS name;
- Parameter **ulOption** is RMS option;
- Output parameter **pulld** returns RMS handle.

Prototype analysis of Function Lw_Rms_Delete:

- For success of the function, return 0. For failure, return the error number;
- Parameter **pulld** is RMS handle pointer.

Prototype analysis on Function Lw_Rms_DeleteEx:

- For success of the function, return 0. For failure, return the error number;
- Parameter **pulld** is RMS handle pointer;
- Parameter **bForce** is delete type.

Prototype analysis of Function Lw_Rms_Cancel:

- For success of the function, return 0. For failure, return the error number;
- Parameter **ulld** is RMS handle.

Calling Lw_Rms_Create function can create an RMS scheduler; calling Lw_Rms_Delete function can delete an RMS scheduler. It might also be noted that if RMS scheduler is at the task blocking state, the RMS object will not be deleted, and errno will be set as ERROR_RMS_STATUS; Calling Lw_Rms_DeleteEx function can delete an RMS scheduler. Different from Lw_Rms_Delete function, if **bForce** is true, the RMS scheduler will be deleted regardless of the state. Otherwise, the behavior is the same with that of the Lw_Rms_Delete function; calling the Lw_Rms_Cancel function will stop the specified RMS scheduler, but will not delete RMS object.

```
#include <SylixOS.h>
ULONG Lw_Rms_Period(LW_HANDLE          ulld,
                   ULONG              ulPeriod);
ULONG Lw_Rms_ExecTimeGet(LW_HANDLE    *pulld,
                        ULONG          *pulExecTime);
```

Prototype analysis of Function Lw_Rms_Period:

- For success of the function, return 0. For failure, return the error number;
- Parameter **ulld** is the RMS handle;

- Parameter ***ulPeriod*** is the program execution cycle.

Prototype analysis of Function `Lw_Rms_ExecTimeGet`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ulld*** is the RMS handle;
- Output parameter ***pulExecTime*** returns the running time.

After the `Lw_Rms_Period` function is called, RMS scheduler will start work according to the period specified by Parameter ***ulPeriod***; calling the `Lw_Rms_ExecTimeGet` function will obtain the time from the start of `Lw_Rms_Period` function call to the current execution (unit: Tick).

```
#include <SylixOS.h>
ULONG Lw_Rms_Status(LW_HANDLE      ulld,
                   UINT8          *pucStatus,
                   ULONG          *pulTimeLeft,
                   LW_HANDLE      *pulOwnerId);
ULONG Lw_Rms_GetName(LW_HANDLE ulld, PCHAR pcName);
```

Prototype analysis of Function `Lw_Rms_Status`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ulld*** is the RMS handle;
- Output parameter ***pucStatus*** returns RMS state;
- Output parameter ***pulTimeLeft*** returns the remaining waiting time;
- Output parameter ***pulOwnerId*** returns the owner ID.

Prototype analysis of Function `Lw_Rms_GetName`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ulld*** is the RMS handle;
- Output parameter ***pcName*** returns RMS name.

Calling `Lw_Rms_Status` will return the status of the RMS scheduler. If Parameter ***pucStatus*** is not NULL, the status as shown in Table 6.8 will be returned . If Parameter ***pulTimeLeft*** is not NULL, the remaining time of the schedule will be returned. If Parameter ***pulOwnerId*** is not NULL, the thread handle of the RMS scheduler owner will be returned; calling the `Lw_Rms_GetName` function will return the name of the RMS scheduler.

Table 6.8 RMS scheduler status

State name	Note
LW_RMS_INACTIVE	RMS scheduler just created
LW_RMS_ACTIVE	Initialize the cycle, and measure the execution time
LW_RMS_EXPIRED	With task blocking

As an extension to POSIX, SylixOS provides the following set of functions to implement the POSIX RMS scheduler. Compared with the previous RMS implementation, the following functions are easier to use, and have higher time accuracy (nanosecond-level).

```
#include <sched_rms.h>
int sched_rms_init(sched_rms_t *prms, pthread_t thread);
int sched_rms_destroy(sched_rms_t *prms);
int sched_rms_period(sched_rms_t *prms, const struct timespec *period);
```

Prototype analysis of Function sched_rms_init:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *prms* is the RMS scheduler pointer;
- Parameter *thread* is the call thread handle.

Prototype analysis of Function sched_rms_destroy:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *prms* is the RMS scheduler pointer;

Prototype analysis of Function sched_rms_period:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *prms* is the RMS scheduler pointer;
- Parameter *period* is the program execution cycle.

Calling the sched_rms_init function will initialize the RMS scheduler specified by Parameter *prms*. Unlike the Lw_Rms_Create function, the application creates an RMS scheduler of sched_rms_t type, and then the sched_rms_init function is called to initialize the scheduler for the former. That is to say, the scheduler will be created and destroyed by the application. However, the RMS scheduler created by the latter is managed by the kernel, i.e., the application will not directly manage the scheduler used.

Calling the `sched_rms_destroy` function will destroy the scheduler initialized by the `sched_rms_init` function. The destroyed scheduler cannot be used unless it is reinitialized.

Calling the `sched_rms_period` function enables the RMS scheduler to start working.

The following program shows how to use the RMS scheduler.

Program List 6.10 RMS scheduler

```
#include <sched_rms.h>
#include <pthread.h>

sched_rms_t  rms;

void process_func(void)
{
    int i = 1;

    for (; i >= 0; --i) {
        sleep(1);
    }
}

void *rms_thread (void *arg)
{
    struct timespec  *period = (struct timespec *)arg;

    sched_rms_init(&rms, pthread_self());
    while (1) {
        if (sched_rms_period(&rms, period) != 0) {
            break;
        }
        process_func();
        fprintf(stdout, "rms thread running...\n");
    }

    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t      tid;
    int            ret;
}
```

```
    struct timespec    period;

    period.tv_nsec    = 0;
    period.tv_sec     = 3;

    ret = pthread_create(&tid, NULL, rms_thread, (void *)&period);
    if (ret < 0) {
        fprintf(stderr, "pthread_create error.\n");
        return (-1);
    }

    pthread_join(tid, NULL);
    sched_rms_destroy(&rms);

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./rms_test
rms thread running...
rms thread running...
.....
```

The scheduling cycle set by the program is 3 seconds, and the running time of the running function `process_func` of the thread `rms_thread` is more than 2 seconds and less than 3 seconds. Therefore, the thread can be scheduled normally, and program operation results also prove the point. If we change the `i` value in the `process_func` function to a value larger than or equal to 2, the thread will only be scheduled once. The running time of the thread at the moment is greater than the cycle value of the RMS scheduler, which will cause timeout overflow error of the scheduler (E_OVERFLOW).

6.9 SylixOS coroutine

Coroutine is the executable code sequence smaller than the thread. A thread can have multiple coroutines, which share resources of the thread except the stack, such as the priority, kernel object and so on. All coroutines in the thread share the kernel object of the thread itself. Therefore, the scheduler itself does not know existence of the coroutine, and the coroutine is executed when the thread to which it belongs is scheduled. The internal coroutine of a thread cannot be preempted, and can only run in cycle. Only when the current running coroutine voluntarily abandons the processor, other coroutines in the same thread can get the processor. When the thread is deleted, all coroutines within the thread are deleted at the same time.

The SylixOS kernel supports coroutines instead of using third-party library emulation, which makes coroutine management within SylixOS easier and more efficient.

Calling the `Lw_Coroutine_Create` function will create a coroutine in the current thread. It might also be noted that creation of each thread creation will create a starting coroutine at default. Therefore, the thread always runs from the starting coroutine.

```
#include <SylixOS.h>
PVOID Lw_Coroutine_Create(PCOROUTINE_START_ROUTINE    pCoroutineStartAddr,
                          size_t                      stStackSize,
                          PVOID                       pvArg);
```

Prototype analysis of Function `Lw_Coroutine_Create`:

- For success of the function, return the coroutine control pointer. For failure, return `LW_NULL` and set the error number;
- Parameter *pCoroutineStartAddr* is the coroutine starting address;
- Parameter *stStackSize* is the coroutine stack size^①;
- Parameter *pvArg* is the entry parameter.

Calling the `Lw_Coroutine_Delete` function will delete an appointed coroutine. If the current coroutine system is deleted, the system will directly call the `Lw_Coroutine_Exit` function for exit. The thread will exit when the last coroutine in the thread exits.

```
#include <SylixOS.h>
ULONG Lw_Coroutine_Delete(PVOID pvCrcb);
ULONG Lw_Coroutine_Exit(VOID);
```

Prototype analysis of Function `Lw_Coroutine_Delete`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pvCrcb* is the coroutine control pointer.

Prototype analysis of Function `Lw_Coroutine_Exit`:

- For success of the function, return 0. For failure, return the error number;

As we mentioned earlier, the scheduler does not know existence of the coroutine, and the coroutines run in cycle, which determines that coroutine scheduling must be managed by the user program. SylixOS provides the following functions to change the scheduling sequence of the coroutine.

```
#include <SylixOS.h>
VOID Lw_Coroutine_Yield(VOID);
ULONG Lw_Coroutine_Resume(PVOID pvCrcb);
```

Prototype analysis of Function `Lw_Coroutine_Resume`:

- For success of the function, return 0. For failure, return non-0 value;
- Parameter ***pvCrcb*** is the coroutine control pointer.

Calling the `Lw_Coroutine_Yield` function can actively abandon the processor, and calling the `Lw_Coroutine_Resume` function can recover the appointed coroutine.

Calling the `Lw_Coroutine_StackCheck` function can check the coroutine stack.

```
#include <SylixOS.h>
ULONG Lw_Coroutine_StackCheck(PVOID          pvCrcb,
                               size_t        *pstFreeByteSize,
                               size_t        *pstUsedByteSize,
                               size_t        *pstCrcbByteSize);
```

Prototype analysis of Function `Lw_Coroutine_StackCheck`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pvCrcb*** is the coroutine control pointer;
- Output parameter ***pstFreeByteSize*** returns the idle stack size;
- Output parameter ***pstUsedByteSize*** returns the use stack size;
- Output parameter ***pstCrcbByteSize*** returns the size of the coroutine control block.

Parameters `pstFreeByteSize`, `pstUsedByteSize` and `pstCrcbByteSize` can be NULL. If the corresponding parameter is NULL, stack of the appointed type will not be concerned.

It might also be noted that if service of the coroutine stack is checked, the stack check option must be used for the stack the coroutine parent, as shown in Table 6.3.

The following program shows how to use the SylixOS coroutine.

Program List 6.11 Coroutine use

```
#include <SylixOS.h>
#include <stdio.h>

VOID coroutine0 (PVOID pvArg)
{
    INT i;

    for (i = 0; i < 5; i++) {
        fprintf(stdout, "coroutine0 running...\n");
        Lw_Time_SSleep(1);
    }
}
```

```
VOID coroutinel (PVOID pvArg)
{
    INT i;

    for (i = 0; i < 5; i++) {
        fprintf(stdout, "coroutinel running...\n");
        Lw_Time_SSsleep(1);
    }
}

PVOID tTest (PVOID pvArg)
{
    PVOID pcCrcb0, pcCrcb1;

    pcCrcb0 = Lw_Coroutine_Create(coroutine0, 2 * 1024, LW_NULL);
    if (pcCrcb0 == LW_NULL) {
        return (LW_NULL);
    }

    pcCrcb1 = Lw_Coroutine_Create(coroutinel, 2 * 1024, LW_NULL);
    if (pcCrcb1 == LW_NULL) {
        return (LW_NULL);
    }

    Lw_Coroutine_Yield(); /* Make other coroutines
run */

    while (1) {
        Lw_Time_SSsleep(10);
    }

    return ((PVOID)1);
}

int main (int argc, char *argv[])
{
    LW_HANDLE hId;

    hId = Lw_Thread_Create("t_test", tTest, LW_NULL, LW_NULL);
    if (hId == LW_HANDLE_INVALID) {
        return (PX_ERROR);
    }
}
```



```
Lw_Thread_Join(hId, LW_NULL);  
  
return (ERROR_NONE);  
}
```

Run the program under the SylixOS Shell:

```
# ./coroutine_test  
coroutine0 running...  
coroutine0 running...  
coroutine0 running...  
coroutine0 running...  
coroutine0 running...  
coroutine0 running...  
coroutine1 running...  
coroutine1 running...  
coroutine1 running...  
coroutine1 running...  
coroutine1 running...
```

It can be seen from running results that the coroutine0 gets execution first, and is not interrupted due to delay and blockage at running time. Let's analyze the process below.

In the tTest thread, the program creates the coroutine0 coroutine firstly and then the coroutine1 coroutine. It was introduced above that when the thread is created, a starting coroutine will be created at default. Therefore, the thread starts running from the starting coroutine firstly. Here the program calls the Lw_Coroutine_Yield function to actively abandon the processor, and the coroutine0 get the processor for running at the moment. The coroutines run in cycle. Therefore, the coroutine0 will not actively abandon the processor before running (the program does not actively call the Lw_Coroutine_Yield function. Meanwhile, it is proved that the scheduler does not perceive existence of the coroutine). The coroutine1 will be automatically switched to after running of coroutine0. This also indicates that the coroutine firstly created will get the processor in priority, which conforms to the FIFO principle.

Chapter 7 Inter-thread communication

7.1 Shared resource

The entity, such as a variable, device or memory block accessible by the thread, is called as **resource**.

The resources which can be accessed by multiple threads are called as **shared resources**; while the behavior of simultaneous access to shared resources is called as **shared resource competition**.

If you do not monopolize the shared resource while accessing the shared resource, it may cause resource exception (such as variable value confusion, device error, or the memory block content not being the expected value, etc.), which may lead to the program to run abnormally or even crash.

Now, two threads (Thread A and Thread B) need to perform add one operation to the same Variable V (the initial value is 0).

On the RISC machine, the load/store system structure is generally adopted, that is to say, access to the memory only allows load and store operations; the machine instruction process for auto increment operation of Variable V is as follows:

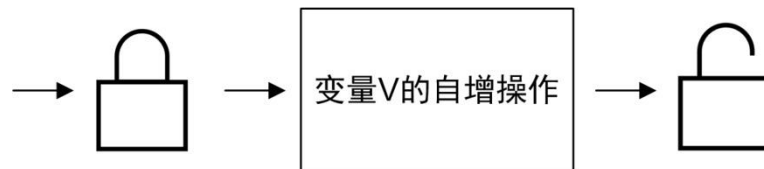
- (1) Load the address of Variable V to the working register 0 of the CPU;
- (2) The load instruction loads the contents of the address stored in working register 0 into working register 1;
- (3) The inc instruction adds 1 to the value of working register 1;
- (4) The store instruction saves the value of working register 1 to the address pointed to by working register 0.

It can be seen from the above that the auto increment operation of Variable V is not completed in one step. If Thread A and Thread B complete the above four steps in sequence, then the final value of Variable V will be 2.

If Thread A completes the previous three steps, then Thread B interrupts the work of Thread A, Thread B rewrites Variable V to 1; Thread A continues to perform the fourth step, and then the final value of Variable V is still 1, which is obviously not what we expect.

In order to solve this problem, we need to perform mutex access to the process. Mutex is an exclusive behavior, which means that only one thread is allowed to access shared resources at the same time. There are several methods to implement mutex: disable interrupt, disable task scheduling, semaphore and so on.

For the above process, we can add a lock (semaphore), which must be held before auto increment operation of Variable V, and released after the operation is completed; assuming that the lock has been occupied by Thread A, if the thread B also wants to apply for the lock, Thread B will be blocked because the lock is exclusive; this ensures that only one thread can access the variable at the same time, so that the value of Variable V will not have the risk of confusion.



(变量 V 的自增操作=The autogenous operation of variable V)

Figure 7.1 Auto increment operation of Variable V

We call the area protected by the lock as **the critical area**.

If the critical area protection code cannot be interrupted, then the process is atomic operation. Non-interruptible means that there is no blocking or hardware interrupt in the critical area, and the atomic operation masks the hardware interrupt response of the current CPU core. Therefore, the atomic operation shall be as brief as possible.

In the multi-thread environment, operation of each variable needs to be treated with care. Multi-threaded programming can make our program clear and easy to implement, but it requires careful design. SylixOS has prepared us a large number of solutions to mutex problems during multi-threaded programming, such as semaphore, mutex lock, message queue and so on.

7.2 Inter-thread communication

In the process of executing the thread, it is inevitable to communicate with other threads. For example, if Thread A notifies Thread B of the processing results after processing an event, Thread B will continue to run after receiving the processing results of the event.

There are mainly the following types of inter-thread communication:

- **Mutex type communication:** shared resources require exclusive access, and semaphore and mutex can be used for mutex type communication.
- **Notification type communication:** the above Thread A informs Thread B, and the notification-type communication can be performed by using semaphore, event set and condition variable;

Message type

communication: a thread or an interrupt service program is only responsible for collecting data, but does not directly process data, and passes the data to another thread for data processing. Message queue can be used for message type communication.

SylixOS operating system provides rich means for inter-thread communication, as shown in Table 7.1. These communication means satisfy demands for inter-thread communication for embedded system software development.

Table 7.1 Inter-thread communication means

Inter-thread communication means	Purpose
Binary semaphore	Mutex type communication and notification type communication
Counting semaphore	Notification type communication
Mutex type semaphore	Mutex type communication
Event set	Notification type communication
Condition variable	Notification type communication
Message queue	Message type communication

7.3 SylixOS semaphore

As described in 7.1, multiple threads must implement mutex access or synchronous access (for example, Thread B waits for the result of Thread A to continue running) to the shared data through a certain method when reading or writing certain shared data (global variables, etc.). Among them, semaphore is one of the most common methods.

In fact, semaphore is an agreed mechanism: during mutex access to shared resources, it is agreed that when a thread gets the semaphore (Wait), other threads cannot get the semaphore again until the semaphore is released (Give). In the synchronization mechanism, it is agreed that the thread waiting for the semaphore (Take) (or waiting signal is more precise) shall be at the blocking state before receiving the semaphore until other thread send the semaphore (Post).

In general, only three operations can be performed on the semaphore: create a semaphore (Create), wait semaphore (Wait) or pend (Pend), give semaphore (Give) or post (Post). The operating system shall include a semaphore waiting thread queue (for storing thread waiting for semaphore). When the semaphore can be obtained, the operating system selects a thread which can get the semaphore from the queue to continue running according to a certain strategy.

SylixOS semaphore include four types: binary semaphore, counting semaphore, mutex semaphore (for short mutex) and read-write semaphore.

The value of binary semaphore is limited to FALSE and TRUE; while the minimum value of the counting semaphore is 0, and the maximum value is determined when creating the counting semaphore.

The binary semaphore is mainly used in the following situations:

- There is a resource which allows the thread to access, the binary semaphore is used as the mutex means, and the initial value is TRUE;
- The thread or interrupt informs another thread that an event occurs, and the initial value is FALSE.

The counting semaphore is mainly used in the following situations:

- There are n resources which allows the thread to access, the counting semaphore is used as the remaining count of resources, and the initial value is n;
- The thread or interrupt informs another thread that some event occurs, the counting semaphore is used as the event count, and the initial value is 0.

7.3.1 Binary semaphore

As mentioned above, the semaphore must be created before use. SylixOS provides the following functions to create a binary semaphore.

```
#include <SylixOS.h>
LW_HANDLE Lw_SemaphoreB_Create(CPCHAR      pcName,
                               BOOL        bInitValue,
                               ULONG       ulOption,
                               LW_OBJECT_ID *pulId);
```

Prototype analysis of Function Lw_SemaphoreB_Create:

- The function returns the handle of binary semaphore. For failure, return NULL and set the error number.
- Parameter *pcName* is the name of binary semaphore;
- Parameter *bInitValue* is the initial value of binary semaphore (FALSE or TRUE);
- Parameter *ulOption* is the creation option of binary semaphore, as shown in Table 7.2;
- Output parameter *pulId* is used to return the handle of binary semaphore (same with the return value), which can be NULL.

Table 7.2 Creation Options of binary semaphore

Macro name	Meaning
------------	---------

LW_OPTION_WAIT_PRIORITY	Wait in priority order
LW_OPTION_WAIT_FIFO	Wait in FIFO order
LW_OPTION_OBJECT_GLOBAL	Global object
LW_OPTION_OBJECT_LOCAL	Native object

SylixOS provides two semaphore waiting queues: Priority (LW_OPTION_WAIT_PRIORITY) and FIFO (LW_OPTION_WAIT_FIFO). The priority mode is to take the thread conforming to conditions from the queue for running according to the thread priority; the FIFO mode is to take the thread conforming to conditions from the queue for running according to the FIFO principle;

It might be noted that either LW_OPTION_WAIT_PRIORITY or LW_OPTION_WAIT_FIFO can be selected. Likewise, either LW_OPTION_OBJECT_GLOBAL or LW_OPTION_OBJECT_LOCAL^① can also be selected.

Different parameter **blnitValue** determines different purposes of binary semaphores, which can be used for mutex access of shared resources when the value of **blnitValue** is TRUE, as shown in Figure 7.2. When the value is FALSE, the **blnitValue** can be used for synchronization between multiple threads, as shown in Figure 7.3.

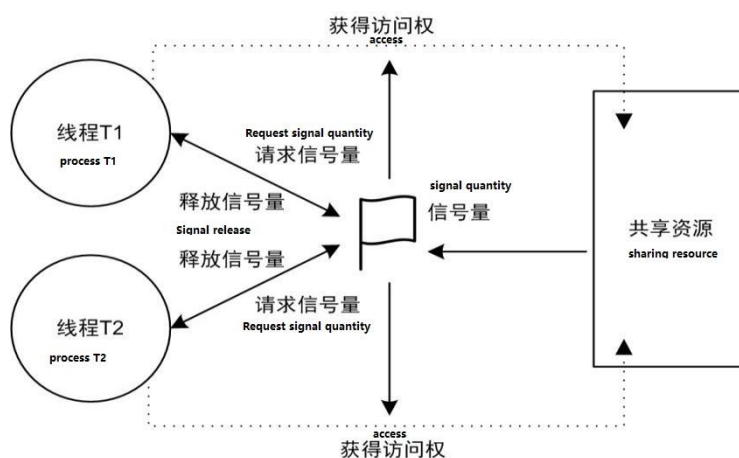


Figure 7.2 Mutex access of shared resources

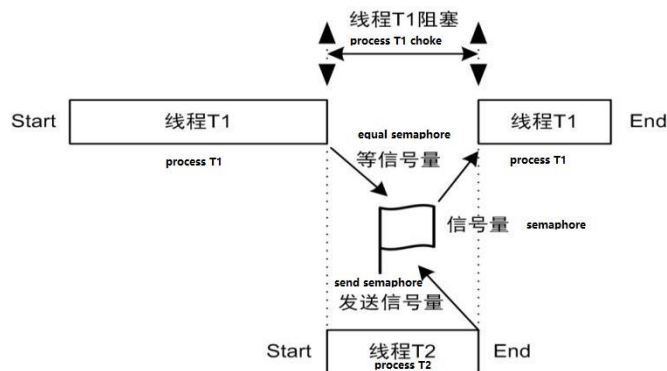


Figure 7.3 Thread synchronization

The binary semaphore not required can be deleted by calling the following functions, and SylixOS will recover kernel resources occupied (unknown errors will occur when trying to use the deleted binary semaphore).

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Delete(LW_HANDLE *pUId);
```

Prototype analysis of Function Lw_SemaphoreB_Delete:

- For success of the function, return 0. For failure, return the error number;
- Parameter **pUId** is the handle of binary semaphore.

If a thread needs to wait for a binary semaphore, the Lw_SemaphoreB_Wait function can be called. It might be noted that the interrupt service routine cannot call the Lw_SemaphoreB_Wait function to wait for a binary semaphore, because the function will block the currently executing task when the value of binary semaphore is FALSE, and the interrupt service routine is used to handle the most urgent things. Therefore, blockage shall not be allowed. Otherwise, other threads will not get the scheduling opportunity.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Wait(LW_HANDLE uId,
                        ULONG ulTimeout);
ULONG Lw_SemaphoreB_TryWait(LW_HANDLE uId);
```

Prototype analysis of above functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter **uId** is the handle of binary semaphore;
- Parameter **ulTimeout** is the waiting timeout, and the unit is Tick.

Parameter **ulTimeout**^① can use macros as shown in Table 7.3 in addition to digital.

Table 7.3 Available macros of Parameter **ulTimeout**

Macro name	Meaning
LW_OPTION_NOT_WAIT	Immediately exit without waiting
LW_OPTION_WAIT_INFINITE	Always wait
LW_OPTION_WAIT_A_TICK	Wait for a clock tick
LW_OPTION_WAIT_A_SECOND	Wait for a second

SylixOS provides a timeout mechanism for binary semaphore wait. It is returned immediately and `errno` is set as `ERROR_THREAD_WAIT_TIMEOUT` when the waiting time is timed out.

`Lw_SemaphoreB_TryWait` is a non-blocking semaphore wait function, and difference between the function and `Lw_SemaphoreB_Wait` is that if the initial value created by binary semaphore is `FALSE`, `Lw_SemaphoreB_TryWait` will immediately exit and return, while `Lw_SemaphoreB_Wait` will be blocked until awakened.

The interrupt service routine can use the `Lw_SemaphoreB_TryWait` function to try to wait for binary semaphore, because the `Lw_SemaphoreB_TryWait` function will immediately return when the value of binary semaphore is `FALSE`, and the current thread will not be blocked.

Release of a binary semaphore can call the `Lw_SemaphoreB_Post`, `Lw_SemaphoreB_Post2`, or `Lw_SemaphoreB_Release` function.

The `Lw_SemaphoreB_Post2` function can return the activated thread handle via Parameter ***pullId*** when return. If the parameter ***pullId*** is set as `NULL`, the behavior is the same with `Lw_SemaphoreB_Post`.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Post(LW_HANDLE uId);
```

Prototype analysis of Function `Lw_SemaphoreB_Post`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***uId*** is the handle of binary semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Post2(LW_HANDLE uId, LW_HANDLE *pullId);
```

Prototype analysis of Function `Lw_SemaphoreB_Post`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***uId*** is the handle of binary semaphore;
- Parameter ***pullId*** returns the activated thread ID.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Release(LW_HANDLE uId,
```



```

        ULONG      ulReleaseCounter,
        BOOL       *pbPreviousValue);

```

Prototype analysis of Function Lw_SemaphoreB_Release:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***uId*** is the handle of binary semaphore;
- Parameter ***ulReleaseCounter*** is the release counter of binary semaphore;
- Output parameter ***pbPreviousValue*** is used to receive the original binary semaphore state, which can be NULL.

The Lw_SemaphoreB_Release function is an advanced API, and multiple threads can be released at one time by calling the function when wait for the same semaphore (POSIX thread barrier calls the function to release multiple waiting threads, see Section 7.13).

Figure 7.4 shows the operation process of the basic operation function of binary semaphore between threads and between interrupt and thread.

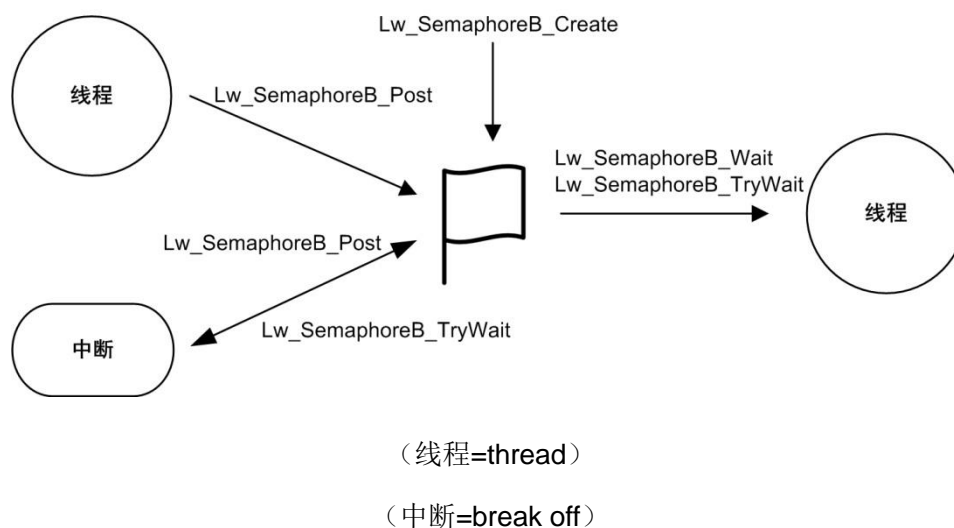


Figure 7.4 SylixOS binary semaphore

Calling the Lw_SemaphoreB_Clear function will clear binary semaphore, so that the initial value of binary semaphore will be set as FALSE.

```

#include <SylixOS.h>
ULONG Lw_SemaphoreB_Clear(LW_HANDLE uId);

```

Prototype analysis of Function Lw_SemaphoreB_Clear:

- The function returns the error number;

- Parameter ***uId*** is the handle of binary semaphore.

Calling the `Lw_SemaphoreB_Flush` function will release all threads waiting on the appointed semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Flush(LW_HANDLE uId,
                          ULONG pulThreadUnblockNum);
```

Prototype analysis of Function `Lw_SemaphoreB_Flush`:

- The function returns the error number;
- Parameter ***uId*** is the handle of binary semaphore;
- Output parameter ***pulThreadUnblockNum*** is used to receive the number of unblocked threads, which can be NULL.

The `Lw_SemaphoreB_Status` function returns state information of a valid semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_Status(LW_HANDLE uId,
                           BOOL pbValue,
                           ULONG pulOption,
                           ULONG pulThreadBlockNum);
```

Prototype analysis of Function `Lw_SemaphoreB_Status`:

- The function returns the error number;
- Parameter ***uId*** is the handle of binary semaphore;
- Output parameter ***pbValue*** is used to receive the current value of binary semaphore (FALSE or TRUE);
- Output parameter ***pulOption*** is used to receive creation options of binary semaphore;
- Output parameter ***pulThreadBlockNum*** is used to receive the number of the thread currently blocked at the binary semaphore.

Calling the `Lw_SemaphoreB_GetName` function can get the name of the appointed semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreB_GetName(LW_HANDLE uId, PCHAR pcName)
```

Prototype analysis of Function `Lw_SemaphoreB_GetName`:

- The function returns the error number;
- Parameter ***uId*** is the handle of binary semaphore;

- Output parameter **pcName** is the name of the binary semaphore, and pcName shall be pointed at a character array with size of LW_CFG_OBJECT_NAME_SIZE.

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreB_WaitEx(LW_HANDLE  ulId,
                               ULONG      ulTimeout,
                               PVOID      *ppvMsgPtr);
```

Prototype analysis of Function Lw_SemaphoreB_WaitEx:

- The function returns the error number;
- Parameter **ulId** is the handle of binary semaphore;
- Parameter **ulTimeout** is the waiting timeout, and the unit is Tick;
- Output parameter **ppvMsgPtr** (a null pointer) is used to receive messages transferred by the Lw_SemaphoreB_PostEx function.

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreB_PostEx(LW_HANDLE  ulId,
                               PVOID      pvMsgPtr);
```

Prototype analysis of Function Lw_SemaphoreB_PostEx:

- The function returns the error number;
- Parameter **ulId** is the handle of binary semaphore;
- Parameter **pvMsgPtr** is the message pointer (a null pointer which can point to any type of data). The message will be transferred to Output parameter pvMsgPtr of the Lw_SemaphoreB_WaitEx function.

The message passing function is added in the Lw_SemaphoreB_WaitEx and Lw_SemaphoreB_PostEx functions, and the additional message can be transferred in semaphore via Parameter **pvMsgPtr**. For example, the following program fragment shows the process:

```
threadA ()
{
    PVOID  ppvMsgPtr;

    Lw_SemaphoreB_WaitEx(ulId, &ppvMsgPtr);
}

threadB ()
{
    Lw_SemaphoreB_PostEx(ulId, "msg");
```

```
}

```

Note: thread A and thread B are two different threads, and the above process implements a simply inter-thread synchronous creation of additional message.

Function composition of Lw_SemaphoreB_WaitEx and Lw_SemaphoreB_PostEx has played the role of the traditional RTOS E-mail, and SylixOS does not provide API for E-mail.

The following program shows how to use SylixOS binary semaphore, the program creates two threads and a SylixOS binary semaphore, two threads respectively perform auto increment operation and printing for Variable `_G_iCount`, and SylixOS binary semaphore is used as mutex means of the access variable `_G_iCount`.

Program List 7.1 How to use SylixOS binary semaphore

```
#include <SylixOS.h>

static LW_HANDLE  _G_hLock;
static INT        _G_iCount = 0;

static PVOID tTestA (PVOID pvArg)
{
    INT    iError;

    while (1) {
        iError = Lw_SemaphoreB_Wait(_G_hLock, LW_OPTION_WAIT_INFINITE);
        if (iError != ERROR_NONE) {
            break;
        }

        _G_iCount++;
        printf("tTestA(): count = %d\n", _G_iCount);
        Lw_SemaphoreB_Post(_G_hLock);

        Lw_Time_MSsleep(500);
    }
    return (LW_NULL);
}

static PVOID tTestB (PVOID pvArg)
{
    INT    iError;

    while (1) {
```

```
        iError = Lw_SemaphoreB_Wait(_G_hLock, LW_OPTION_WAIT_INFINITE);
    if (iError != ERROR_NONE) {
        break;
    }

    _G_iCount++;
    printf("tTestB(): count = %d\n", _G_iCount);
    Lw_SemaphoreB_Post(_G_hLock);

    Lw_Time_MSsleep(500);
}

return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_HANDLE          hThreadAId;
    LW_HANDLE          hThreadBId;

    _G_hLock = Lw_SemaphoreB_Create("count_lock",
                                    LW_TRUE,
                                    LW_OPTION_WAIT_FIFO |
                                    LW_OPTION_OBJECT_LOCAL,
                                    LW_NULL);

    if (_G_hLock == LW_OBJECT_HANDLE_INVALID) {
        printf("semaphore create failed.\n");
        return (-1);
    }

    hThreadAId = Lw_Thread_Create("t_testa", tTestA, LW_NULL, LW_NULL);
    if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testa create failed.\n");
        return (-1);
    }

    hThreadBId = Lw_Thread_Create("t_testb", tTestB, LW_NULL, LW_NULL);
    if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testb create failed.\n");
        return (-1);
    }

    Lw_Thread_Join(hThreadAId, LW_NULL);
}
```

```

Lw_Thread_Join(hThreadBId, LW_NULL);

Lw_SemaphoreB_Delete(&_G_hLock);

return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./SemaphoreB
tTestA(): count = 1
tTestB(): count = 2
tTestA(): count = 3
tTestB(): count = 4
tTestA(): count = 5
tTestB(): count = 6

```

7.3.2 Counting semaphore

As mentioned above, counting semaphore is usually used for multiple threads to share a certain resource. For example, manage a certain equipment ID pool with semaphore. It is assumed that the ID pool can apply for 15 device IDs at the same time, and we use counting semaphore to perform mutex access for the ID pool in this case. For each equipment ID applied, the counting semaphore is reduced by 1, and it is reduced to 0, the thread for ID reapplication will be blocked. If the equipment ID is released, the counting semaphore is added by 1, and the new thread can be reapplied at the moment. The process is as shown in Figure 7.5.

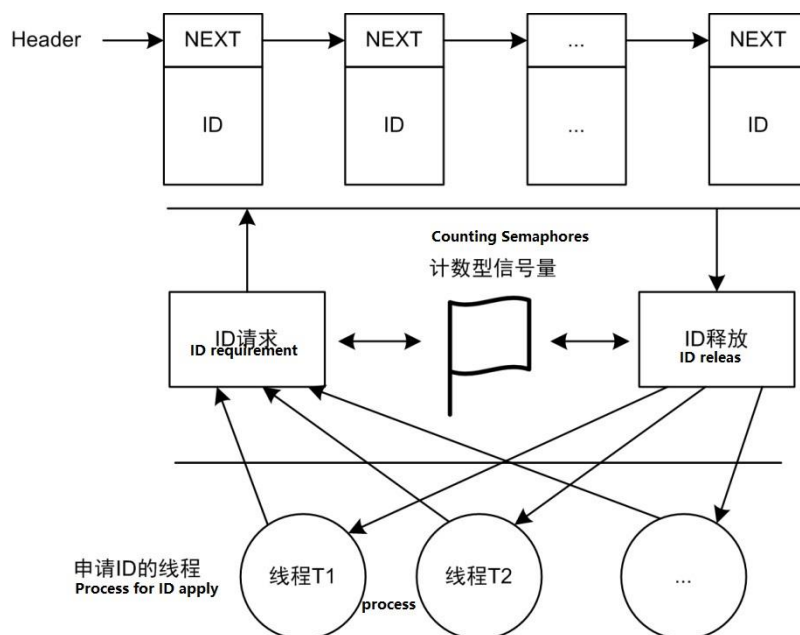


Figure 7.5 How to use counting semaphore

A SylixOS counting semaphore can call the `Lw_SemaphoreC_Create` function for creation, and a handle of counting semaphore will be returned after successful creation.

```
#include <SylixOS.h>
LW_HANDLE Lw_SemaphoreC_Create(CPCHAR      pcName,
                               ULONG       ulInitCounter,
                               ULONG       ulMaxCounter,
                               ULONG       ulOption,
                               LW_OBJECT_ID *pulId);
```

Prototype analysis of Function `Lw_SemaphoreC_Create`:

- For success of the function, return the handle of the counting semaphore. For failure, return NULL and set the error number;
- Parameter *pcName* is the name of counting semaphore;
- Parameter *ulInitCounter* is the handle of counting semaphore;
- Parameter *ulMaxCounter* is the maximum value of counting semaphore;
- Parameter *ulOption* is the creation option of the counting semaphore, as shown in Table 7.2.
- Output parameter *pulId* returns ID of counting semaphore (same with the return value), which can be NULL.

The value range of counting semaphore is $0 \leq \text{counted value (ulInitCounter)} < \text{ulMaxCounter}$ ^①. Especially, if the value of *ulInitCounter* is 0, it can be used for synchronization between multiple threads.

A counting semaphore not used can be deleted via calling the following functions. The deleted semaphore system automatically recovers the occupied system resources (unknown errors will occur when trying to use the counting semaphore deleted).

```
#include <SylixOS.h>
ULONG Lw_SemaphoreC_Delete(LW_HANDLE *pulId);
```

Prototype analysis of Function `Lw_SemaphoreC_Delete`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pulId* is the handle of counting semaphore.

If the thread needs to wait for a counted semaphore, the `Lw_SemaphoreC_Wait` function can be called. It might be noted that the interrupt service routine cannot call the `Lw_SemaphoreC_Wait` function to wait for a counting semaphore, because the

`Lw_SemaphoreC_Wait` will block the current thread when the value of counting semaphore is 0 (thread synchronization function).

```
#include <SylixOS.h>
ULONG   Lw_SemaphoreC_Wait(LW_HANDLE  uLId,
                           ULONG      ulTimeout);
ULONG   Lw_SemaphoreC_TryWait(LW_HANDLE uLId);
```

Prototype analysis of above two functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter *uLId* is the handle of counting semaphore;
- Parameter *ulTimeout* is the waiting timeout, and the unit is Tick.

Difference between `Lw_SemaphoreC_TryWait` and `Lw_SemaphoreC_Wait` is that if the current value of counting semaphore is FALSE, `Lw_SemaphoreC_TryWait` will immediately exit and return `ERROR_THREAD_WAIT_TIMEOUT`, while `Lw_SemaphoreC_Wait` will be blocked until awakened.

The interrupt service routine can use the `Lw_SemaphoreC_TryWait` function to try to wait for counting semaphore, because the `Lw_SemaphoreC_TryWait` function will immediately return when the value of counting semaphore is FALSE, and the current thread will not be blocked.

Releasing a counting semaphore can call the `Lw_SemaphoreC_Post` function.

```
#include <SylixOS.h>
ULONG   Lw_SemaphoreC_Post(LW_HANDLE  uLId);
```

Prototype analysis of Function `Lw_SemaphoreC_Post`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *uLId* is the handle of counting semaphore.

Releasing multiple counting semaphores at a time can call the `Lw_SemaphoreC_Release` function.

```
#include <SylixOS.h>
ULONG   Lw_SemaphoreC_Release(LW_HANDLE  uLId,
                              ULONG      ulReleaseCounter,
                              ULONG      *pulPreviousCounter);
```

Prototype analysis of Function `Lw_SemaphoreC_Release`:

- For success of the function, return 0. For failure, return the error number;
- Parameter *uLId* is the handle of counting semaphore;
- Parameter *ulReleaseCounter* is the release counter of counting semaphore;

- Output parameter ***pulPreviousCounter*** is used to receive original semaphore counter, which can be NULL.

Lw_SemaphoreC_Release is an advanced API, and POSIX read-write lock can call the function to release multiple read-write threads simultaneously (for POSIX read-write lock, see Section 7.8).

Figure 7.6 shows the operation process of the basic operation function of counting semaphore between threads and between interrupt and thread.

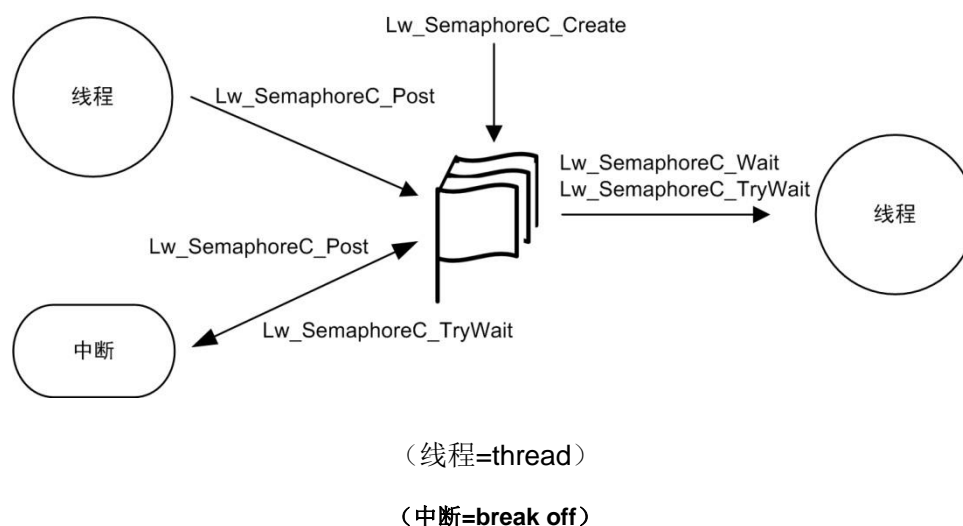


Figure 7.6 SylixOS counting semaphore

Calling the Lw_SemaphoreC_Clear function will clear counting semaphore, so that the initial value of counting semaphore will be set as 0.

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreC_Clear(LW_HANDLE  ulId);
```

Prototype analysis of Function Lw_SemaphoreC_Clear:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter ***ulId*** is the handle of counting semaphore;

Calling Lw_SemaphoreC_Flush will release all threads waiting on the appointed counting semaphore.

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreC_Flush(LW_HANDLE  ulId,
                             ULONG      *pulThreadUnblockNum);
```

Prototype analysis of Function Lw_SemaphoreC_Flush:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter **uId** is the handle of counting semaphore;
- Output parameter **pulThreadUnblockNum** is used to receive the number of unblocked threads, which can be NULL.

The following two functions can get state information of the appointed counting semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreC_Status(LW_HANDLE uId,
                           ULONG *pulCounter,
                           ULONG *pulOption,
                           ULONG *pulThreadBlockNum);
ULONG Lw_SemaphoreC_StatusEx(LW_HANDLE uId,
                              ULONG *pulCounter,
                              ULONG *pulOption,
                              ULONG *pulThreadBlockNum,
                              ULONG *pulMaxCounter);
```

Prototype analysis of above two functions:

- Above two functions return the error number;
- Parameter **uId** is the handle of counting semaphore;
- Output parameter **pulCounter** is used to receive the current value of counting semaphore;
- Output parameter **pulOption** is used to receive creation options of counting semaphore;
- Output parameter **pulThreadBlockNum** is used to receive the number of the thread currently blocked in counting semaphore.
- Output parameter **pulMaxCounter** is used to receive the maximum counted value of the counting semaphore.

The Lw_SemaphoreC_GetName function can get the name of the appointed counting semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreC_GetName(LW_HANDLE uId, PCHAR pcName);
```

Prototype analysis of Function Lw_SemaphoreC_GetName:

- The function returns the error number;
- Parameter **uId** is the handle of counting semaphore;

- Output parameter **pcName** is the name of the counting semaphore, and pcName shall be pointed at a character array with size of LW_CFG_OBJECT_NAME_SIZE.

The following program shows how to use SylixOS counting semaphore, and the program creates two threads and a SylixOS counting semaphore; counting semaphore is taken as the remaining count of resources, the initial number of resources is 5, and the maximum number is 100; Thread A is the consumer of the resource, and Thread B is the producer of the resource.

Program List 7.2 How to use SylixOS counting semaphore

```
#include <SylixOS.h>

static LW_HANDLE    _G_hResCntSema;

static PVOID tTestA (PVOID pvArg)
{
    INT    iError;

    while (1) {
        iError = Lw_SemaphoreC_Wait(_G_hResCntSema, LW_OPTION_WAIT_INFINITE);
        if (iError != ERROR_NONE) {
            break;
        }

        printf("tTestA(): get a resource\n");
    }
    return (LW_NULL);
}

static PVOID tTestB (PVOID pvArg)
{
    while (1) {
        Lw_Time_SSsleep(1);

        Lw_SemaphoreC_Post(_G_hResCntSema);
    }
    return (LW_NULL);
}

int main (int argc, char *argv[])
{
```

```

LW_HANDLE          hThreadAId;
LW_HANDLE          hThreadBId;

_G_hResCntSema = Lw_SemaphoreC_Create("res_sema",
                                     5,
                                     100,
                                     LW_OPTION_WAIT_FIFO |
                                     LW_OPTION_OBJECT_LOCAL,
                                     LW_NULL);

if (_G_hResCntSema == LW_OBJECT_HANDLE_INVALID) {
    printf("semaphore create failed.\n");
    return (-1);
}

hThreadAId = Lw_Thread_Create("t_testa", tTestA, LW_NULL, LW_NULL);
if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
    printf("t_testa create failed.\n");
    return (-1);
}

hThreadBId = Lw_Thread_Create("t_testb", tTestB, LW_NULL, LW_NULL);
if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
    printf("t_testb create failed.\n");
    return (-1);
}

Lw_Thread_Join(hThreadAId, LW_NULL);
Lw_Thread_Join(hThreadBId, LW_NULL);

Lw_SemaphoreC_Delete(&_G_hResCntSema);

return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./SemaphoreC
tTestA(): get a resource
tTestA(): get a resource
tTestA(): get a resource
tTestA(): get a resource
tTestA(): get a resource
tTestA(): get a resource

```

7.3.3 Mutex semaphore

During introduction to binary semaphore, it was discussed that if the ***blnitValue*** parameter is set as TRUE when binary semaphore is created, it can be used for mutex access of shared resources. Actually, mutex implemented by SylixOS binary semaphore is to initialize and mark a variable as 1, and reduce the variable by 1 (equal to 0 at the moment) at semaphore waiting (Wait). If another thread waits again, the semaphore will be blocked until released (variable added by 1), so that mutex access of shared resources is implemented.

If there are only two threads in the system, the above process has no problem. However, once multiple threads are involved, the above process will have the following problems:

A high-priority thread may also have access to the same shared resource (it is entirely possible). At the moment, only blocking wait can be performed. However, another thread with medium priority will control the thread occupying semaphore. The process causes failure in running high-priority thread for a long time (the situation is not allowed in SylixOS).

The problem in above process is the classic priority inversion, which will be continuously discussed in Section 7.5.

Mutex semaphore is used at the place where mutex access of shared resources, it can be understood as the binary semaphore with priority ceiling and priority inheritance mechanism (intended to solve the priority inversion problem) with an initial value of TRUE, and only the thread with mutex semaphore has the right to release mutex semaphore.

Note: Mutex semaphore needs to record the owner thread and adjust the priority, but the interrupt priority cannot be modified. Therefore, the interrupt service routine cannot operate mutex semaphore.

The following pseudocode fragment shows the process to use mutex semaphore.

```
Define global shared resources (_G_shared)

void *thread (void *)
{
    Wait for mutex(Wait)

    Operation to shared resources (_G_shared)

    Release the mutex (Post)

    Thread exit (Exit)
```

```

}

void main_func (void)
{
    Define mutex handler (semM)

    Create mutex (Create)

    Create thread (thread)

    Join the thread (join)

    Delete mutex (Delete)
}

```

A SylixOS mutex semaphore can be used after creation by calling the `Lw_SemaphoreM_Create` function, and the function will return a handle of mutex semaphore after successful creation.

```

#include <SylixOS.h>
LW_HANDLE Lw_SemaphoreM_Create(CPCHAR          pcName,
                               UINT8           ucCeilingPriority,
                               ULONG           ulOption,
                               LW_OBJECT_ID    *pulId);

```

Prototype analysis of Function `Lw_SemaphoreM_Create`:

- For success of the function, return the handle of the mutex semaphore. For failure, return NULL and set the error number;
- Parameter ***pcName*** is the name of mutex semaphore;
- Parameter ***ucCeilingPriority*** is valid when priority ceiling algorithm is used, and the parameter is the ceiling priority;
- Parameter ***ulOption*** is the creation option of mutex semaphore;
- Output parameter ***pulId*** returns the handle of mutex semaphore (same with the return value), which can be NULL.

Create options include the create option of the binary signal. In addition, the unique create option of mutex semaphore shown in Table 7.4 can also be used.

Table 7.4 Creation option of mutex semaphore

Macro name	Meaning
------------	---------

LW_OPTION_INHERIT_PRIORITY	Priority inheritance algorithm
LW_OPTION_PRIORITY_CEILING	Priority ceiling algorithm
LW_OPTION_NORMAL	Not checked during recursive locking (not recommended)
LW_OPTION_ERRORCHECK	Report errors during recursive locking
LW_OPTION_RECURSIVE	Support recursive locking

It might be noted either LW_OPTION_INHERIT_PRIORITY or LW_OPTION_PRIORITY_CEILING can be selected. Likewise, only one of LW_OPTION_NORMAL, LW_OPTION_ERRORCHECK and LW_OPTION_RECURSIVE can be selected.

A mutex semaphore not used can be deleted by calling the following functions. The semaphore system deleted will automatically recover system resources occupied (unknown errors will occur when trying to use the deleted mutex semaphore).

```
#include <SylixOS.h>
ULONG Lw_SemaphoreM_Delete(LW_HANDLE *p $u$ lId);
```

Prototype analysis of Function Lw_SemaphoreM_Delete:

- The function returns the error number;
- Parameter ***pulId*** is the handle of mutex semaphore;

If the thread needs to wait for a mutex semaphore, the Lw_SemaphoreM_Wait function can be called. Releasing a mutex semaphore can call the Lw_SemaphoreM_Post function.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreM_Wait(LW_HANDLE  $u$ lId,
                        ULONG  $u$ lTimeout);
```

Prototype analysis of Function Lw_SemaphoreM_Wait:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ulId*** is the handle of mutex semaphore;
- Parameter ***ulTimeout*** is the waiting timeout, and the unit is Tick.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreM_Post(LW_HANDLE  $u$ lId);
```

Prototype analysis of Function Lw_SemaphoreM_Post:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ulId*** is the handle of mutex semaphore;

It might be noted that only the owner can release mutex semaphore

Figure 7.7 shows the operation process of the basic operation function of mutex semaphore in the thread.

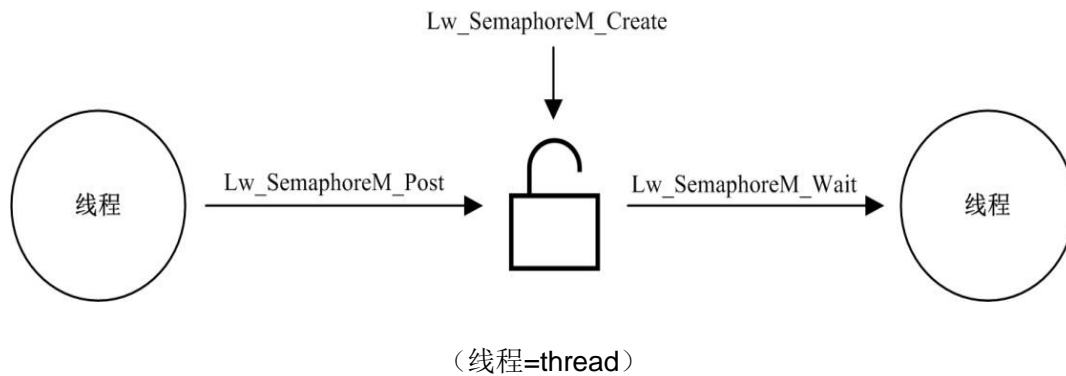


Figure 7.7 SylixOS mutex semaphore

The following functions can get state information of mutex semaphore.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreM_Status(LW_HANDLE ulId,
                           BOOL *pbValue,
                           ULONG *pulOption,
                           ULONG *pulThreadBlockNum);
ULONG Lw_SemaphoreM_StatusEx(LW_HANDLE ulId,
                              BOOL *pbValue,
                              ULONG *pulOption,
                              ULONG *pulThreadBlockNum,
                              LW_HANDLE *pulOwnerId);
```

Prototype analysis of above two functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter *ulId* is the handle of mutex semaphore;
- Output parameter *pbValue* is used to receive the current state of mutex semaphore;
- Output parameter *pulOption* is used to receive creation option of mutex semaphore;
- Output parameter *pulThreadBlockNum* is used to receive the number of threads currently blocked at the mutex semaphore.
- Output parameter *pulOwnerId* is used to receive the handle of the thread currently owning the mutex semaphore.

If you want to get the name of a mutex semaphore, the following functions can be called.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreM_GetName(LW_HANDLE uLld,
                             PCHAR pcName);
```

Prototype analysis of Function Lw_SemaphoreM_GetName:

- For success of the function, return 0. For failure, return the error number;
- Parameter *uLld* is the handle of mutex semaphore;
- Output parameter *pcName* is the name of the mutex semaphore, and *pcName* shall be pointed at a character array with size of LW_CFG_OBJECT_NAME_SIZE.

The following program shows how to use SylixOS mutex semaphore, the program creates two threads with different priorities and a SylixOS mutex semaphore, two threads perform auto increment operation and printing for Variable `_G_iCount` respectively, and SylixOS mutex semaphore is used as mutex means of Variable `_G_iCount`. Where, the mutex semaphore uses priority inheritance algorithm.

Program List 7.3 How to use SylixOS counting mutex semaphore

```
#include <SylixOS.h>

static LW_HANDLE _G_hLock;
static INT _G_iCount = 0;

static PVOID tTestA (PVOID pvArg)
{
    INT iError;

    while (1) {
        iError = Lw_SemaphoreM_Wait(_G_hLock, LW_OPTION_WAIT_INFINITE);
        if (iError != ERROR_NONE) {
            break;
        }

        _G_iCount++;
        printf("tTestA(): count = %d\n", _G_iCount);
        Lw_SemaphoreM_Post(_G_hLock);

        Lw_Time_MSsleep(500);
    }

    return (LW_NULL);
}
```

```
    }

static PVOID tTestB (PVOID pvArg)
{
    INT    iError;

    while (1) {
        iError = Lw_SemaphoreM_Wait(_G_hLock, LW_OPTION_WAIT_INFINITE);
        if (iError != ERROR_NONE) {
            break;
        }

        _G_iCount++;
        printf("tTestB(): count = %d\n", _G_iCount);

        Lw_SemaphoreM_Post(_G_hLock);

        Lw_Time_MSsleep(500);
    }
    return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_CLASS_THREADATTR  threadattr;
    LW_HANDLE             hThreadAId;
    LW_HANDLE             hThreadBId;

    _G_hLock = Lw_SemaphoreM_Create("count_lock",
                                    LW_PRIO_HIGH,
                                    LW_OPTION_WAIT_FIFO |
                                    LW_OPTION_OBJECT_LOCAL |
                                    LW_OPTION_INHERIT_PRIORITY |
                                    LW_OPTION_ERRORCHECK,
                                    LW_NULL);

    if (_G_hLock == LW_OBJECT_HANDLE_INVALID) {
        printf("mutex create failed.\n");
        return (-1);
    }

    Lw_ThreadAttr_Build(&threadattr,
                       4 * LW_CFG_KB_SIZE,
                       LW_PRIO_NORMAL - 1,
```

```

        LW_OPTION_THREAD_STK_CHK,
        LW_NULL);
hThreadAId = Lw_Thread_Create("t_testa", tTestA, &threadattr, LW_NULL);
if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
    printf("t_testa create failed.\n");
    return (-1);
}

Lw_ThreadAttr_Build(&threadattr,
                    4 * LW_CFG_KB_SIZE,
                    LW_PRIO_NORMAL,
                    LW_OPTION_THREAD_STK_CHK,
                    LW_NULL);
hThreadBId = Lw_Thread_Create("t_testb", tTestB, &threadattr, LW_NULL);
if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
    printf("t_testb create failed.\n");
    return (-1);
}

Lw_Thread_Join(hThreadAId, LW_NULL);
Lw_Thread_Join(hThreadBId, LW_NULL);

Lw_SemaphoreM_Delete(&_G_hLock);

return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./SemaphoreM
tTestA(): count = 1
tTestB(): count = 2
tTestA(): count = 3
tTestB(): count = 4
tTestA(): count = 5
tTestB(): count = 6

```

7.3.4 Read-write semaphore

As introduced in 7.8 POSIX read-write lock, when there are multiple readers and single writer, purely using mutex semaphore will greatly weaken the processing performance of the multi-threaded operating system. In order to satisfy requirements for

high concurrency processing speed, SylixOS introduces read-write semaphore, and its application scenario is similar to POSIX read-write lock.

SylixOS read-write semaphore satisfies the principle of write priority, that is to say, if the write semaphore has existed, the read semaphore cannot be applied until the write semaphore is released. However, when the read semaphore has existed, the read semaphore may be requested again. The mechanism satisfies high concurrency of read.

A SylixOS read-write semaphore can be used after creation by calling the `Lw_SemaphoreRW_Create` function, and the function will return a handle of read-write semaphore after successful creation.

```
#include <SylixOS.h>
LW_HANDLE Lw_SemaphoreRW_Create(CPCHAR      pcName,
                               ULONG        ulOption,
                               LW_OBJECT_ID *pulId);
```

Prototype analysis of Function `Lw_SemaphoreRW_Create`:

- For success of the function, return the handle of the read-write semaphore. For failure, return NULL and set the error number;
- Parameter *pcName* is the name of read-write semaphore;
- Parameter *ulOption* is creation option of read-write semaphore;
- Output parameter *pulId* returns the handle of read-write semaphore (same with the return value), which can be NULL.

A read-write semaphore not used can be deleted by calling the following functions. The semaphore system deleted will automatically recover system resources occupied (unknown errors will occur when trying to use the deleted read-write semaphore).

```
#include <SylixOS.h>
ULONG Lw_SemaphoreRW_Delete(LW_HANDLE *pulId);
```

Prototype analysis of Function `Lw_SemaphoreRW_Delete`:

- For success of the function, return `ERROR_NONE`. For failure, return the error number;
- Parameter *pulId* is the handle of read-write semaphore.

If the thread needs to wait for a read semaphore, the `Lw_SemaphoreRW_RWait` function can be called. If the thread needs to wait for a write semaphore, the `Lw_SemaphoreRW_WWait` can be called. Releasing a read-write semaphore can call the `Lw_SemaphoreRW_Post` function.

```
#include <SylixOS.h>
ULONG Lw_SemaphoreRW_RWait(LW_HANDLE ulId,
```

```
ULONG    ulTimeout);
```

Prototype analysis of Function Lw_SemaphoreRW_RWait:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulld* is the handle of read semaphore;
- Parameter *ulTimeout* is the waiting timeout, and the unit is Tick.

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreRW_WWait(LW_HANDLE ulld,
                               ULONG    ulTimeout);
```

Prototype analysis of Function Lw_SemaphoreRW_WWait:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulld* is the handle of write semaphore;
- Parameter *ulTimeout* is the waiting timeout, and the unit is Tick.

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreRW_Post(LW_HANDLE ulld);
```

Prototype analysis of Function Lw_SemaphoreRW_Post:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulld* is the handle of read-write semaphore;

It might be noted that only the owner of read-write semaphore can release the read-write semaphore.

Calling the following functions can get detailed information of read-write semaphore:

```
#include <SylixOS.h>
ULONG    Lw_SemaphoreRW_Status(LW_OBJECT_HANDLE ulld,
                               ULONG            *pulRWCount,
                               ULONG            *pulRPend,
                               ULONG            *pulWPend,
                               ULONG            *pulOption,
                               LW_OBJECT_HANDLE *pulOwnerId);
```

Prototype analysis of Function Lw_SemaphoreRW_Status:

- For success of the function, return ERROR_NONE. For failure, return the error number;

- Parameter ***ulld*** is the handle of read-write semaphore;
- Parameter ***pulRWCount*** returns the number of threads which are performing concurrent operation of read-write semaphore, and the parameter can be NULL.
- Parameter ***pulRPend*** returns the number of blocks of current read operation, and the parameter can be NULL.
- Parameter ***pulWPend*** returns the number of blocks of current write operation, and the parameter can be NULL.
- Parameter ***pulOption*** returns option information of current read-write semaphore, and the parameter can be NULL.
- Parameter ***pulOwnerld*** returns the owner ID of current write semaphore, and the parameter can be NULL.

7.4 POSIX semaphore

There are two types of POSIX semaphore: anonymous semaphore and named semaphore. The anonymous semaphore only exists in memory, which requires that the thread using semaphore can have access to memory. Therefore, the anonymous semaphore can be applied for inter-thread communication in the same process, and the memory shall be mapped to the address space between different processes. Named semaphores can be accessed via name and can therefore be applied for interprocess communication (see Chapter 9 Interprocess Communication). The essence of the POSIX semaphore is the counting semaphore.

The POSIX semaphore is defined as the `sem_t` type, and the variable of the `sem_t` type shall be defined before use. For example:

```
sem_t sem;
```

7.4.1 POSIX anonymous semaphore

A POSIX anonymous semaphore can be used after creation by calling the `sem_init` function.

```
#include <semaphore.h>
int sem_init(sem_t *psem, int pshared, unsigned int value);
```

Prototype analysis of Function `sem_init`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Output parameter ***psem*** returns the pointer of POSIX semaphore;

- Parameter ***pshared*** identifies whether the POSIX semaphore is shared by the process (SylixOS does not use the item);
- Parameter ***value*** is the initial value of the POSIX semaphore.

After a POSIX anonymous semaphore is used (and it is no longer used in the future), it shall be deleted by calling the `sem_destroy` function, and SylixOS will recover kernel resources occupied by the semaphore.

```
#include <semaphore.h>
int sem_destroy(sem_t *psem);
```

Prototype analysis of Function `sem_destroy`:

- The function returns the error number;
- Parameter ***psem*** is the pointer of POSIX semaphore;

If the thread needs to wait for a POSIX semaphore, the `sem_wait` function can be called, and the interrupt service routine cannot call any API of the POSIX semaphore. Release a semaphore to use the `sem_post` function.

```
#include <semaphore.h>
int sem_wait(sem_t *psem);
int sem_trywait(sem_t *psem);
int sem_timedwait(sem_t *psem, const struct timespec *abs_timeout);
int sem_reltimedwait_np(sem_t *psem, const struct timespec *rel_timeout);
```

Prototype analysis of above several functions:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***psem*** is the pointer of POSIX semaphore;
- Parameter ***timeout*** is the waiting absolute timeout;
- Parameter ***rel_timeout*** is the waiting relative timeout.

`sem_trywait` is the "try wait" version of `sem_wait`. When the value of the POSIX semaphore is 0, `sem_wait` will be blocked until awakened, while `sem_trywait` will return immediately.

The `sem_timedwait` is the version of the `sem_wait` with wait timeout, and the timeout is the absolute wait timeout. The absolute timeout can be got by adding a relative timeout based on the current time during use. For example:

```
struct timespec ts;

clock_gettime(CLOCK_REALTIME, &ts);
ts.tv_sec += 1;
```

```
sem_timedwait(&sem, &ts);
```

`sem_reltimedwait_np` is the non-POSIX standard version of `sem_timedwait`, and Parameter `rel_timeout` is relative wait timeout. For example:

```
struct timespec ts;

ts.tv_sec = 1;
ts.tv_nsec = 0;
sem_reltimedwait_np(&sem, &ts);
```

It can be seen that the `sem_reltimedwait_np` is more convenient than the `sem_timedwait` in use.

```
#include <semaphore.h>
int sem_post(sem_t *psem);
```

Prototype analysis of Function `sem_post`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***psem*** is the pointer of POSIX semaphore;

Figure 7.8 shows the operation process of the basic operation function of the anonymous semaphore in the thread.

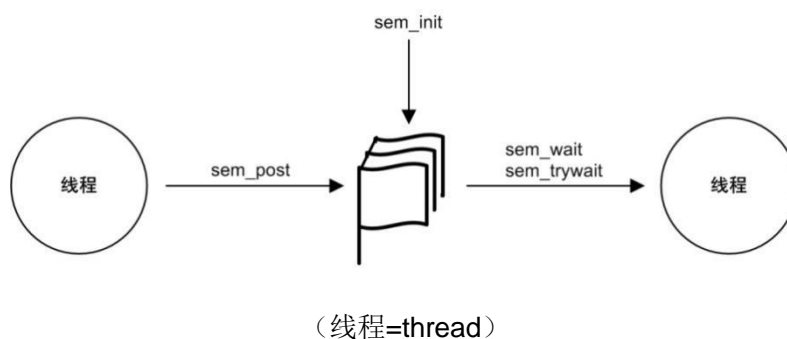


Figure 7.8 POSIX anonymous semaphore

The `sem_getvalue` function can be used to retrieve the semaphore value. It might be noted that when we try to use the value just read out, the semaphore value may have changed. It is not suggested to use the `sem_getvalue` function unless other synchronization mechanisms are used to avoid this competition.

```
#include <semaphore.h>
int sem_getvalue(sem_t *psem, int *pvalue);
```

Prototype analysis of Function `sem_getvalue`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *psem* is the pointer of POSIX semaphore;
- Output parameter *pivalue* is used to receive the current counted value of the POSIX semaphore.

The following program shows how to use POSIX semaphore, the program creates two threads and a POSIX semaphore, two threads perform auto increment operation and printing for Variable count respectively, and POSIX semaphore is used as mutex means of the access variable count.

Program List 7.4 Use of POSIX semaphore

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

static sem_t lock;
static int count = 0;

static void *thread_a (void *arg)
{
    while (1) {
        sem_wait(&lock);
        count++;
        printf("thread_a(): count = %d\n", count);
        sem_post(&lock);
        usleep(500 * 1000);
    }
    return (NULL);
}

static void *thread_b (void *arg)
{
    while (1) {
        sem_wait(&lock);
        count++;
        printf("thread_b(): count = %d\n", count);
        sem_post(&lock);
        usleep(500 * 1000);
    }
    return (NULL);
}
```

```
int main (int argc, char *argv[])
{
    pthread_t      threada_tid;
    pthread_t      threadb_tid;
    int            ret;

    ret = sem_init(&lock, 1, 1);
    if (ret != 0) {
        fprintf(stderr, "semaphore create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threada_tid, NULL, thread_a, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threadb_tid, NULL, thread_b, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    pthread_join(threada_tid, NULL);
    pthread_join(threadb_tid, NULL);

    sem_destroy(&lock);

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./posix_sema
thread_a(): count = 1
thread_b(): count = 2
thread_a(): count = 3
thread_b(): count = 4
thread_a(): count = 5
```

7.5 Priority inversion

7.5.1 What's priority inversion

We present an example of shared resource competition above: two threads shall perform increment operation of the same variable V (initial value of 0) simultaneously. The solution to shared resource competition is to add a lock. The lock is occupied before access to variable V, and released after access. In general, we can use THE binary semaphore with the initial value of TRUE or the counting semaphore with the initial value of 1 as the lock.

Now we will slightly change the example. There are three threads (thread A, thread B and thread C) and a variable V, and thread A and thread B shall have access to Variable V simultaneously. Obviously, we need a lock (i.e., Lock L of the protected variable V).

The priorities of Thread A, thread B and thread C are 1, 2 and 3 respectively, i.e., priority: thread A > thread C > thread B.

We assume that now thread A and thread C are at the blocking state, and thread B is at running state. Thread B occupies lock L (#1 in Figure 7.9). At the moment, the event waited by Thread C enters the ready state. The priority of Thread C is higher than that of thread B, and thread C will preempt execution of Thread B (#2 in Figure 7.9); at the moment, the event waited by Thread A enters the ready state. The priority of Thread A is higher than that of thread C, and thread A will preempt execution of Thread C (#3 in Figure 7.9); At the moment, Thread A also applies for lock L. Lock L has been occupied by Thread B, Thread A must block and wait for Lock L to be released by Thread B, while Thread C with medium priority will continue to run (Figure 7.9 #4), At the moment, the running statuses of 3 threads are as follows: thread A blocked, thread C running normally, and thread B blocked, which obviously violates the real-time principle of RTOS.

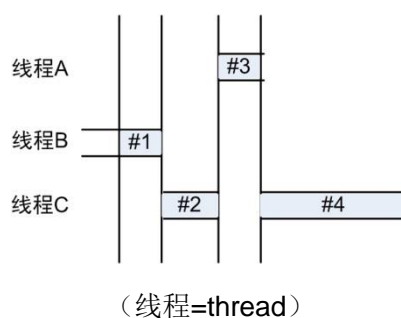


Figure 7.9 Priority inversion

When a high-priority thread has access to shared resources via the semaphore mechanism, the semaphore has been occupied by a low-priority thread. However, the low-priority thread may be occupied by other medium-priority threads when access to

shared resources. Therefore, high-priority threads are blocked by many threads with lower priority, and we call the phenomenon **priority inversion**.

7.5.2 How to solve priority inversion

There are two ways to solve the priority inversion problem: priority ceiling and priority inheritance (supported by SylixOS mutex semaphore simultaneously).

The priority ceiling is to raise the priority of a thread to the highest priority of all threads which can have access to the resource when applying for a certain shared resource. The priority is called the priority ceiling of the resource. The method is simple and easy to implement, without complicated judgment. Even if the thread blocks running of the high-priority thread, the priority of the thread will be promoted as long as the thread has access to shared resources.

Priority inheritance is shown as follows: when Thread A applies for the shared resource V which is being used by Thread B, if it is found that the priority of Thread B is lower than its own priority through comparison, the priority of Thread B is promoted to its own priority, and recovered to the original priority after Thread B releases the shared resource V. The method dynamically changes the priority of the thread only when the low-priority thread occupying the resource blocks the high-priority thread.

Both binary semaphore and counting semaphore do not support priority ceiling and priority inheritance, and only mutex semaphore supports priority ceiling and priority inheritance.

7.6 POSIX mutex semaphore

POSIX mutex semaphore is the “POSIX” interface function performing mutex access to shared resources, and the function implemented is same with that of SylixOS semaphore.

The type of POSIX mutex semaphore is `pthread_mutex_t`. A variable of `pthread_mutex_t` type shall be defined during use. For example:

```
pthread_mutex_t mutex;
```

A POSIX mutex semaphore must be initialized firstly before use. The initial value of the mutex can be set as `PTHREAD_MUTEX_INITIALIZER` (static initialization), and dynamic initialization can also be performed by calling the `pthread_mutex_init` function.

If the thread needs to wait for a mutex semaphore, the `pthread_mutex_lock` function can be called. Release a mutex semaphore to use the `pthread_mutex_unlock` function.

A mutex semaphore will be deleted by calling the `pthread_mutex_destroy` function after use, and SylixOS will recover kernel resources occupied by the mutex semaphore. It

might be noted that unknown errors will occur if a deleted semaphore is tried to be used again.

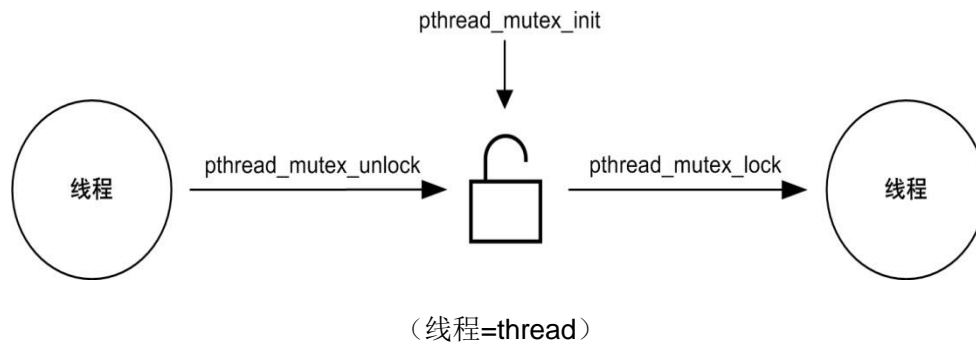


Figure 7.10 POSIX mutex semaphore

The thread has attribute objects, SPOSIX mutex semaphore has its own attribute objects, and attribute blocks of POSIX mutex semaphore shall be used to create a POSIX mutex semaphore. The type of POSIX mutex semaphore attribute block is `pthread_mutexattr_t`. A variable of `pthread_mutexattr_t` type shall be defined during use. For example:

```
pthread_mutexattr_t mutexattr;
```

7.6.1 Mutex semaphore attribute block

1. Initialization and deletion of the mutex semaphore attribute block

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *pmutexattr);
```

Prototype analysis of Function `pthread_mutexattr_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pmutexattr*** is the pointer of POSIX mutex semaphore attribute block.

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *pmutexattr);
```

Prototype analysis of Function `pthread_mutexattr_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pmutexattr*** is the pointer of POSIX mutex semaphore attribute block.

2. Set and get the type of mutex semaphore attribute block

```
#include <pthread.h>
```

```
int pthread_mutexattr_settype(pthread_mutexattr_t *pmutexattr,
                             int type);
```

Prototype analysis of Function pthread_mutexattr_settype:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutexattr* is the pointer of POSIX mutex semaphore attribute block;
- Parameter *type* is the type of POSIX mutex semaphore attribute block.

The type of mutex semaphore attribute block can use the macro shown in Table 7.5.

Table 7.5 Type of mutex semaphore attribute block

Macro name	Meaning
PTHREAD_MUTEX_NORMAL	Generate deadlock at recursive locking
PTHREAD_MUTEX_ERRORCHECK	Return error at recursive locking
PTHREAD_MUTEX_RECURSIVE	Support recursive locking
PTHREAD_MUTEX_FAST_NP	Equivalent to PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK_NP	Equivalent to PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE_NP	Equivalent to PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_DEFAULT	Equivalent to PTHREAD_MUTEX_RECURSIVE

```
#include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t *pmutexattr,
                             int type);
```

Prototype analysis of Function pthread_mutexattr_gettype:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutexattr* is the pointer of POSIX mutex semaphore attribute block;
- Output parameter *type* is the type of mutex semaphore attribute block.

3. Set and get algorithm type of the mutex semaphore attribute block

```
#include <pthread.h>
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *pmutexattr,
                                  int protocol);
```

Prototype analysis of Function pthread_mutexattr_setprotocol:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutexattr* is the pointer of POSIX mutex semaphore attribute block;
- Parameter *protocol* is the algorithm type of POSIX mutex semaphore attribute block.

The type of mutex semaphore attribute block can use the macro shown in Table 7.6.

Table 7.6 Algorithm type of the mutex semaphore attribute block

Macro name	Meaning
PTHREAD_PRIO_NONE	Priority inheritance algorithm, wait in FIFO order
PTHREAD_PRIO_INHERIT	Priority inheritance algorithm, wait in priority order
PTHREAD_PRIO_PROTECT	Priority ceiling

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *pmutexattr,
    int protocol
);
```

Prototype analysis of Function pthread_mutexattr_getprotocol:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutexattr* is the pointer of POSIX mutex semaphore attribute block;
- Parameter *protocol* is the algorithm type of POSIX mutex semaphore attribute block.

4. Set and get the ceiling priority of mutex semaphore attribute block.

```
#include <pthread.h>
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *pmutexattr,
    int prioceiling);
```

Prototype analysis of Function pthread_mutexattr_setprioceiling:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutexattr* is the pointer of POSIX mutex semaphore attribute block;
- Parameter *prioceiling* is the ceiling priority of POSIX mutex semaphore attribute block.

```
#include <pthread.h>
int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t *pmutexattr,
    int prioceiling
);
```

Prototype analysis of Function pthread_mutexattr_setprioceiling:

- For success of the function, return 0. For failure, return the error number;

- Parameter ***pmutexattr*** is the pointer of POSIX mutex semaphore attribute block;
- Output parameter ***prioceiling*** is the ceiling priority of POSIX mutex semaphore attribute block.

5. Set and get process shared attributes of the mutex semaphore attribute block

```
#include <pthread.h>
int  pthread_mutexattr_setpshared(pthread_mutexattr_t *pmutexattr,
                                int                    pshared);
```

Prototype analysis of Function pthread_mutexattr_setpshared^①:

- The function returns 0;
- Parameter ***pmutexattr*** is the pointer of POSIX mutex semaphore attribute block;
- Parameter ***pshared*** identifies whether the POSIX mutex semaphore attribute block is shared by processes.

Process sharing parameters can use the macro shown in Table 7.7.

Table 7.7 Parameter shared by processes

Macro name	Meaning
PTHREAD_PROCESS_SHARED	Process share
PTHREAD_PROCESS_PRIVATE	Process private

```
#include <pthread.h>
int  pthread_mutexattr_getpshared(const pthread_mutexattr_t *pmutexattr,
                                int                    *pshared);
```

Prototype analysis of Function pthread_mutexattr_getpshared:

- The function returns 0;
- Parameter ***pmutexattr*** is the pointer of POSIX mutex semaphore attribute block;
- Output parameter ***pshared*** identifies whether the POSIX mutex semaphore attribute block is shared by processes.

It might be noted that SylixOS is always private to the process (PTHREAD_PROCESS_PRIVATE).

7.6.2 Mutex semaphore

1. Initialization and deletion of the mutex semaphore


```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *pmutex,
                      const pthread_mutexattr_t *pmutexattr);
```

Prototype analysis of Function pthread_mutex_init:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutex* is the pointer of POSIX mutex semaphore;
- Parameter *pmutexattr* is the pointer of POSIX mutex semaphore attribute block, which can be NULL;

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *pmutex);
```

Prototype analysis of Function pthread_mutex_destroy:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutex* is the pointer of POSIX mutex semaphore;

2. Wait of mutex semaphore

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *pmutex);
int pthread_mutex_trylock(pthread_mutex_t *pmutex);
int pthread_mutex_timedlock(pthread_mutex_t *pmutex,
                             const struct timespec *abs_timeout);
int pthread_mutex_reltimedlock_np(pthread_mutex_t *pmutex,
                                  const struct timespec *rel_timeout);
```

Prototype analysis of above functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutex* is the pointer of POSIX mutex semaphore;
- Parameter *abs_timeout* is the waiting absolute timeout;
- Parameter *rel_timeout* is the waiting relative timeout.

The pthread_mutex_trylock is the "try wait" version of pthread_mutex_lock. When POSIX mutex semaphore has been occupied, the pthread_mutex_lock will be blocked until awakened, while the pthread_mutex_trylock will return immediately.

The pthread_mutex_timedlock is the version of the pthread_mutex_lock with wait timeout, and the abs_timeout is the absolute wait timeout. The absolute timeout can be got by adding a relative timeout based on the current time during use.

The pthread_mutex_reltimedlock_np is the non-POSIX standard version of the pthread_mutex_timedlock, and Parameter rel_timeout is relative wait timeout.

3. Release of mutex semaphore

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *pmutex);
```

Prototype analysis of Function pthread_mutex_unlock:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutex* is the pointer of POSIX mutex semaphore.

4. Set and get the ceiling priority of mutex semaphore.

```
#include <pthread.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *pmutex,
                                int prioceiling);
```

Prototype analysis of Function pthread_mutex_setprioceiling:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutex* is the pointer of POSIX mutex semaphore;
- Parameter *prioceiling* is the ceiling priority of POSIX mutex semaphore.

```
#include <pthread.h>
int pthread_mutex_getprioceiling(pthread_mutex_t *pmutex,
                                int *prioceiling);
```

Prototype analysis of Function pthread_mutex_getprioceiling:

- For success of the function, return 0. For failure, return the error number;
- Parameter *pmutex* is the pointer of POSIX mutex semaphore;
- Output parameter *prioceiling* is the ceiling priority of POSIX mutex semaphore.

The following program shows how to use POSIX mutex semaphore, the program creates two threads and a POSIX mutex semaphore, two threads perform auto increment operation and printing for Variable count respectively, and POSIX mutex semaphore is used as mutex for access to Variable count. Where, the mutex semaphore uses priority inheritance algorithm, and wait in priority order.

Program List 7.5 Use of POSIX mutex semaphore

```
#include <stdio.h>
#include <pthread.h>

static pthread_mutex_t lock;
static int count = 0;

static void *thread_a (void *arg)
```

```
{
while (1) {
    pthread_mutex_lock(&lock);
    count++;
    printf("thread_a(): count = %d\n", count);
    pthread_mutex_unlock(&lock);
    usleep(500 * 1000);
}
return (NULL);
}

static void *thread_b (void *arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        count++;
        printf("thread_b(): count = %d\n", count);
        pthread_mutex_unlock(&lock);
        usleep(500 * 1000);
    }
    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_mutexattr_t    mutexattr;
    pthread_t              threada_tid;
    pthread_t              threadb_tid;
    int                    ret;

    pthread_mutexattr_init(&mutexattr);
    pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_NORMAL);
    pthread_mutexattr_setprotocol(&mutexattr, PTHREAD_PRIO_INHERIT);

    ret = pthread_mutex_init(&lock, &mutexattr);
    if (ret != 0) {
        fprintf(stderr, "mutex create failed.\n");
        return (-1);
    }

    pthread_mutexattr_destroy(&mutexattr);

    ret = pthread_create(&threada_tid, NULL, thread_a, NULL);
```

```
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threadb_tid, NULL, thread_b, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    pthread_join(threada_tid, NULL);
    pthread_join(threadb_tid, NULL);

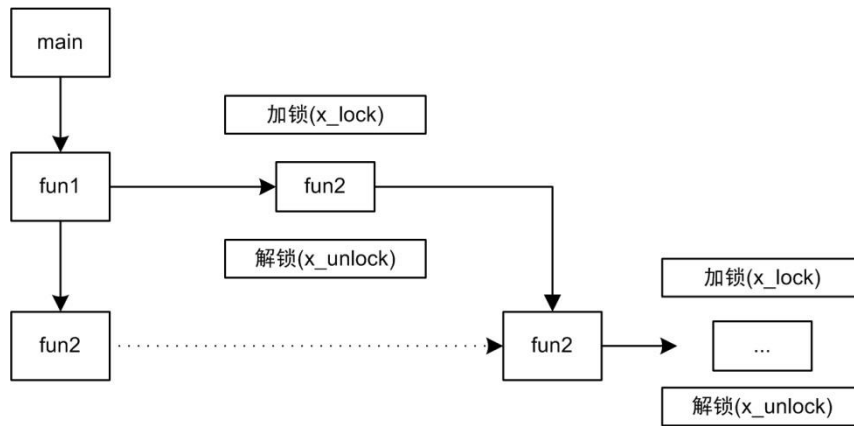
    pthread_mutex_destroy(&lock);

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./posix_mutex
thread_a(): count = 1
thread_b(): count = 2
thread_a(): count = 3
thread_b(): count = 4
thread_a(): count = 5
thread_b(): count = 6
```

The above program sets the type of mutex attribute as `PTHREAD_MUTEX_NORMAL`, which will not check recursive locking (a possible recursive locking as shown in the Figure 7.11), and deadlock will occur in case of recursive locking as shown in the following figure (we will discuss deadlock concept in details in Section 7.7). Therefore, it is not suggested to use the lock of the type in actual application, and SylixOS suggests to set the type of lock as `PTHREAD_MUTEX_ERRORCHECK`. The lock of the type will automatically check lock recursion when locked, and Error `EDEADLK` will be returned in case of lock recursion.



(加锁=lock)

(解锁=unlock)

Figure 7.11 Recursive locking

Program List can be simplified as the following pseudocode, which locks shared resources in different thread contexts. so as to well avoid recursive lock.

```

thread_a ()
{
    lock(lock)
    count++
    unlock(lock)
}

thread_b ()
{
    lock(lock)
    count++
    unlock(lock)
}

main ()
{
    Create lock(lock)
    Create thread thread_a thread_b
}
  
```

7.7 Deadlock

7.7.1 What's deadlock

The so-called deadlock refers to the indefinite standoff situation where multiple threads circularly wait for resources occupied by other parties. Obviously, various threads involving deadlock will be at blocking state without external force.

Just like two people passing the single-plank bridge, if both of them want to pass firstly, and refuse to retreat, deadlock will occur due to resource competition; however, if either of them goes on the bridge after checking that there is no person at the opposite side of the bridge, the problem will be solved.

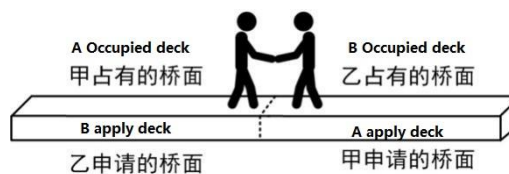


Figure 7.12 Deadlock

7.7.2 Generation conditions of deadlock

If there are four necessary conditions in the computer system **simultaneously**, deadlock will occur. In other words, as long as one of the following four conditions not be provided, the system will not be deadlocked.

- Mutex conditions

That is to say, a certain resource can only be occupied by a thread for a period, and cannot be occupied by two or more than two threads simultaneously. The exclusive resources such as CD-ROM driver and printer can only be occupied by other threads after actively released by the thread occupying the resources. It is determined by attribute of resources. For example, the single-plank bridge is a kind of exclusive resource, and people on both sides cannot cross the bridge simultaneously.

- Non-preemptive condition

Before the resource got by the thread is used up, the resource applicant cannot forcibly seize the resource from the resource occupant, while can only be released independently by the occupant thread of the resource. If the person passing the single-plank bridge cannot force the person at the opposite side to retreat, or cannot illegally push the person at the opposite off the bridge, the person at the opposite side can pass the bridge after the person on the bridge passes the bridge and the deck is empty (i.e., the resources occupied are actively released).

- Possession and application conditions

The thread has occupied a resource at least, while applies for the new resources; the resource has been occupied by other threads, and the thread is blocked at the moment; However, it still continue to occupy the resources occupied when waiting for the new resources. Take passing the single-plank bridge as an example again, A and B meet on the bridge. A walks through a section of deck (i.e., occupy some resources), and needs to walk the rest of deck) (apply for new resources). However, the part of deck has been occupied by B (B walks through a section of deck). A cannot move forward or backward; B is in the same condition.

- Circular wait conditions

There is a thread wait sequence $\{P_1, P_2, \dots, P_n\}$. Where, P_1 waits a certain resource occupied by P_2 , P_2 waits a certain resource occupied by P_3 ,, while P_n waits a certain resource occupied by P_1 , causing circular wait. Just like the above case of crossing single-wood bridge, A waits for the bridge occupied by B, while B waits for the bridge occupied by B, causing circular wait.

The above four conditions we mentioned will occur simultaneously during deadlock. That is to say, deadlock will not occur as long as a necessary condition is not satisfied.

7.7.3 Deadlock prevention

The four necessary conditions for deadlock are introduced above. As long as any of the four conditions is broken, the deadlock will not occur. This provides possibility to solve the deadlock problem. Generally, the method to solve the deadlock is divided into three kinds of deadlock prevention, avoidance, detection and recovery (note: deadlock detection and recovery is a method).

Deadlock prevention is a strategy to guarantee that the system does not enter the deadlock state. Its basic thought is to require the thread to comply with certain protocols when applying for resources, so as to break one or more of the four necessary conditions causing deadlock, and guarantee that the system does not enter the deadlock state.

- Mutex breaking conditions

That is to say, the thread is allowed to have access to some resources. However, some resources are not allowed to be accessed simultaneously, such as printers, etc. which is determined by the attribute of the resources themselves. Therefore, the approach has no practical value.

- Non-preemptive breaking conditions

That is to say, the thread is allowed to forcibly seize some resources from the occupant. That is to say, when a thread has occupied some resources, and cannot be immediately satisfied when the new resource is applied, it must release all occupied

resources, and reapply in the future. The resources released can be allocated to other threads. This means that the resources occupied by the thread are occupied snugly. It is difficult to implement the method of deadlock prevention, and the system performance will be reduced.

- Possession and application breaking conditions

The resource pre-allocation strategy can be implemented. The thread applies for all necessary resources to the system at a time. If all resources required for a certain thread cannot be satisfied, any resource will not be allocated, and the thread does not run temporarily. Only when the system can satisfy all resource demands of the current thread, all resources applied can be allocated to the thread at a time. The running thread has occupied all necessary resources, and phenomena of simultaneous resource occupation and resource application will not occur. Therefore, deadlock will not occur. However, the strategy also has the following disadvantages:

- ◆ In many cases, a thread is impossible to know all resources required before execution. The thread is dynamic and unpredictable during execution;
- ◆ The resource utilization rate is low. No matter when the allocated resources are used, a thread can be executed only after all necessary resources are occupied. Even if some resources are used once by the thread finally, the thread has occupied them for the duration of its existence, causing long-term occupation without use. This is a great waste of resources apparently;
- ◆ Reduces concurrency of the thread. Because of limited resources and existed waste, the number of threads which can be allocated with all necessary resources is less inevitably.

- Break cycle wait conditions

Implement resource allocation strategy in order. Adopting the strategy, the resources are classified and numbered in advance, and allocated by number, so that the thread will not form the loop when applying for and occupying resources. Request of all threads to resources must be proposed in ascending order of resource numbers. The thread can apply for large resources after small resources are occupied, and the loop will not be generated, so as to prevent deadlock. Compared with the previous strategy, the strategy has greatly improved resource utilization and system throughput. However, the following disadvantages are existed:

- ◆ Request of the thread to resources is restricted, it is difficult to reasonably number all resources in the system, and system overhead is increased.
- ◆ In order to comply with the number application order, the resources not used temporarily shall also be applied in advance, so that the duration when the thread occupies the resources is increased.

SylixOS does not support avoidance, detection and recovery of deadlock, which can only be prevented. Generally, we use circular wait breaking conditions to prevent deadlock, and use timeout wait to dissolve deadlock simultaneously. However, it is required that the application has the perfect timeout error processing mechanism.

7.8 POSIX read-write lock

We present an example of shared resource competition above: two threads shall perform increment operation of the same variable V (initial value of 0) simultaneously. The solution to shared resource competition is to add a lock. The lock is occupied before access to variable V, and released after access. Generally, we can use the binary semaphore with an initial value of TRUE or the counting semaphore or mutex semaphore with an initial value of 1 as the lock.

We now slightly modify the example. There are ten threads (thread A, thread 1... thread 9) and a Variable V. Thread A needs to write Variable V, and thread [1-9] need to read Variable V. Apparently, Variable V has more reader threads than writer threads. In this case, if we continue to use the binary semaphore with an initial value of TRUE or the counting semaphore or mutex semaphore with an initial value of 1 as the lock, other reader threads will be blocked on the lock when a reader thread occupies the lock, apparently causing low read concurrent efficiency of shared resources, because direct read operation by multiple threads to Variable V will not confuse the value of Variable V.

In order to solve the problem of low read concurrent efficiency of the common locking mechanism in the case of "read more and write less" of shared resources, the POSIX standard defines read-write lock and operation thereof, and the read-write lock has three states: read state, write state and unlock state. Provisions for read-write lock: the read-write lock at the read state can lock any read lock again, while the requested write lock will be firstly responded after the read lock is unlocked (SylixOS supports the write lock priority principle); the read-write lock at the write state will not respond to any lock, i.e., any lock request will fail.

The type of POSIX read-write lock is `pthread_rwlock_t`. A variable of `pthread_mutexattr_t` type is defined during use. For example:

```
pthread_rwlock_t rwlock;
```

A POSIX read-write lock can be used after creation by calling the `pthread_rwlock_init` function.

If the thread needs to wait a read-write lock, the `pthread_rwlock_rdlock` or `pthread_rwlock` function is called separately according to use (read or write) for shared resources, and the interrupt service routine cannot call any POSIX read-write lock function. A read-write lock is unlocked by calling the `pthread_rwlock_unlock` function.

A read-write lock shall be deleted by calling the `pthread_rwlock_destroy` function after use (guarantee no use in the future), and SylixOS will recover kernel resources occupied by the read-write lock.

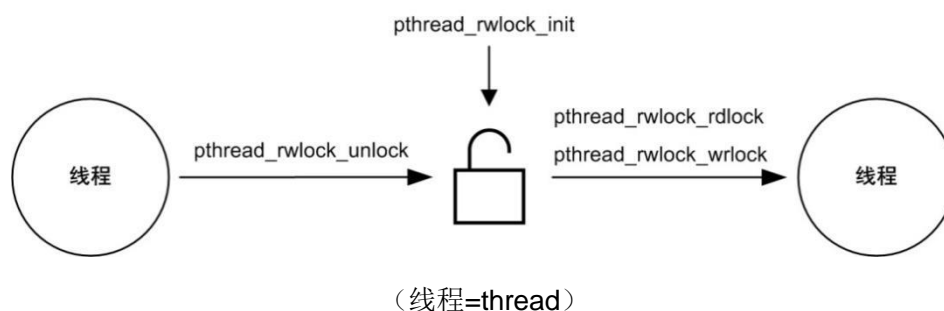


Figure 7.13 POSIX read-write lock

A POSIX read-write lock attribute block shall be used as the parameter when a POSIX read-write lock is created. The type of POSIX read-write lock attribute block is `pthread_rwlockattr_t`. A variable of `pthread_rwlockattr_t` type is defined during use. For example:

```
pthread_rwlockattr_t rwlockattr;
```

7.8.1 Read-write lock attribute block^①

1. Initialization and deletion of the read-write lock attribute block

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *prwlockattr);
```

Prototype analysis of Function `pthread_rwlockattr_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlockattr*** is the pointer of POSIX read-write lock attribute block;

```
#include <pthread.h>
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *prwlockattr);
```

Prototype analysis of Function `pthread_rwlockattr_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlockattr*** is the pointer of POSIX read-write lock attribute block;

2. Set and get process shared attributes of the read-write lock attribute block

```
#include <pthread.h>
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *prwlockattr,
```

```
int pthread_rwlockattr_setpshared(
```

```
pthread_rwlockattr_t *prwlockattr,
```

```
int pshared);
```

Prototype analysis of Function `pthread_rwlockattr_setpshared`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlockattr*** is the pointer of POSIX read-write lock attribute block;
- Parameter ***pshared*** identifies whether the POSIX read-write lock attribute block is shared by processes.

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *prwlockattr,
int pshared);
```

Prototype analysis of Function `pthread_rwlockattr_getpshared`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlockattr*** is the pointer of POSIX read-write lock attribute block;
- Output parameter ***pshared*** identifies whether the POSIX read-write lock attribute block is shared by processes.

7.8.2 Read-write lock

1. Initialization and deletion of the read-write lock

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *prwlock,
const pthread_rwlockattr_t *prwlockattr);
```

Prototype analysis of Function `pthread_rwlock_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlock*** is the pointer of POSIX read-write lock;
- Parameter ***prwlockattr*** is the pointer of POSIX read-write lock attribute object, which can be NULL;

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *prwlock);
```

Prototype analysis of Function `pthread_rwlock_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlock*** is the pointer of POSIX read-write lock.

2. Read wait of read-write lock

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *prwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *prwlock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *prwlock,
                               const struct timespec *abs_timeout);
```

Prototype analysis of above three functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlock*** is the pointer of POSIX read-write lock;
- Parameter ***abs_timeout*** is the waiting absolute timeout.

The `pthread_rwlock_timedrdlock` is the version of the `pthread_rwlock_rdlock` with wait timeout, and the `abs_timeout` is the absolute wait timeout (see Chapter 11 Time Management).

The `pthread_rwlock_tryrdlock` is the "try wait" version of the `pthread_rwlock_rdlock`. When the read-write lock has been occupied by the write lock, the `pthread_rwlock_rdlock` will be blocked until awakened, while the `pthread_rwlock_tryrdlock` will return immediately, and the error number `EBUSY` is returned.

3. Write wait of read-write lock

```
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *prwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *prwlock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *prwlock,
                               const struct timespec *abs_timeout);
```

Prototype analysis of above three functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlock*** is the pointer of POSIX read-write lock;
- Parameter ***abs_timeout*** is the waiting absolute timeout.

The `pthread_rwlock_timedwrlock` is the version of the `pthread_rwlock_wrlock` with wait timeout, and the `abs_timeout` is the absolute wait timeout

The `pthread_rwlock_trywrlock` is the "try wait" version of the `pthread_rwlock_wrlock`. When the read-write lock has been occupied by the read lock, the `pthread_rwlock_wrlock` will be blocked until awakened, while the `pthread_rwlock_trywrlock` will return immediately, and the error number `EBUSY` is returned.

4. Unlock of read-write lock

```
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *prwlock);
```

Prototype analysis of Function `pthread_rwlock_unlock`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***prwlock*** is the pointer of POSIX read-write lock.

The following program shows how to use POSIX read-write lock, the program creates four reader threads, a writer semaphore and a POSIX read-write lock, the writer thread performs auto increment operation for Variable count, four reader threads perform printing for Variable count, and POSIX read-write lock is used as mutex means of the access variable count.

Program List 7.6 Use of POSIX read-write lock

```
#include <stdio.h>
#include <pthread.h>

#define READ_THREAD_NR    4

static pthread_rwlock_t    lock;
static int                 count = 0;

static void *thread_read (void *arg)
{
    while (1) {
        pthread_rwlock_rdlock(&lock);
        printf("thread_read(): count = %d\n", count);
        pthread_rwlock_unlock(&lock);
        sleep(1);
    }
    return (NULL);
}

static void *thread_write (void *arg)
{
    while (1) {
        pthread_rwlock_wrlock(&lock);
        count++;
        pthread_rwlock_unlock(&lock);
        sleep(1);
    }
    return (NULL);
}

int main (int argc, char *argv[])
```

```
{
pthread_t      threadrd_tid[READ_THREAD_NR];
pthread_t      threadwr_tid;
int            ret;
int            i;

ret = pthread_rwlock_init(&lock, NULL);
if (ret != 0) {
    fprintf(stderr, "rwlock create failed.\n");
    return (-1);
}

for (i = 0; i < READ_THREAD_NR; i++) {
    ret = pthread_create(&threadrd_tid[i], NULL, thread_read, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }
}

ret = pthread_create(&threadwr_tid, NULL, thread_write, NULL);
if (ret != 0) {
    fprintf(stderr, "pthread create failed.\n");
    return (-1);
}

for (i = 0; i < READ_THREAD_NR; i++) {
    pthread_join(threadrd_tid[i], NULL);
}
pthread_join(threadwr_tid, NULL);
pthread_rwlock_destroy(&lock);

return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./posix_rwlock
thread_read(): count = 0
thread_read(): count = 0
thread_read(): count = 0
thread_read(): count = 0
thread_read(): count = 1
thread_read(): count = 1
```

```
thread_read(): count = 1
thread_read(): count = 1
```

7.9 SylixOS condition variable

We continue to use instance in read-write lock section of POSIX:

There are ten threads (thread A, thread 1... thread 9) and a Variable V. Thread A needs to write Variable V, and thread1...thread 9 need to read Variable V.

We assume that the reader thread needs to read Variable V only when the value of Variable V is changed, and the reader thread needs to be blocked when the value of Variable V is unchanged.

The reader thread may need a "judgment" operation before blocking, so as to judge whether the current value of Variable V is inconsistent with the last read value; "judgment" shall be locked before operation, and the reader thread shall be blocked if consistent. The reader thread needs to release the lock before entering the blocking state, and lock releasing and block need a uninterruptible atomic operation

We can imagine the situation that lock releasing and blocking are not an atomic operation. If the reader thread is occupied by Thread A between lock releasing and blocking. Thread A can successfully get the lock certainly. Thread A writes Variable V, and the value of Variable V is changed, while the reader thread is blocked. Apparently, the reader thread loses a response to change in the value of Variable V!

Meanwhile, multiple threads are "informed" to read the variable via broadcast after Thread A writes Variable V.

We need a new means of inter-thread communication - **condition variable** to solve the above problem - lock releasing and blocking are an atomic operation and can "inform" multiple reader threads via broadcast.

The condition variable is a synchronization mechanism between multiple threads. When the condition variable is used together with the mutex lock, the thread is allowed to wait for conditions to occur without competition. The condition itself is protected by the mutex. Therefore, the mutex must be locked before the thread changes the condition, and other threads will not detect change in conditions before the mutex is got. The following pseudocode process is one possible way to use the condition variable:

```
Define global condition (global_cond)
Defining a global mutex (global_lock)
Global variables (global_value)

t1 ()
{
```

```
    Get a mutex (lock global_lock)
    Wait condition (Wait)
    Release the mutex (unlock global_lock)
}

t2 ()
{
    Get a mutex (lock global_lock)
    Global variable operations
    Notification conditions are met (Signal)
    Release the mutex (unlock global_lock)
}

main ()
{
    Initialize condition (global_cond)
    Create mutex (global_lock)
    Create thread      (t1 t2)
    Join threads (join t1 t2)
    Destroy condition variables      (global_cond)
    Delete mutex      (global_lock)
}
```

The type of SylixOS condition variable is `LW_THREAD_COND`.

A variable of `LW_THREAD_COND` type shall be defined during use. For example:

```
LW_THREAD_COND tcd;
```

A SylixOS condition variable can be used after creation by calling the `Lw_Thread_Cond_Init` function.

If it is required to wait for a condition variable, the `Lw_Thread_Cond_Wait` function can be called. The interrupt service routine cannot call the `Lw_Thread_Cond_Wait` function to wait for a condition variable, because the `Lw_Thread_Cond_Wait` function will block the current thread.

Sending a condition variable can use the `Lw_Thread_Cond_Signal` or `Lw_Thread_Cond_Broadcast` function, and the interrupt service routine can also send a condition variable.

A condition variable shall be deleted by calling the `Lw_Thread_Cond_Destroy` function after use, and SylixOS will recover kernel resources occupied by the condition variable. It might be noted that trying to use a condition variable deleted will generate unknown errors.

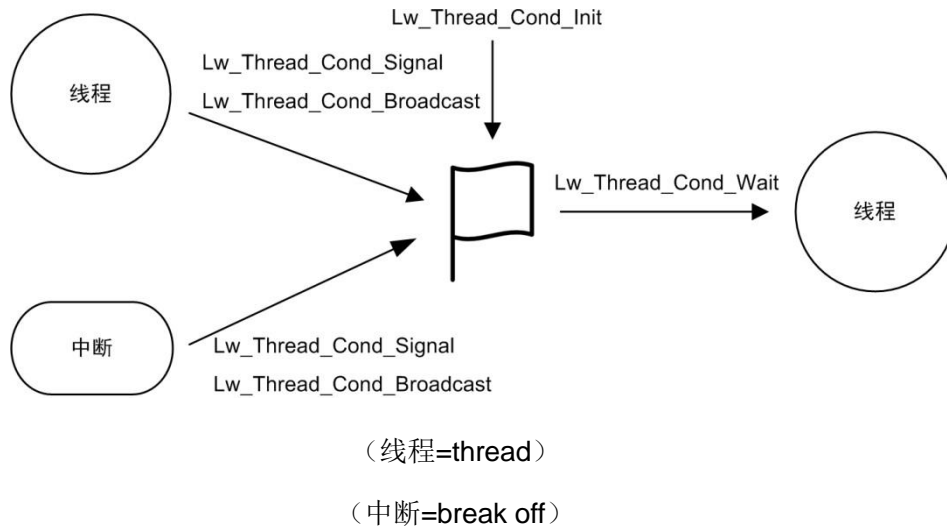


Figure 7.14 SylixOS condition variable

The SylixOS condition variable attribute block shall be used to create a SylixOS condition variable. The type of SylixOS condition variable attribute block is ULONG.

A variable of ULONG type is defined during use. For example:

```
ULONG ulCondAttr;
```

7.9.1 Condition variable attribute block

1. Initialization and deletion of the condition variable attribute block

```
#include <SylixOS.h>
ULONG Lw_Thread_Condattr_Init(ULONG *pulAttr);
```

Prototype analysis of Function Lw_Thread_Condattr_Init:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pulAttr*** is the pointer of SylixOS condition variable attribute block.

```
#include <SylixOS.h>
ULONG Lw_Thread_Condattr_Destroy(ULONG *pulAttr);
```

Prototype analysis of Function Lw_Thread_Condattr_Destroy:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pulAttr*** is the pointer of SylixOS condition variable attribute block.

2. Shared attributes of the progress to set and obtain condition variable attribute block

```
#include <SylixOS.h>
ULONG Lw_Thread_Condattr_Setpshared(ULONG *pulAttr, INT iShared);
```

Prototype analysis of Function Lw_Thread_Condattr_Setpshared:

- For success of the function, return 0. For failure, return the error number;
- Parameter **pulAttr** is the pointer of SylixOS condition variable attribute block;
- Parameter **iShared** identifies whether the SylixOS condition variable attribute block is shared by processes.

```
#include <SylixOS.h>
ULONG Lw_Thread_Condattr_Getpshared(const ULONG *pulAttr, INT *piShared);
```

Prototype analysis of Function Lw_Thread_Condattr_Getpshared:

- For success of the function, return 0. For failure, return the error number;
- Parameter **pulAttr** is the pointer of SylixOS condition variable attribute block;
- Output parameter **iShared** identifies whether the SylixOS condition variable attribute block is shared by processes.

7.9.2 Condition variable

1. Initialization and deletion of the condition variable

```
#include <SylixOS.h>
ULONG Lw_Thread_Cond_Init(PLW_THREAD_COND ptcd, ULONG ulAttr);
```

Prototype analysis of Function Lw_Thread_Cond_Init:

- For success of the function, return 0. For failure, return the error number;
- Parameter **ptcd** is the pointer of SylixOS condition variable;
- Parameter **ulAttr** is SylixOS condition variable attribute block.

```
#include <SylixOS.h>
ULONG Lw_Thread_Cond_Destroy(PLW_THREAD_COND ptcd);
```

Prototype analysis of Function Lw_Thread_Cond_Destroy:

- For success of the function, return 0. For failure, return the error number;
- Parameter **ptcd** is the pointer of SylixOS condition variable.

2. Waiting of condition variable

```
#include <SylixOS.h>
ULONG Lw_Thread_Cond_Wait(PLW_THREAD_COND ptcd,
```

```

LW_HANDLE      ulMutex,
ULONG          ulTimeout);

```

Prototype analysis of Function `Lw_Thread_Cond_Wait`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ptcd*** is the pointer of SylixOS condition variable;
- Parameter ***ulMutex*** is the handle of SylixOS mutex semaphore;
- Parameter ***ulTimeout*** is the waiting timeout, and the unit is Tick.

3. Sending of condition variable

```

#include <SylixOS.h>
ULONG   Lw_Thread_Cond_Signal(PLW_THREAD_COND   ptcd);
ULONG   Lw_Thread_Cond_Broadcast(PLW_THREAD_COND ptcd);

```

Prototype analysis of above two functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***ptcd*** is the pointer of SylixOS condition variable.

The difference between `Lw_Thread_Cond_Broadcast` and `Lw_Thread_Cond_Signal` is that `Lw_Thread_Cond_Broadcast` will awaken all threads blocked at the condition variable via broadcast, while `Lw_Thread_Cond_Signal` will only awaken a thread.

The following program shows how to use the SylixOS condition variable, the program creates two threads, a SylixOS mutex semaphore and a SylixOS condition variable; Thread `tTestA` waits for condition satisfied and prints the value of Variable `_G_iCount`, and Thread `tTestB` performs auto increment operation for Variable `_G_iCount` and sends condition satisfied signal via broadcast. The SylixOS mutex semaphore is used as mutex for access to Variable `_G_iCount`, and the SylixOS condition variable is used to inform change in the value of Variable `_G_iCount`.

Program List 7.7 How to use SylixOS condition variable

```

#include <SylixOS.h>

static INT          _G_iCount = 0;
static LW_HANDLE    _G_hLock;
static LW_THREAD_COND _G_threadCond;

static PVOID tTestA (PVOID pvArg)
{
    INT    iError;

    while (1) {

```

```

        iError = Lw_SemaphoreM_Wait(_G_hLock, LW_OPTION_WAIT_INFINITE);
    if (iError != ERROR_NONE) {
        break;
    }
    iError = Lw_Thread_Cond_Wait(&_G_threadCond,
                                _G_hLock,
                                LW_OPTION_WAIT_INFINITE);

    if (iError != ERROR_NONE) {
        Lw_SemaphoreM_Post(_G_hLock);
        break;
    }

    printf("tTestA(): count = %d\n", _G_iCount);

    Lw_SemaphoreM_Post(_G_hLock);
}
return (LW_NULL);
}

static PVOID tTestB (PVOID pvArg)
{
    INT    iError;

    while (1) {
        iError = Lw_SemaphoreM_Wait(_G_hLock, LW_OPTION_WAIT_INFINITE);
        if (iError != ERROR_NONE) {
            break;
        }

        _G_iCount++;
        Lw_Thread_Cond_Broadcast(&_G_threadCond);

        Lw_SemaphoreM_Post(_G_hLock);

        Lw_Time_SSsleep(1);
    }
    return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_HANDLE        hThreadAId;
    LW_HANDLE        hThreadBId;
    ULONG            ulCondAttr;

```

```
INT                                     iError;

Lw_Thread_Condattr_Init(&ulCondAttr);
Lw_Thread_Condattr_Setpshared(&ulCondAttr, LW_FALSE);

iError = Lw_Thread_Cond_Init(&_G_threadCond, ulCondAttr);
if (iError != ERROR_NONE) {
    printf("cond create failed.\n");
    return (-1);
}

Lw_Thread_Condattr_Destroy(&ulCondAttr);

_G_hLock = Lw_SemaphoreM_Create("count_lock",
                                LW_PRIO_HIGH,
                                LW_OPTION_WAIT_FIFO |
                                LW_OPTION_OBJECT_LOCAL |
                                LW_OPTION_INHERIT_PRIORITY |
                                LW_OPTION_ERRORCHECK,
                                LW_NULL);
if (_G_hLock == LW_OBJECT_HANDLE_INVALID) {
    printf("mutex create failed.\n");
    return (-1);
}

hThreadAId = Lw_Thread_Create("t_testa", tTestA, LW_NULL, LW_NULL);
if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
    printf("t_testa create failed.\n");
    return (-1);
}

hThreadBId = Lw_Thread_Create("t_testb", tTestB, LW_NULL, LW_NULL);
if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
    printf("t_testb create failed.\n");
    return (-1);
}

Lw_Thread_Join(hThreadAId, LW_NULL);
Lw_Thread_Join(hThreadBId, LW_NULL);

Lw_Thread_Cond_Destroy(&_G_threadCond);
Lw_SemaphoreM_Delete(&_G_hLock);
```

```

return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./ThreadCond
tTestA(): count = 1
tTestA(): count = 2
tTestA(): count = 3
tTestA(): count = 4

```

7.10 POSIX condition variable

The type of POSIX condition variable is `pthread_cond_t`. A variable of `pthread_cond_t` type shall be defined during use. For example:

```
pthread_cond_t cond;
```

POSIX condition variable need to be used in combination with POSIX mutex semaphore. and we need to firstly create a POSIX mutex semaphore for sharing resource lock before POSIX condition variable is used.

A POSIX condition variable can be used after creation by calling the `pthread_cond_init` function.

If the thread needs to wait for a condition variable, the `pthread_cond_wait` function can be called. The interrupt service routine cannot call the `pthread_cond_wait` function to wait for a condition variable, because the `pthread_cond_wait` function will block the current thread.

Sending a condition variable can use the `pthread_cond_signal` or `pthread_cond_broadcast` function, and the interrupt service routine cannot send a POSIX condition variable.

A condition variable shall be deleted by calling the `pthread_cond_destroy` function after use (guarantee no use in the future), and SylixOS will recover kernel resources occupied by the condition variable.

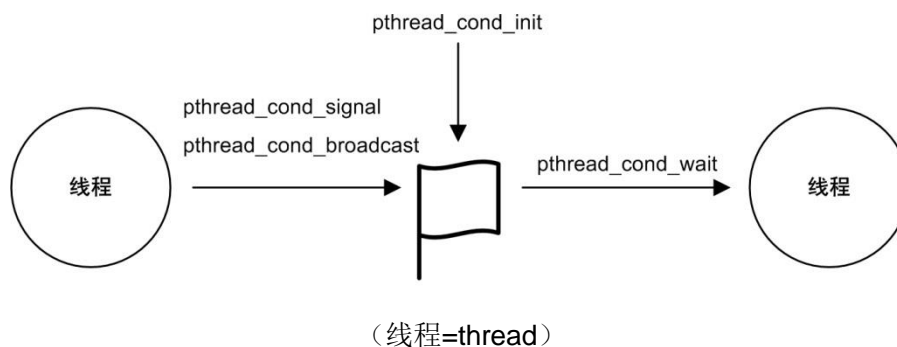


Figure 7.15 POSIX condition variable

The POSIX condition variable attribute block shall be used to create a POSIX condition variable. The type of POSIX condition variable is `pthread_condattr_t`. A variable of `pthread_condattr_t` type shall be defined during use. For example:

```
pthread_condattr_t    condattr;
```

7.10.1 Condition variable attribute block

1. Initialization and deletion of the condition variable attribute block

```
#include <pthread.h>
int  pthread_condattr_init(pthread_condattr_t  *pcondattr);
```

Prototype analysis of Function `pthread_condattr_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcondattr*** is the pointer of POSIX condition variable attribute block.

```
#include <pthread.h>
int  pthread_condattr_destroy(pthread_condattr_t  *pcondattr);
```

Prototype analysis of Function `pthread_condattr_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcondattr*** is the pointer of POSIX condition variable attribute block.

2. Shared attributes of the progress to set and obtain condition variable attribute block

```
#include <pthread.h>
int  pthread_condattr_setpshared(pthread_condattr_t  *pcondattr,
                                int                  ishare);
```

Prototype analysis of Function `pthread_condattr_setpshared`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcondattr*** is the pointer of POSIX condition variable attribute block;
- Parameter ***ishare*** identifies whether the POSIX condition variable attribute block is shared by processes.

```
#include <pthread.h>
int  pthread_condattr_getpshared(const pthread_condattr_t  *pcondattr,
                                int                  *pishare);
```

Prototype analysis of Function `pthread_condattr_getpshared`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcndattr*** is the pointer of POSIX condition variable attribute block;
- Output parameter ***pishare*** identifies whether the POSIX condition variable attribute block is shared by processes.

3. Set and get clock type of the condition variable attribute block

```
#include <pthread.h>
int  pthread_condattr_setclock(pthread_condattr_t *pcndattr,
                               clockid_t         clock_id);
```

Prototype analysis of Function `pthread_condattr_setclock`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcndattr*** is the pointer of POSIX condition variable attribute block;
- Parameter ***clock_id*** is the clock type.

Note: The current parameter ***clock_id*** can only use Macro `CLOCK_REALTIME` (see Chapter 11 Time Management).

```
#include <pthread.h>
int  pthread_condattr_getclock(const pthread_condattr_t *pcndattr,
                              clockid_t                 pclock_id);
```

Prototype analysis of Function `pthread_condattr_getclock`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcndattr*** is the pointer of POSIX condition variable attribute block;
- Output parameter ***clock_id*** is the clock type.

7.10.2 Condition variable

1. Initialization and deletion of the condition variable

```
#include <pthread.h>
int  pthread_cond_init(pthread_cond_t          *pcnd,
                      const pthread_condattr_t pcndattr);
```

Prototype analysis of Function `pthread_cond_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcnd*** is the pointer of POSIX condition variable;

- Parameter ***pcondattr*** is the pointer of POSIX condition variable attribute block, which can be NULL.

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *pcond);
```

Prototype analysis of Function `pthread_cond_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcond*** is the pointer of POSIX condition variable.

2. Sending of condition variable

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *pcond);
int pthread_cond_broadcast(pthread_cond_t *pcond);
```

Prototype analysis of above two functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcond*** is the pointer of POSIX condition variable.

The difference between `pthread_cond_broadcast` and `pthread_cond_signal` is that `pthread_cond_broadcast` will awaken all threads blocked at the condition variable via broadcast, while `pthread_cond_signal` will only awaken a thread.

3. Waiting of condition variable

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *pcond,
                    pthread_mutex_t *pmutex);
int pthread_cond_timedwait(pthread_cond_t *pcond,
                          pthread_mutex_t *pmutex,
                          const struct timespec *abs_timeout);
int pthread_cond_reltimedwait_np(pthread_cond_t *pcond,
                                 pthread_mutex_t *pmutex,
                                 const struct timespec *rel_timeout);
```

Prototype analysis of above three functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pcond*** is the pointer of POSIX condition variable;
- Parameter ***pmutex*** is the pointer of POSIX mutex semaphore;
- Parameter ***abs_timeout*** is the waiting absolute timeout;
- Parameter ***rel_timeout*** is the waiting relative timeout.

The `pthread_cond_timedwait` is the version of the `pthread_cond_wait` with wait timeout, and the ***abs_timeout*** is the absolute wait timeout.

The `pthread_cond_reltimedwait_np` is the non-POSIX standard version of the `pthread_cond_timedwait`, and Parameter `rel_timeout` is relative wait timeout.

The following program shows how to use POSIX condition variable, the program creates two threads, a POSIX mutex semaphore and a POSIX condition variable, Thread `thread_a` prints Variable count, and Thread `thread_b` performs auto increment operation for Variable count. POSIX mutex semaphore is used as mutex for access to Variable count, and POSIX condition variable is used to inform change in the value of Variable count.

Program List 7.8 Use of POSIX condition variable

```
#include <stdio.h>
#include <pthread.h>

static pthread_mutex_t lock;
static pthread_cond_t cond;
static int count = 0;

static void *thread_a (void *arg)
{
    while (1) {
        pthread_mutex_lock(&lock);

        pthread_cond_wait(&cond, &lock);

        printf("thread_a(): count = %d\n", count);

        pthread_mutex_unlock(&lock);
    }
    return (NULL);
}

static void *thread_b (void *arg)
{
    while (1) {
        pthread_mutex_lock(&lock);

        count++;

        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&lock);
    }
}
```

```
        sleep(1);
    }
    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t      threada_tid;
    pthread_t      threadb_tid;
    int            ret;

    ret = pthread_mutex_init(&lock, NULL);
    if (ret != 0) {
        fprintf(stderr, "mutex create failed.\n");
        return (-1);
    }

    ret = pthread_cond_init(&cond, NULL);
    if (ret != 0) {
        fprintf(stderr, "cond create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threada_tid, NULL, thread_a, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threadb_tid, NULL, thread_b, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    pthread_join(threada_tid, NULL);
    pthread_join(threadb_tid, NULL);

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&lock);

    return (0);
}
```

```
}
```

Run the program under the SylixOS Shell:

```
# ./posix_cond
thread_a(): count = 1
thread_a(): count = 2
thread_a(): count = 3
thread_a(): count = 4
thread_a(): count = 5
```

7.11 SylixOS message queue

Before instructions for the message queue, let's have a look at the instance: there are two threads (Thread A and Thread B, Thread A is higher than Thread B in priority) and a variable V. Thread A needs to write Variable V, and Thread B needs to read Variable V.

We assume that Thread B needs to read Variable V only when the value of Variable V is changed, and Thread B needs to be blocked when the value of Variable V is unchanged.

If we continue to use the condition variable to perform inter-thread communication, when Thread A rapidly and frequently modifies the value of Variable V, Thread B may lose a part of response to change in the value of Variable V — the old value to be read has been overwritten by the new value.

The message queue is an FIFO queue which can store multiple messages. If we use the message queue as communication means between Threads A and B, Thread A stores the modified value of Variable V as a message to the message queue, Thread B only needs to read the message from the message queue (i.e., the modified value of Variable V). then Thread B will not loss a part of response to change in the value of Variable V before the message queue is full.

The message queue makes it easier for our software to be divided and implemented by function modules, and different function modules are implemented by using different threads. The message queue is used to perform communication decoupling between function modules instead of definition of calling interface.

For example, the ADC^① thread reads results of ADC after conversion and deposits them in the message queue, and the UI thread takes out results from the message queue and displays it on the screen; the key thread reads the key pressed by the user and stores the key value in the message queue, and the UI thread takes out the key value to switch the display page.... Therefore, only the UI thread can operate the display interface, so as to avoid display error.

The SylixOS message queue also supports sending of emergency messages, which are inserted directly at the head of the message queue. Emergency messages will be processed firstly, so as to guarantee security in case of some anomalies.

A SylixOS message queue can be used after creation by calling the `Lw_MsgQueue_Create` function, and the `Lw_MsgQueue_Create` function will return a handle of message queue after successful creation.

If the thread needs to receive message, the `Lw_MsgQueue_Receive` function can be called. The interrupt service routine cannot call the `Lw_MsgQueue_Receive` function to receive message, because the `Lw_MsgQueue_Receive` function will block the current thread when the message queue is empty.

The interrupt service routine can use the `Lw_MsgQueue_TryReceive` function to receive message, the `Lw_MsgQueue_TryReceive` function will immediately return when the message queue is empty, and the current thread will not be blocked.

The message can be sent by calling the `Lw_MsgQueue_Send` function.

A message queue shall be deleted by calling the `Lw_MsgQueue_Delete` function after use (guarantee no use in the future), and SylixOS will recover kernel resources occupied by the message queue.

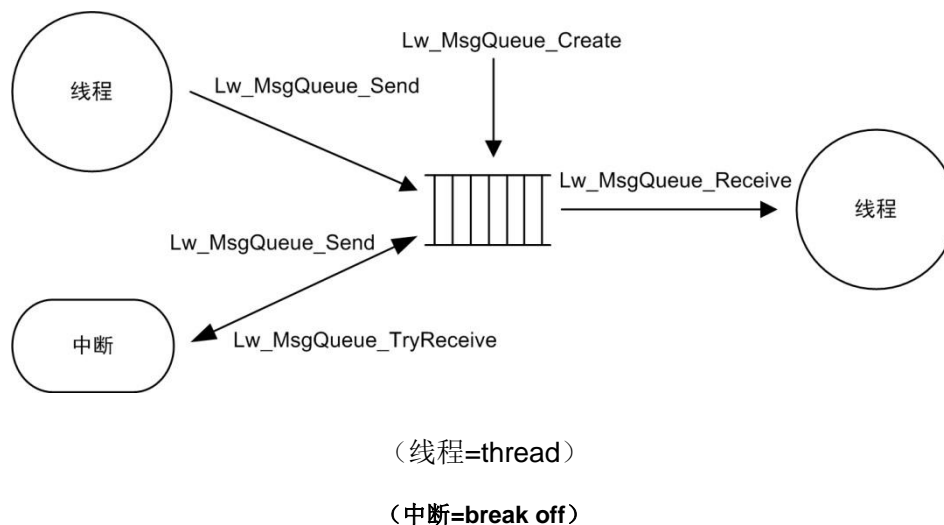


Figure 7.16 SylixOS message queue

7.11.1 Message queue

1. Creation and deletion of the message queue

```
#include <SylixOS.h>
LW_HANDLE Lw_MsgQueue_Create(CPCHAR          pcName,
                             ULONG           uLMaxMsgCounter,
```

```

                                size_t      stMaxMsgByteSize,
                                ULONG        ulOption,
                                LW_OBJECT_ID *pulId ;

```

Prototype analysis of Function Lw_MsgQueue_Create:

- For function success, return the handle of the message queue. For failure, return LW_HANDLE_INVALID, and set the error number;
- Parameter *pcName* is the message queue name;
- Parameter *ulMaxMsgCounter* is the maximum number of messages which can be contained in the message queue;
- Parameter *stMaxMsgByteSize* is the maximum length of single message in the message queue;
- Parameter *ulOption* is the creation option of message queue, as shown in Table 7.2;
- Output parameter *pulId* is used to receive the message queue ID.

It might be noted that the minimum value of the largest message queue in the message queue is sizeof(size_t), i.e., the minimum capacity to create the message queue is sizeof(size_t) bytes.

```

#include <SylixOS.h>
ULONG   Lw_MsgQueue_Delete(LW_HANDLE  *pulId) ;

```

Prototype analysis of Function Lw_MsgQueue_Delete:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *pulId* is the handle of message queue.

2. Receive message

```

#include <SylixOS.h>
ULONG   Lw_MsgQueue_Receive(LW_HANDLE  ulId,
                             PVOID     pvMsgBuffer,
                             size_t    stMaxByteSize,
                             size_t    *pstMsgLen,
                             ULONG     ulTimeout) ;
ULONG   Lw_MsgQueue_ReceiveEx(LW_HANDLE ulId,
                              PVOID    pvMsgBuffer,
                              size_t   stMaxByteSize,
                              size_t   *pstMsgLen,
                              ULONG    ulTimeout,
                              ULONG    ulOption) ;

```

```

ULONG    Lw_MsgQueue_TryReceive(LW_HANDLE    uId,
                                PVOID        pvMsgBuffer,
                                size_t       stMaxByteSize,
                                size_t       *pstMsgLen);

```

Prototype analysis of above three functions:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *uId* is the handle of message queue;
- Parameter *pvMsgBuffer* points at the message buffer zone used to send message (a pointer of void type, can point at any type);
- Parameter *stMaxByteSize* is the length of the message buffer zone;
- Output parameter *pstMsgLen* is used to receive the message length;
- Parameter *ulTimeout* is the waiting timeout, and the unit is Tick.
- Parameter *ulOption* is the receive option of message queue, as shown in Table 7.8.

Table 7.8 Receive option of message queue

Macro name	Meaning
LW_OPTION_NOERROR	The message larger than the buffer zone is automatically truncated (default for the option)

Calling the Lw_MsgQueue_Receive function will get the message from the message queue represented by *uId*:

- When there is a message in the queue, the function will copy the message to the message buffer zone to the direction of Parameter *pvMsgBuffer*. If the buffer is larger than the message in length, the rest of the buffer is not modified; if the buffer zone is less than the message in length, the message will be truncated without any error returned. The Lw_MsgQueue_ReceiveEx function provides the message error checking mechanism, and will return the error number E2BIG when the message is truncated;
- When there is no message in the queue, the thread will be blocked. If the timeout value of *ulTimeout* is set as LW_OPTION_WAIT_INFINITE, the thread will be blocked forever until the message arrives; if the timeout value of *ulTimeout* is not LW_OPTION_WAIT_INFINITE, the thread automatically awakes the thread after the appointed timeout.

3. Send message

```
#include <SylixOS.h>
ULONG   Lw_MsgQueue_Send(LW_HANDLE      uId,
                        const PVOID     pvMsgBuffer,
                        size_t          stMsgLen);
ULONG   Lw_MsgQueue_SendEx(LW_HANDLE    uId,
                           const PVOID  pvMsgBuffer,
                           size_t       stMsgLen,
                           ULONG        uOption);
```

Prototype analysis of above two functions:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *uId* is the handle of message queue;
- Parameter *pvMsgBuffer* points at the buffer zone of message to be sent (a pointer of void type can point at any type);
- Parameter *stMsgLen* is the length of the message to be sent;
- Parameter *uOption* is the message sending option, as shown in Table 7.9.

Table 7.9 Send option of message queue

Macro name	Meaning
LW_OPTION_DEFAULT	Default options
LW_OPTION_URGENT	Emergency message sending
LW_OPTION_BROADCAST	Send broadcast

If the LW_OPTION_URGENT option is used, the message will be inserted at the head of the message queue. If the LW_OPTION_BROADCAST option is used, the message will b

4. Message sending with time delay

```
#include <SylixOS.h>
ULONG   Lw_MsgQueue_Send2(LW_HANDLE      uId,
                          const PVOID     pvMsgBuffer,
                          size_t          stMsgLen,
                          ULONG           uTimeout);
ULONG   Lw_MsgQueue_SendEx2(LW_HANDLE    uId,
                             const PVOID  pvMsgBuffer,
                             size_t       stMsgLen,
```



```

        ULONG          ulTimeout
        ULONG          ulOption);

```

Prototype analysis of above two functions:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulld* is the handle of message queue;
- Parameter *pvMsgBuffer* points at the buffer zone of message to be sent (a pointer of void type can point at any type);
- Parameter *stMsgLen* is the length of the message to be sent;
- Parameter *ulTimeout* is the delay waiting time to send message;
- Parameter *ulOption* is the message sending option, as shown in Table 7.9.

Parameter *ulTimeout* is added in passing of different parameters between the above two functions and the Lw_MsgQueue_Send function, which indicates that the message is sent with the delay waiting function. It means that message sending will wait for the *ulTimeout* time when the message queue sent is full. If the message queue is still at full state when the timeout is up, the message will be abandoned. Otherwise, the message will be sent successfully.

5. Clear message queue

```

#include <SylixOS.h>
ULONG   Lw_MsgQueue_Clear(LW_HANDLE ulld);

```

Prototype analysis of Function Lw_MsgQueue_Clear:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulld* is the handle of message queue.

Message queue clearing means that all messages in the queue will be deleted (the queue is still valid when the message is abandoned), and the expected results will not be got when trying to receive the message from it.

6. Release all threads in the message waiting queue

```

#include <SylixOS.h>
ULONG   Lw_MsgQueue_Flush(LW_HANDLE ulld,
                          ULONG      *pulThreadUnblockNum);

```

Prototype analysis of Function Lw_MsgQueue_Flush:

- For success of the function, return ERROR_NONE. For failure, return the error number;

- Parameter ***uId*** is the handle of message queue;
- Output parameter ***pulThreadUnblockNum*** returns the number of unblocked threads, which can be NULL.

Calling the `Lw_MsgQueue_Flush` function will make all threads (including send and receive threads) blocked in the appointed message queue ready, so as to avoid the thread from long-term blocking.

```
#include <SylixOS.h>
ULONG    Lw_MsgQueue_FlushSend(LW_HANDLE    uId,
                               ULONG        *pulThreadUnblockNum);
```

Prototype analysis of Function `Lw_MsgQueue_FlushSend`:

- For success of the function, return `ERROR_NONE`. For failure, return the error number;
- Parameter ***uId*** is the handle of message queue;
- Output parameter ***pulThreadUnblockNum*** returns the number of unblocked threads, which can be NULL.

Calling the `Lw_MsgQueue_FlushSend` function will make all sending threads blocked in the appointed message queue ready, so as to avoid the sending thread from long-term blocking due to not sending out the message for a long term.

```
#include <SylixOS.h>
ULONG    Lw_MsgQueue_FlushReceive(LW_HANDLE uId,
                                   ULONG     *pulThreadUnblockNum);
```

Prototype analysis of Function `Lw_MsgQueue_FlushReceive`:

- For success of the function, return `ERROR_NONE`. For failure, return the error number;
- Parameter ***uId*** is the handle of message queue;
- Output parameter ***pulThreadUnblockNum*** returns the number of unblocked threads, which can be NULL.

Calling the `Lw_MsgQueue_FlushReceive` function will make all receiving threads blocked in the appointed message queue ready, so as to avoid the receiving thread from long-term blocking due to not receiving the message for a long term.

7. Get the state of the message queue

```
#include <SylixOS.h>
ULONG    Lw_MsgQueue_Status(LW_HANDLE    uId,
                             ULONG        *pulMaxMsgNum,
                             ULONG        *pulCounter,
```

```

                                size_t      *pstMsgLen,
                                ULONG        *pulOption,
                                ULONG        *pulThreadBlockNum) ;
ULONG  Lw_MsgQueue_StatusEx(LW_HANDLE  ulId,
                                ULONG    *pulMaxMsgNum,
                                ULONG    *pulCounter,
                                size_t   *pstMsgLen,
                                ULONG    *pulOption,
                                ULONG    *pulThreadBlockNum,
                                size_t   *pstMaxMsgLen) ;

```

Prototype analysis of above functions:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulId* is the handle of message queue;
- Output parameter *pulMaxMsgNum* is used to receive the maximum number of messages which can be contained in the message queue;
- Output parameter *pulCounter* is used to receive the number of current messages in the message queue.
- Output parameter *pstMsgLen* is used to receive the length of the recent message in the message queue;
- Output parameter *pulOption* is used to receive creation option of the message queue;
- Output parameter *pulThreadBlockNum* is used to receive the number of threads currently blocked in the message queue;
- Output parameter *pstMaxMsgLen* is used to receive the maximum length of single message in the message queue.

8. Get the name of the message queue

```

#include <SylixOS.h>
ULONG  Lw_MsgQueue_GetName(LW_HANDLE  ulId,
                            PCHAR     pcName) ;

```

Prototype analysis of Function Lw_MsgQueue_GetName:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulId* is the handle of message queue;

Output parameter

● **pcName** is the name of the counting semaphore, and **pcName** shall be pointed at a character array with size of LW_CFG_OBJECT_NAME_SIZE.

The following program shows how to use the SylixOS message queue. The program creates two threads and a SylixOS message queue; Thread tTestB sends the character string as a message to the message queue, and Thread tTestA reads and prints the message from the message queue.

Program List 7.9 How to use SylixOS message queue

```
#include <SylixOS.h>
#include "string.h"

static LW_HANDLE    _G_hMsgQ;

static PVOID tTestA (PVOID pvArg)
{
    INT        iError;
    CHAR       acMsg[64];
    size_t     stLen;

    while (1) {
        iError = Lw_MsgQueue_Receive(_G_hMsgQ, acMsg, sizeof(acMsg), &stLen,
                                     LW_OPTION_WAIT_INFINITE);

        if (iError != ERROR_NONE) {
            break;
        }

        printf("tTestA(): get a msg \"%s\"\n", acMsg);
    }

    return (LW_NULL);
}

static PVOID tTestB (PVOID pvArg)
{
    INT        iError;
    CHAR       acMsg[64];
    size_t     stLen;
    INT        iCount = 0;

    while (1) {
        sprintf(acMsg, "hello SylixOS %d", iCount);
        stLen = strlen(acMsg) + 1;
    }
}
```

```
        iCount++;

        iError = Lw_MsgQueue_Send(_G_hMsgQ, acMsg, stLen);
        if (iError != ERROR_NONE) {
            break;
        }

        Lw_Time_SSsleep(1);
    }
    return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_HANDLE          hThreadAId;
    LW_HANDLE          hThreadBId;

    _G_hMsgQ = Lw_MsgQueue_Create("msgq", 10, 64,
                                  LW_OPTION_WAIT_FIFO |
                                  LW_OPTION_OBJECT_LOCAL,
                                  LW_NULL);

    if (_G_hMsgQ == LW_OBJECT_HANDLE_INVALID) {
        printf("message queue create failed.\n");
        return (-1);
    }

    hThreadAId = Lw_Thread_Create("t_testa", tTestA, LW_NULL, LW_NULL);
    if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testa create failed.\n");
        return (-1);
    }

    hThreadBId = Lw_Thread_Create("t_testb", tTestB, LW_NULL, LW_NULL);
    if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testb create failed.\n");
        return (-1);
    }

    Lw_Thread_Join(hThreadAId, LW_NULL);
    Lw_Thread_Join(hThreadBId, LW_NULL);

    Lw_MsgQueue_Delete(&_G_hMsgQ);
}
```

```
    return (0);  
}
```

Run the program under the SylixOS Shell:

```
# ./MsgQueue  
tTestA(): get a msg "hello SylixOS 0"  
tTestA(): get a msg "hello SylixOS 1"  
tTestA(): get a msg "hello SylixOS 2"  
tTestA(): get a msg "hello SylixOS 3"
```

7.12 SylixOS event set

We generally have used the P2P software (such as BT, eMule and so on) to download movies and so on. P2P software divides the files to be downloaded into many small pieces, downloads these different file fragments from multiple file sources, and assembles them into one file after all file fragments have been downloaded:

The P2P software needs to record the download status of these file fragments and implement the online playback function, and the event set provided by SylixOS solves these problems very well via inter-thread communication.

The event set is defined as the ULONG type, and each bit represents an event. For the above instance, each bit represents a file fragment. Therefore, the problem to record the download status of file fragments is solved very well.

The online play function relies on current file fragments to be played. If current file fragments to be played is not downloaded, play can only be paused. Download and play are implemented with different threads. The download thread cannot simply download a file fragment to awaken the play thread for play, because it is uncertain which file fragments can be downloaded, and the file fragments downloaded currently may not be required for the play thread currently. The event set provides API for sending and waiting for events, which solves the problem of synchronization of download and play threads very well.

Here is the instance of P2P software to indicate the purpose of the event set, and the function of the event set is not limited to this actually.

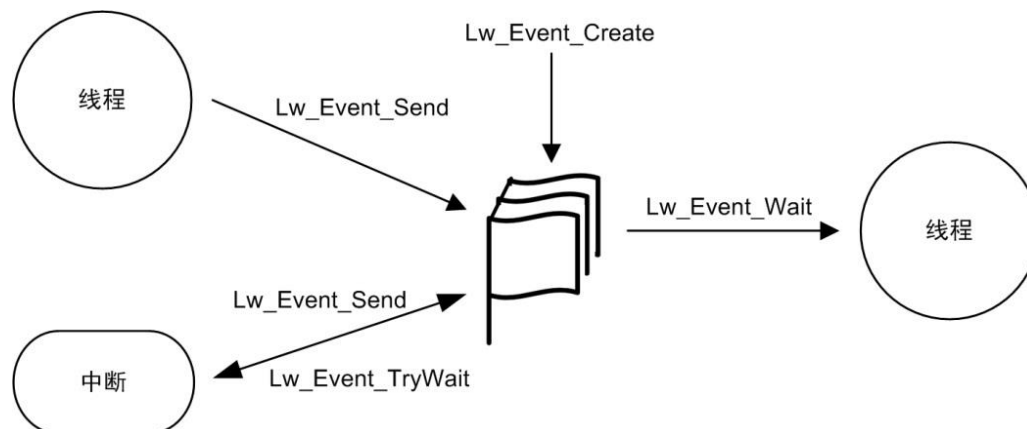
A SylixOS event set can be used after creation by calling the Lw_Event_Create function, and the Lw_Event_Create function will return a handle of event set after successful creation.

If it is required to wait for the event, the Lw_Event_Wait function can be called. The interrupt service routine cannot call the Lw_Event_Wait function to wait for the event, because the Lw_Event_Wai function will block the current thread when the event is invalid

The interrupt service routine can use the `Lw_Event_TryWait` function to try to wait for the event, because the `Lw_Event_TryWait` function will immediately return when the event are invalid, and the current thread will not be blocked.

The event can be sent by calling the `Lw_Event_Send` function.

An event set shall be deleted by calling the `Lw_Event_Delete` function after use (guarantee no use in the future), and SylixOS will recover kernel resources occupied by the event set.



(线程=thread)

(中断=break off)

Figure 7.17 SylixOS event set

7.12.1 Event set

1. Creation and deletion of the event set

```
#include <SylixOS.h>
LW_HANDLE Lw_Event_Create(CPCHAR      pcName,
                          ULONG       ulInitEvent,
                          ULONG       ulOption,
                          LW_OBJECT_ID *pulId);
```

Prototype analysis of Function `Lw_Event_Create`

- For function success, return handle of the event set. For failure, return `LW_HANDLE_INVALID`, and set the error number;
- Parameter *pcName* is the name of the event set;
- Parameter *ulInitEvent* is the initial value of the event set;
- Parameter *ulOption* is creation option of the event set;

- Output parameter ***pullId*** is used to receive the event set ID.

Combination of Macro LW_OPTION_EVENT_n (n ranges from 0 to 31) can be used for the event, while Macro LW_OPTION_EVENT_ALL can be used for all events.

```
#include <SylixOS.h>
ULONG Lw_Event_Delete(LW_HANDLE *pullId);
```

Prototype analysis of Function Lw_Event_Delete:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter ***pullId*** is the handle of receive event set.

2. Send event set

```
#include <SylixOS.h>
ULONG Lw_Event_Send(LW_HANDLE ulId,
                   ULONG ulEvent,
                   ULONG ulOption);
```

Prototype analysis of Function Lw_Event_Send:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter ***ulId*** is the handle of event set.
- Parameter ***ulEvent*** is the event to be sent;
- Parameter ***ulOption*** is the message send option, as shown in Table 7.10.

Table 7.10 Sent option of the event set

Macro name	Meaning
LW_OPTION_EVENTSET_SET	Set the designated event as 1
LW_OPTION_EVENTSET_CLR	Set the designated event as 0

3. Wait of the event set

```
#include <SylixOS.h>
ULONG Lw_Event_Wait(LW_HANDLE ulId,
                   ULONG ulEvent,
                   ULONG ulOption,
                   ULONG ulTimeout);
ULONG Lw_Event_WaitEx(LW_HANDLE ulId,
                     ULONG ulEvent,
```



```

        ULONG          ulOption,
        ULONG          ulTimeout,
        ULONG          *pulEvent);
ULONG   Lw_Event_TryWait(LW_HANDLE ulId,
        ULONG          ulEvent,
        ULONG          ulOption);
ULONG   Lw_Event_TryWaitEx(LW_HANDLE ulId,
        ULONG          ulEvent,
        ULONG          ulOption,
        ULONG          *pulEvent);

```

Prototype analysis of above functions:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *ulId* is the handle of event set.
- Parameter *ulEvent* is the event to be waited;
- Parameter *ulOption* is the wait event option, as shown in Table 7.11.
- Parameter *ulTimeout* is the waiting timeout, and the unit is Tick.
- Output parameter *pulEvent* identifies the events received.

The difference between Lw_Event_TryWait and Lw_Event_Wait is that if the current event to be waited is invalid, Lw_Event_TryWait will immediately exit and return ERROR_THREAD_WAIT_TIMEOUT, while Lw_Event_Wait will be blocked until awakened.

Table 7.11 Wait option of the event set

Macro name	Meaning
LW_OPTION_EVENTSET_WAIT_CLR_ALL	Activated when the designated event is 0
LW_OPTION_EVENTSET_WAIT_CLR_ANY	Activated when any bit of the designated event is 0
LW_OPTION_EVENTSET_WAIT_SET_ALL	Activated when the designated event is 1
LW_OPTION_EVENTSET_WAIT_SET_ANY	Activated when any bit of the designated event is 1
LW_OPTION_EVENTSET_RETURN_ALL	Return all valid events after getting the event
LW_OPTION_EVENTSET_RESET	Automatically clear the event after getting the event

LW_OPTION_EVENTSET_RESET_ALL	Clear all events after getting the event
------------------------------	------------------------------------------

4. Get the state of the event set

```
#include <SylixOS.h>
ULONG    Lw_Event_Status(LW_HANDLE  uId,
                        ULONG        *pulEvent,
                        ULONG        *pulOption);
```

Prototype analysis of Function Lw_Event_Status:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *uId* is the handle of event set.
- Output parameter *pulEvent* identifies the event at the current position;
- Output parameter *pulOption* is creation option of the event set;

5. Get the name of the event set

```
#include <SylixOS.h>
ULONG    Lw_Event_GetName(LW_HANDLE  uId,
                          PCHAR      pcName);
```

Prototype analysis of Function Lw_Event_GetName:

- For success of the function, return ERROR_NONE. For failure, return the error number;
- Parameter *uId* is the handle of event set.
- Output parameter *pcName* is the name of the event set, and *pcName* shall be pointed at a character array with size of LW_CFG_OBJECT_NAME_SIZE.

The following program shows how to use the SylixOS event set. The program creates two threads and a SylixOS event set; Thread tTestB constantly sends events 0 to 31, and Thread tTestA waits for the event and prints the event number.

Program List 7.10 How to use SylixOS event set

```
#include <SylixOS.h>

static LW_HANDLE  _G_hEventSet;

static PVOID tTestA (PVOID  pvArg)
```

```
{
    INT      iError;
    ULONG    ulEvent;
    INT      i;

    while (1) {
        iError = Lw_Event_WaitEx(_G_hEventSet,
                                LW_OPTION_EVENT_ALL,
                                LW_OPTION_EVENTSET_WAIT_SET_ANY |
                                LW_OPTION_EVENTSET_RESET,
                                LW_OPTION_WAIT_INFINITE,
                                &ulEvent);

        if (iError != ERROR_NONE) {
            break;
        }

        for (i = 0; i < 32; i++) {
            if (ulEvent & (1 << i)) {
                printf("tTestA(): get event %d\n", i);
            }
        }
    }
    return (LW_NULL);
}

static PVOID tTestB (PVOID pvArg)
{
    INT      iError;
    INT      i;

    while (1) {
        for (i = 0; i < 32; i++) {
            iError = Lw_Event_Send(_G_hEventSet, (1 << i),
                                   LW_OPTION_EVENTSET_SET);

            if (iError != ERROR_NONE) {
                return (LW_NULL);
            }

            Lw_Time_SSsleep(1);
        }
    }
    return (LW_NULL);
}
```

```
int main (int argc, char *argv[])
{
    LW_HANDLE      hThreadAId;
    LW_HANDLE      hThreadBId;

    _G_hEventSet = Lw_Event_Create("event_set", 0,
                                   LW_OPTION_WAIT_FIFO |
                                   LW_OPTION_OBJECT_LOCAL,
                                   LW_NULL);
    if (_G_hEventSet == LW_OBJECT_HANDLE_INVALID) {
        printf("event set create failed.\n");
        return (-1);
    }

    hThreadAId = Lw_Thread_Create("t_testa", tTestA, LW_NULL, LW_NULL);
    if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testa create failed.\n");
        return (-1);
    }

    hThreadBId = Lw_Thread_Create("t_testb", tTestB, LW_NULL, LW_NULL);
    if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testb create failed.\n");
        return (-1);
    }

    Lw_Thread_Join(hThreadAId, LW_NULL);
    Lw_Thread_Join(hThreadBId, LW_NULL);

    Lw_Event_Delete(&_G_hEventSet);

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./event_test
tTestA(): get event 0
tTestA(): get event 1
tTestA(): get event 2
tTestA(): get event 3
tTestA(): get event 4
```

7.13 POSIX thread barrier

We must have the experience in organizing the team to visit tourist attractions. Assuming that we have five companions, Companion A goes to the ticket office to buy the group ticket, the rest four companions can only wait at the ticket gate (because tickets have not been bought yet). When Companion A buys the group ticket and collects at entrance of the ticket office, we can enter the tourist attraction from the ticket office.

In the field of computer, there are many similar application scenarios: sort the extra large arrays. In order to exert the concurrent performance of the multi-core processor, 10 threads can be used to sort 10 parts of the extra large arrays. The subsequent incorporate operation can be performed after respective sorting of 10 threads is completed. The firstly completed thread will be suspended to wait, and all wait threads can be awakened until all threads are completed.

We can use semaphore, condition variable and other inter-thread communication methods to complete the above application scenario. However, the POSIX standard defines the thread barrier and operation thereof for more elegant and concise implementation.

Thread barrier Barrier is also called as the thread fence, which mainly used to coordinate with multiple threads to concurrently and jointly complete a certain task. A thread barrier object can enable each thread to be blocked, and continue to run until all coordination threads to (combination to complete a certain task) execute to a certain appointed point (like the dam for water storage).

The type of POSIX thread barrier is `pthread_barrier_t`. A variable of `pthread_spinlock_t` type shall be defined during use, such as:

```
pthread_barrier_t barrier;
```

A POSIX thread barrier can be used after creation by calling the `pthread_barrier_init` function.

If it is required to wait for a thread barrier, the `pthread_spin_lock` function can be called and the interrupt service routine cannot call any POSIX spin lock API.

A thread barrier shall be deleted by calling the `pthread_barrier_destroy` function after use (guarantee no use in the future), and SylixOS will recover kernel resources occupied by the thread barrier.

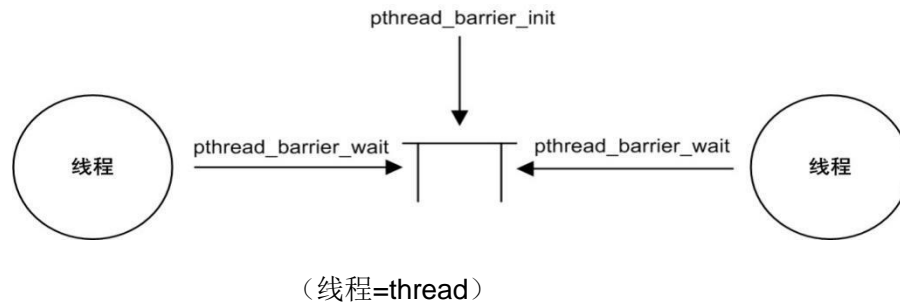


Figure 7.18 POSIX thread barrier

The POSIX thread barrier attribute block shall be used to create a POSIX thread barrier. The type of POSIX thread barrier attribute block is `pthread_barrierattr_t`. A variable of `pthread_barrierattr_t` type shall be defined during use, such as:

```
pthread_barrierattr_t    barrierattr;
```

7.13.1 Thread barrier attribute block

1. Initialization and deletion of the thread barrier attribute block

```
#include <pthread.h>
int  pthread_barrierattr_init(pthread_barrierattr_t    *pbarrierattr);
int  pthread_barrierattr_destroy(pthread_barrierattr_t *pbarrierattr);
```

Prototype analysis of above two functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pbarrierattr*** is the pointer of POSIX thread barrier attribute block.

2. Set and get process shared attributes of the thread barrier attribute block

```
#include <pthread.h>
int  pthread_barrierattr_setpshared(pthread_barrierattr_t *pbarrierattr,
                                   int                    shared);
```

Prototype analysis of Function `pthread_barrierattr_setpshared`:

- The function returns 0;
- Parameter ***pbarrierattr*** is the pointer of POSIX thread barrier attribute block;
- Parameter ***shared*** identifies whether the POSIX thread barrier attribute block is shared by processes.

```
#include <pthread.h>
int  pthread_barrierattr_getpshared(const pthread_barrierattr_t *pbarrierattr,
```

```

int
*pshared);

```

Prototype analysis of Function `pthread_barrierattr_getpshared`:

- The function returns 0;
- Parameter ***pbarrierattr*** is the pointer of POSIX thread barrier attribute block;
- Output parameter ***pshared*** identifies whether the POSIX thread barrier attribute block is shared by processes.

7.13.2 Thread barrier

1. Initialization and deletion of the thread barrier

```

#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *pbarrier,
                        const pthread_barrierattr_t *pbarrierattr,
                        unsigned int count);

```

Prototype analysis of Function `pthread_barrier_init`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pbarrier*** is the pointer of POSIX thread barrier;
- Parameter ***pbarrierattr*** is the pointer of POSIX thread barrier attribute block, which can be NULL;
- Parameter ***count*** identifies how many threads the POSIX thread barrier will barrier.

```

#include <pthread.h>
int pthread_barrier_destroy(pthread_barrier_t *pbarrier);

```

Prototype analysis of Function `pthread_barrier_destroy`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pbarrier*** is the pointer of POSIX thread barrier.

2. Wait of the thread barrier

```

#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *pbarrier);

```

Prototype analysis of Function `pthread_barrier_wait`:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pbarrier*** is the pointer of POSIX thread barrier.

The following program shows how to use the POSIX thread barrier. The program creates four threads and a POSIX thread barrier; four threads delay for different seconds, and then wait for the thread barrier. When all four threads arrive at the thread barrier, the four threads continue to run.

Program List 7.11 Use of POSIX thread barrier

```
#include <stdio.h>
#include <pthread.h>

#define TEST_THREAD_NR    4

static pthread_barrier_t barrier;

static void *thread_test (void *arg)
{
    int sec = (int)arg;

    printf("thread_test(): i will sleep %d second.\n", sec);

    sleep(sec);

    pthread_barrier_wait(&barrier);

    printf("thread_test(): i am here.\n");

    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t          tid[TEST_THREAD_NR];
    pthread_attr_t     attr;
    int                ret;
    int                i;

    ret = pthread_barrier_init(&barrier, NULL, TEST_THREAD_NR);
    if (ret != 0) {
        fprintf(stderr, "barrier create failed.\n");
        return (-1);
    }

    ret = pthread_attr_init(&attr);
    if (ret != 0) {
```



```
        fprintf(stderr, "pthread attr init failed.\n");
    return (-1);
}

for (i = 0; i < TEST_THREAD_NR; i++) {
    ret = pthread_create(&tid[i], &attr, thread_test, (void *) (i + 1));
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }
}

for (i = 0; i < TEST_THREAD_NR; i++) {
    pthread_join(tid[i], NULL);
}

pthread_barrier_destroy(&barrier);

return (0);
}
```

Run the program under the SylixOS Shell, and the results are as follows:

```
# ./posix_barrier
thread_test(): i will sleep 1 second
thread_test(): i will sleep 2 second
thread_test(): i will sleep 3 second
thread_test(): i will sleep 4 second
thread_test(): i am here
thread_test(): i am here
thread_test(): i am here
thread_test(): i am here
```

It can be seen from the program running results that all threads will run to the print position of "i am here" simultaneously after successful creation. Therefore, it can be got that the thread barrier can make different threads run to the same point and continue to run (although the creation time of threads is different).

7.14 POSIX spin lock

Spin lock is a kind of lightweight locking mechanism for protecting shared resources. In fact, the spin lock and the mutex lock are similar, which are designed to solve mutex access to shared resources.

Regardless of mutex lock or spin lock, there can be one owner at most at any time, that is to say, only one thread can acquire lock at most at any time. However, they are very different in the scheduling mechanism. For mutex lock, if the mutex lock has been occupied, the applicant can only enter the dormant state. However, the spin lock will not cause the applicant to sleep. If the spin lock has been occupied by another thread, the applicant has to cyclically judge whether the owner of the spin lock has released the lock. The word "spin" is named after it.

It can be seen therefrom that the spin lock is a lower-level primitive way to protect the data structure or code fragment. This locking mechanism may have the following two problems:

- **Deadlock:** if the applicant tries to recursively acquire the spin lock, deadlock will be caused inevitably;
- **Consume too many CPU resources:** if the application is not successful, the applicant will continue to make cyclic judgment, which undoubtedly reduces the CPU utilization rate.

It can be seen that the spin lock is suitable for relatively short locking time kept by the lock user. The semaphore is suitable for keeping the locking time longer.

It might also be noted that the CPU cannot be abandoned for any reason when the spin-protected critical area code is executed. Therefore, any API which may trigger system task scheduling cannot be called in the area protected by the spin lock.

The type of POSIX spin lock is `pthread_spinlock_t`. Define a variable of `pthread_spinlock_t` type for use, such as:

```
pthread_spinlock_t spin;
```

A POSIX spin lock must be created by calling the `pthread_spin_init` function before use.

If the thread needs to wait for a spinlock, the `pthread_spin_lock` function can be called and the interrupt service routine cannot call any POSIX spin lock API. A spin lock can be released by calling the `pthread_spin_unlock` function.

After a spin lock is used (and it will not be used later), it shall be deleted by calling the `pthread_spin_destroy` function. SylixOS will reclaim the kernel resources occupied by the spin lock.

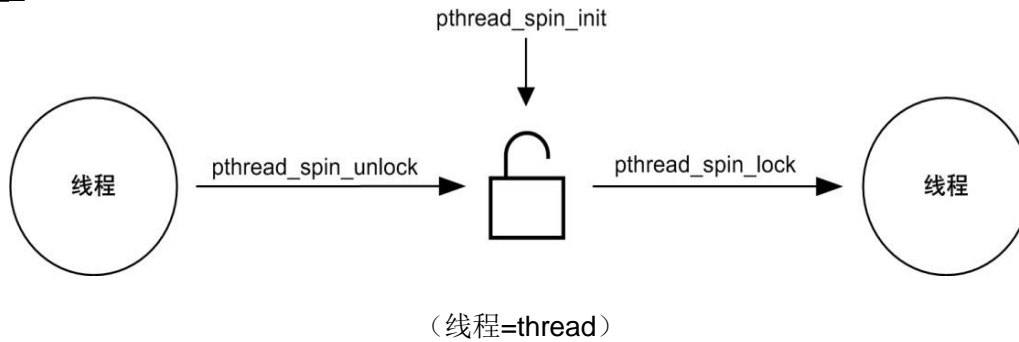


Figure 7.19 POSIX spin lock

7.14.1 Spin lock

1. Initialization and deletion of the spin lock

```
#include <pthread.h>
int  pthread_spin_init(pthread_spinlock_t *pspinlock, int  pshare);
```

Prototype analysis of Function pthread_spin_init:

- For success of the function, return 0. For failure, return the error number;
- Parameter **p~~spinlock~~** is the pointer of POSIX spinlock;
- Parameter **p~~share~~** labels whether POSIX spin lock has process sharing.

```
#include <pthread.h>
int  pthread_spin_destroy(pthread_spinlock_t *pspinlock);
```

Prototype analysis of Function pthread_spin_destroy:

- For success of the function, return 0. For failure, return the error number;
- Parameter **p~~spinlock~~** is the pointer of POSIX spinlock.

2. Locking and unlocking of the spin lock

```
#include <pthread.h>
int  pthread_spin_lock(pthread_spinlock_t *pspinlock);
int  pthread_spin_unlock(pthread_spinlock_t *pspinlock);
int  pthread_spin_trylock(pthread_spinlock_t *pspinlock);
```

Prototype analysis of above functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter **p~~spinlock~~** is the pointer of POSIX spinlock.

`pthread_spin_trylock` is the "try-to-wait" version of `pthread_spin_lock`. When the POSIX spin lock has already occupied, `pthread_spin_lock` will "spin", and `pthread_spin_trylock` will return immediately.

At the moment, the unlock operation of the POSIX spin lock can only call the `pthread_spin_unlock` function.

If the interrupt service routine may also use resources protected by POSIX spin lock, the POSIX spin lock API with an interrupt mask version shall be used. Otherwise, deadlock will occur:

```
#include <pthread.h>
int  pthread_spin_lock_irq_np(pthread_spinlock_t *pspinlock,
                             pthread_int_t      *irqctx);
int  pthread_spin_unlock_irq_np(pthread_spinlock_t *pspinlock,
                                pthread_int_t      irqctx);
int  pthread_spin_trylock_irq_np(pthread_spinlock_t *pspinlock,
                                 pthread_int_t      *irqctx);
```

Prototype analysis of above several functions:

- For success of the function, return 0. For failure, return the error number;
- Parameter ***pspinlock*** is the pointer of POSIX spinlock;
- Parameter ***irqctx*** is the pointer of `pthread_int_t` type, which is used to save and restore the interrupt mask register of CPU.

`pthread_spin_trylock_irq_np` is a "try-to-wait" version of `pthread_spin_lock_irq_np`. When the POSIX spin lock has already been occupied, `pthread_spin_lock_irq_np` will "spin", and `pthread_spin_trylock_irq_np` will return immediately.

At this time, the unlock operation of POSIX spin lock can only call the `pthread_spin_unlock_irq_np` function.

The following program shows how to use POSIX spin lock. The program creates two threads and a POSIX spin lock. Two threads perform auto increment operation and printing of Variable count respectively. POSIX spin lock is used as mutex for access to Variable count.

Program List 7.12 Use of POSIX spin lock

```
#include <stdio.h>
#include <pthread.h>

static pthread_spinlock_t lock;
static int count = 0;

static void *thread_a (void *arg)
```

```
{
int value;

while (1) {
    pthread_spin_lock(&lock);
    value = count;
    pthread_spin_unlock(&lock);

    printf("thread_a(): count = %d\n", value);

    sleep(1);
}
return (NULL);
}

static void *thread_b (void *arg)
{
    while (1) {
        pthread_spin_lock(&lock);
        count++;
        pthread_spin_unlock(&lock);

        sleep(1);
    }
    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t      threada_tid;
    pthread_t      threadb_tid;
    int            ret;

    ret = pthread_spin_init(&lock, FALSE);
    if (ret != 0) {
        fprintf(stderr, "mutex create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threada_tid, NULL, thread_a, NULL);
    if (ret != 0) {
```

```
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threadb_tid, NULL, thread_b, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    pthread_join(threada_tid, NULL);
    pthread_join(threadb_tid, NULL);

    pthread_spin_destroy(&lock);

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./posix_spin
thread_a(): count = 0
thread_a(): count = 1
thread_a(): count = 2
thread_a(): count = 3
thread_a(): count = 4
thread_a(): count = 5
```

7.15 SylixOS atomic number

At the beginning of this chapter, we introduce the risk of chaos in the multi-thread environment for the auto increment operation of Variable V. To solve this problem, we propose the solution to add a lock to protect auto increment operation of Variable V.

When our program has more variables like Variable V or more access to such variables, our program will inevitably have more lock and lock operations. On the one hand, this makes our program difficult to write and maintain. On the other hand, unreasonable lock operation may cause deadlock.

In order to avoid these problems, SylixOS provides the atomic type `atomic_t` and its API. The atomic number type can store an integer INT value. At the same time, the operation performed to the atomic number with atomic number API is an atomic operation, because atomic operation cannot be interrupted. There is no risk of confusion in the multi-thread environment.

The type of SylixOS atomic number is `atomic_t`. A variable of `atomic_t` type shall be defined during use, such as:

```
atomic_t    atomic;
```

7.15.1 Atomic number

1. Setting and getting of atomic number

```
#include <SylixOS.h>
VOID    Lw_Atomic_Set(INT    iVal, atomic_t    *patomic);
```

Prototype analysis of Function `Lw_Atomic_Set`:

- Parameter ***iVal*** is the value to be set;
- Parameter ***patomic*** is the pointer of atomic number.

```
#include <SylixOS.h>
INT     Lw_Atomic_Get(atomic_t    *patomic);
```

Prototype analysis of Function `Lw_Atomic_Get`:

- For success of the function, return the value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***patomic*** is the pointer of atomic number.

2. Addition and subtraction of the atomic number

```
#include <SylixOS.h>
INT     Lw_Atomic_Add(INT    iVal, atomic_t    *patomic);
```

Prototype analysis of Function `Lw_Atomic_Add`:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***iVal*** is the value to be added;
- Parameter ***patomic*** is the pointer of atomic number.

```
#include <SylixOS.h>
INT     Lw_Atomic_Sub(INT    iVal, atomic_t    *patomic);
```

Prototype analysis of Function `Lw_Atomic_Sub`:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***iVal*** is the value to be subtracted;

● pointer of atomic number.

3. Auto increment and auto decrement of atomic number

```
#include <SylixOS.h>
INT    Lw_Atomic_Inc(atomic_t  *patomic);
INT    Lw_Atomic_Dec(atomic_t  *patomic);
```

Prototype analysis of above two functions:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***patomic*** is the pointer of atomic number.

4. Logic bit operation of atomic number

```
#include <SylixOS.h>
INT    Lw_Atomic_And(INT  iVal, atomic_t  *patomic);
```

Prototype analysis of Function Lw_Atomic_And:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***iVal*** is the value requiring logic bit and operation;
- Parameter ***patomic*** is the pointer of atomic number.

```
#include <SylixOS.h>
INT    Lw_Atomic_Nand(INT  iVal, atomic_t  *patomic);
```

Prototype analysis of Function Lw_Atomic_Nand:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***iVal*** is the value requiring logic bit and not operation;
- Parameter ***patomic*** is the pointer of atomic number.

```
#include <SylixOS.h>
INT    Lw_Atomic_Or(INT  iVal, atomic_t  *patomic);
```

Prototype analysis of Function Lw_Atomic_Or:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter ***iVal*** is the value requiring logic bit or operation;
- Parameter ***patomic*** is the pointer of atomic number.


```
#include <SylixOS.h>
INT Lw_Atomic_Xor(INT iVal, atomic_t *patomic);
```

Prototype analysis of Function Lw_Atomic_Xor:

- For success of the function, return the new value of the atomic number. For failure, return -1 and set the error number;
- Parameter *iVal* is the value requiring logic bit XOR operation;
- Parameter *patomic* is the pointer of atomic number.

5. Swap operation of atomic number

```
#include <SylixOS.h>
INT Lw_Atomic_Swp(INT iVal, atomic_t *patomic);
```

Prototype analysis of Function Lw_Atomic_Swp:

- For success of the function, return the old value of the atomic number (value before operation). For failure, return -1 and set the error number;
- Parameter *iVal* is the value to be set;
- Parameter *patomic* is the pointer of atomic number.

The following program shows how to use POSIX atomic number. The program creates two threads. The two threads perform auto increment operation and printing to atomic number `_G_atomicCount` respectively.

Program List 7.13 Use of POSIX atomic number

```
#include <SylixOS.h>

static atomic_t _G_atomicCount;

static PVOID tTestA (PVOID pvArg)
{
    while (1) {
        printf("tTestA(): count = %d\n", API_AtomicGet(&_G_atomicCount));

        Lw_Time_SSsleep(1);
    }
    return (LW_NULL);
}

static PVOID tTestB (PVOID pvArg)
{
    while (1) {
```

```
Lw_Atomic_Inc(&_G_atomicCount);

    Lw_Time_SSleep(1);
}
return (LW_NULL);
}

int main (int argc, char *argv[])
{
    LW_HANDLE          hThreadAId;
    LW_HANDLE          hThreadBId;

    Lw_Atomic_Set(0, &_G_atomicCount);

    hThreadAId = Lw_Thread_Create("t_testa", tTestA, LW_NULL, LW_NULL);
    if (hThreadAId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testa create failed.\n");
        return (-1);
    }

    hThreadBId = Lw_Thread_Create("t_testb", tTestB, LW_NULL, LW_NULL);
    if (hThreadBId == LW_OBJECT_HANDLE_INVALID) {
        printf("t_testb create failed.\n");
        return (-1);
    }

    Lw_Thread_Join(hThreadAId, LW_NULL);
    Lw_Thread_Join(hThreadBId, LW_NULL);

    return (0);
}
```

Run the program under the SylixOS Shell:

```
# ./Atomic
tTestA(): count = 0
tTestA(): count = 1
tTestA(): count = 2
tTestA(): count = 3
tTestA(): count = 4
```

7.16 One-time initialization

Sometimes, we need to perform one-time initialization of some POSIX objects, such as the thread key `pthread_key_t`. If we perform multiple initializations, an error will occur.

During traditional sequential programming, one-time initialization is often managed by using Boolean `BOOL` type variables. Boolean type control variable is statically initialized as `FALSE`, and any code which relies on initialization can test the variable. If the variable value is `FALSE`, initialization is performed, and then the variable value is set as `TRUE`. The code checked later will skip initialization.

However, things will get complicated in the multi-threaded program. If multiple threads concurrently execute the initialization sequence code, multiple threads may find that the variable value is `FALSE` at the same time, and are initialized. However, the process is only performed once.

Although we can add a POSIX mutex semaphore to protect the initialization process from being executed for multiple times, it is much more convenient to use the `pthread_once_t` variable and the `pthread_once` function provided by the POSIX standard.

Definition and initialization of the `pthread_once_t` variable:

```
static pthread_once_t once = PTHREAD_ONCE_INIT;
```

7.16.1 pthread_once_t variable

```
#include <pthread.h>
int pthread_once(pthread_once_t *once, void (*pfunc)(void));
```

Prototype analysis of Function `pthread_once`:

- This function returns error number `ERRNO_NONE` or POSIX standard error number (errno records the cause of error);
- Parameter ***once*** is the pointer of `pthread_once_t` type variable;
- Parameter ***pfunc*** is a function pointer which completes one-time initialization.

SylixOS also provides an API similar to the `pthread_once` function:

```
#include <SylixOS.h>
INT Lw_Thread_Once(BOOL *pbOnce, VOIDFUNCPTR pfuncRoutine);
```

Prototype analysis of Function `Lw_Thread_Once`:

- The function returns the error number;
- Parameter ***pbOnce*** is the pointer to Boolean `BOOL` type variable;

- Parameter ***pfuncRoutine*** is the function pointer which completes one-time initialization.

The following program shows how to use the `pthread_once_t` variable and the `pthread_once` function. This program is modified from Program List 7.8, and places creation of POSIX conditional variable and POSIX mutex in the one-time initialization function. Although the `pthread_once(&once, var_init)` statement is called twice, the `var_init` function will only be called once.

Program List 7.14 Use of `pthread_once_t` variable and `pthread_once` function

```
#include <stdio.h>
#include <pthread.h>

static pthread_once_t      once = PTHREAD_ONCE_INIT;
static pthread_mutex_t     lock;
static pthread_cond_t      cond;
static int                 count = 0;

static void var_init(void)
{
    pthread_mutex_init(&lock, NULL);

    pthread_cond_init(&cond, NULL);
}

static void *thread_a (void *arg)
{
    pthread_once(&once, var_init);

    while (1) {
        pthread_mutex_lock(&lock);

        pthread_cond_wait(&cond, &lock);

        printf("thread_a(): count = %d\n", count);

        pthread_mutex_unlock(&lock);
    }
    return (NULL);
}

static void *thread_b (void *arg)
{
```

```
pthread_once(&once, var_init);

while (1) {
    pthread_mutex_lock(&lock);

    count++;

    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&lock);

    sleep(1);
}
return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t      threada_tid;
    pthread_t      threadb_tid;
    int            ret;

    ret = pthread_create(&threada_tid, NULL, thread_a, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    ret = pthread_create(&threadb_tid, NULL, thread_b, NULL);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    pthread_join(threada_tid, NULL);
    pthread_join(threadb_tid, NULL);

    return 0;
}
```

Run the program under the SylixOS Shell:

```
# ./posix_once
thread_a(): count = 1
thread_a(): count = 2
```

```
thread_a(): count = 3  
thread_a(): count = 4  
thread_a(): count = 5
```

Chapter 8 Process management

8.1 Real-time process

The process is the container for resources in the operating system. All applications must be attached to the process for running, and the process manages the code, data, thread, semaphore and other resources of the program. When a process is destroyed, all resources belonging to the process will also be destroyed, such as: file handles, socket, thread and so on.

SylixOS support process. SylixOS process manages application resources, as described above. Unlike Linux and Windows operating systems, the SylixOS process is designed to take full account of real-time system demands. We call this as real-time process. SylixOS improves process real-time performance in following two aspects:

- All threads in SylixOS process use real-time scheduling algorithm for scheduling.
- All SylixOS processes share an address space. There is no need to switch page tables during task switching. Existence of process has no effect on the real-time performance of task switching.

Executing an executable file in SylixOS Shell will create a process in the system. The current running process can be viewed with the **ps** command. Execute the program named app in the current directory as follows:

```
# ./app&
app is running

# ps

      NAME                FATHER      PID  GRP  MEMORY  UID  GID  USER
-----
kernel                <orphan>    0    0    0       0   0   root
app                    <orphan>    2    2    65536   0   0   root

total vprocess : 2
```

Adding "&" symbol behind the command indicates that the process is executed at the background. The user program part of the process in SylixOS system starts execution from the main function. The thread where the main function is located is the main thread of the process. The main thread can call the thread creation function to create other threads.

Note: the executable files of the process and its child process in all instances set in the section are stored in the current directory of Shell.

8.2 Process state machine

The process state reflects different stages of the process execution process. The process state is converted with process execution and changes in external conditions. SylixOS process has the following four states.

- Initialization state: the process is still in the initialization process, and the program loading, memory initialization and other operations are being performed, and the operating conditions are not yet available;
- Running state: the process is running, and the thread in the process is either involved in scheduling or at the blocked state;
- Exit state: the process has finished running. When the process enters the exit state, the signal will be sent to its parent process. The parent process will timely recover the remaining resources of the child process. In case of zombie process, the system will recycle resources after the exit state is entered.;
- Stopped state: some processes will enter the stopped state in the running process. At the stopped state, all threads of the process stop running, and do not participate in scheduling. In the debugging the process, the debugger will often let the process enter the stopped state, so as to observe the process data.

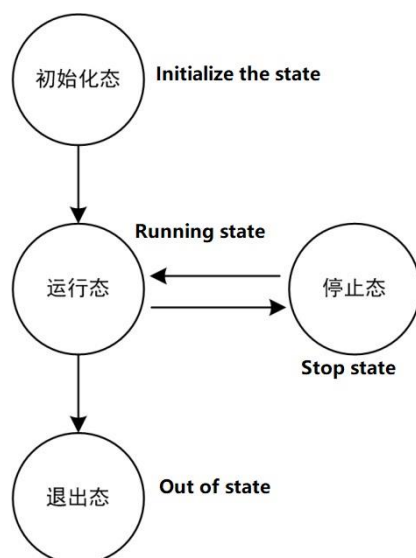


Figure 8.1 Process state machine

8.3 POSIX processAPI

In addition to executing the program creation process in SylixOS Shell, SylixOS also provides APIs for creating process and setting process parameters in the program. When

a process creates another process, this process becomes the parent process of the created process, and the created process becomes the child process of this process. The parent-child processes are related to each other and can call the corresponding API to find each other. When the child process exits, the child process will send the signal to notify the parent process. At the moment, the parent process can obtain the child process exit code, and recycle the child process resources. If the parent process of a process exits before the child process, the child process will become the orphan process, and the system will automatically complete resource recycling of the orphan process at exit.

SylixOS provides a set of POSIX-compatible APIs. Therefore, it is easy to write SylixOS program or port the program to SylixOS.

8.3.1 Execute program

1. Use the exec function to execute program

```
#include <process.h>
int execl(const char *path, const char *argv0, ...);
int execl_e(const char *path, const char *argv0, ...);
int execlp(const char *file, const char *argv0, ...);
int execv(const char *path, char * const *argv);
int execve(const char *path, char * const *argv, char * const *envp);
int execvp(const char *file, char * const *argv);
int execvpe(const char *file, char * const *argv, char * const *envp);
```

Function prototype analysis:

- Return the function execution results. For success of the function, return 0. For failure, return -1 and set the error code.
- Parameter **path** is the executable file path;
- Parameter **argv0** is the first command line parameter, which is generally the command name;
- Parameter **file** is the executable file name. What's different from Parameter **path** is that it does not have the directory, and the application loader search the file at the specified path. The search paths of application dynamic library in SylixOS are as follows in sequence:
 - ◆ Shell current directory (usually the user's home directory), but not the application directory;
 - ◆ Search path included in PATH environment.
- Parameter...is the variable parameter, which represents the remaining parameters in the command line, and the command line parameter is ended with

0. In the `execle` function, there is an array of environment variables behind the command line parameter which ends with 0. The array ends with 0. Refer to instructions for the ***envp*** parameter;

- Parameter ***argv*** is the array of character strings consisting of command line parameter. The array starts with the executable file name and ends with 0. 0 indicates no use of command line parameter;
- Parameter ***envp*** is the array of process environment variable character strings preset before executing the program. The array ends with 0. 0 indicates that it is not required to set the environment variable.

Note: this function can only be called by the main thread of the current process. Otherwise, it will return failure.

The following instance shows how to use `exec` series functions. Program List 8.2, Program List 8.3 and Program List 8.4 show how to use the `execl`, `execle` and `execve` functions respectively.

Program List 8.1 `exec` instance child process

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    char *env_parent = (char*)0;

    if (argc < 2) {
        printf("child process param error!\n");
        return (-1);
    }

    printf("child process: %s\n", argv[1]);

    env_parent = getenv("PARENT");
    if (env_parent) {
        printf("environment variable: PARENT = %s\n", env_parent);
    }

    return (0);
}
```

Program List 8.2 execl instance

```
#include <stdio.h>
#include <process.h>

int main (int argc, char *argv[])
{
    printf("before execl\n");
    execl("./child_process", "child_process",
          "execl test", (char *)0);

    printf("after execl\n");
    return 0;
}
```

Run the program in the SylixOS Shell:

```
# ./execl_test
before execl
child process, execl test
```

Program List 8.3 execl instance

```
#include <stdio.h>
#include <process.h>

int main (int argc, char *argv[])
{
    char *env[] = { "PARENT=execle_demo", (char *)0 };

    printf("before execle\n");
    execle("./child_process", "child_process",
           "execle demo", (char *)0, env);

    printf("after execle\n");
    return 0;
}
```

Run the program in the SylixOS Shell:

```
# ./execle_demo
before execle
child process, execle demo
environment variable PARENT = execle_demo
```

Program List 8.4 execve instance

```

#include <stdio.h>
#include <process.h>

int main (int argc, char *argv[])
{
    char *cmd[] = { "child_process", "execve test", (char *)0 };
    char *env[] = { "PARENT=execve_demo", (char *)0 };

    printf("before execve\n");

    execve("./child_process", cmd, env);

    printf("after execve\n");

    return 0;
}

```

Run the program in the SylixOS Shell:

```

# ./execve_demo
before execve
child process, execve test
environment variable PARENT = execve_demo

```

It can be seen from the instance that the exec function will overwrite this process environment. Therefore, the print statement behind the exec function will not be executed.

8.3.2 Create process

1. Use the posix_spawn function to create the process

The posix_spawn series functions are more powerful than the exec series functions in functions, and more complex in use. The posix_spawn can set the command line parameter and environment variable of the new process, and can also set file operations and process attributes of the new process. The function prototype is as follows:

```

#include <spawn.h>
int posix_spawn(pid_t *pid, const char *path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *attrp,
                char *const argv[],
                char *const envp[]);
int posix_spawnnp(pid_t *pid, const char *file,
                  const posix_spawn_file_actions_t *file_actions,
                  const posix_spawnattr_t *attrp,

```

```

char *const          argv[],
char *const          envp[]);

```

Function prototype analysis:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter **pid** saves the new process ID;
- Parameter **path** is the executable file path;
- Parameter **file** is the executable file name. What's different from Parameter **path** is that it does not have the directory, and the application loader search the file at the specified path. The search paths of application dynamic library in SylixOS are as follows in sequence:
 - ◆ Shell current directory, but not the application directory;
 - ◆ Search path included in PATH environment.
- Parameter **file_actions** is the file operation set to be processed when the new process starts, and 0 indicates that no file operations are performed. Construction, value assignment and other operations for the **file_actions** are introduced in subsequent functions, which is named as the file operation set object in the introduction of the chapter below.
- Parameter **attrp** is the initialization attribute of the new process, and NULL indicates no setting. Construction, value assignment and other operations for the **attrp** are introduced in subsequent functions, which is named as the process attribute object in the introduction of the chapter below.
- Parameter **argv** is the character string array consisting of command line parameters. The array starts with the executable file name and ends with NULL. NULL indicates that command line parameters are not used;
- Parameter **envp** is the character string array of the process environment variable to be preset, ending with NULL. NULL indicates that it is not required to set the environment variable.

2. Initialize the process attribute object

```

#include <spawn.h>
int posix_spawnattr_init(posix_spawnattr_t *attrp);

```

Prototype analysis of Function `posix_spawnattr_init`:

- For success of the function, return 0. For failure, return the error code;
- Parameter **attrp** is the process attribute object to be initialized.

3. Destroy the process attribute object

```
#include <spawn.h>
int posix_spawnattr_destroy(posix_spawnattr_t *attrp);
```

Prototype analysis of Function `posix_spawnattr_destroy`:

- For success of the function, return 0. For failure, return the error code;
- Parameter **attrp** is the process attribute object to be destroyed.

4. Set the working directory in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_setwd(posix_spawnattr_t *attrp, const char *pwd);
```

Prototype analysis of Function `posix_spawnattr_setwd`:

- For success of the function, return 0. For failure, return the error code;
- Parameter **attrp** is the process attribute object;
- Parameter **pwd** is the character string of the new working directory.

5. Get the working directory in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_getwd(const posix_spawnattr_t *attrp,
                          char *pwd, size_t size);
```

Prototype analysis of Function `posix_spawnattr_getwd`:

- For success of the function, return 0. For failure, return the error code;
- Parameter **attrp** is the process attribute object;
- Parameter **pwd** is the buffer zone used to save the character string of the working directory;
- Parameter **size** is the length of the buffer zone.

6. Set the signal mask in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_setsigmask(posix_spawnattr_t *attrp,
                               const sigset_t *sigmask);
```

Prototype analysis of Function `posix_spawnattr_setsigmask`:

- For success of the function, return 0. For failure, return the error code;
- Parameter **attrp** is the process attribute object;
- Parameter **sigmask** is the process signal mask to be set.

7. Get the signal mask in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_getsigmask(const posix_spawnattr_t *attrp,
                               sigset_t *sigmask);
```

Prototype analysis of Function `posix_spawnattr_getsigmask`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***sigmask*** is the process signal mask got.

8. Set the flag bit in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_setflags(posix_spawnattr_t *attrp,
                             short flags);
```

Prototype analysis of Function `posix_spawnattr_setflags`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***flags*** is the flag bit mask of the process attribute. Only the enabled attribute in the flag bit will take effect when the process starts, and the value of the flag bit mask is any combination of the following values.

Table 8.1 Flag bit of the process attribute object

Macro name	Explanation
POSIX_SPAWN_SETPGROUP	Enabled process group setting
POSIX_SPAWN_SETSIGMASK	Enable signal mask setting
POSIX_SPAWN_SETSCHEDULER	Enable scheduler parameter setting
POSIX_SPAWN_SETSCHEDPARAM	Enable process priority setting

9. get the flag bit in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_getflags(const posix_spawnattr_t *attrp,
                             short *flags);
```

Prototype analysis of Function `posix_spawnattr_getflags`:

- For success of the function, return 0. For failure, return the error code;

Parameter ***attrp*** is the

process attribute object;

- Parameter ***flags*** is the flag bit mask of the process attribute. Only the enabled attribute in the flag bit will take effect when the process starts, and values of the flag bit are shown in Table 8.1.

10. Set the process group number to the process attribute object

```
#include <spawn.h>
int posix_spawnattr_setpgroup(posix_spawnattr_t *attrp,
                              pid_t pgroup);
```

Prototype analysis of Function `posix_spawnattr_setpgroup`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***pgroup*** is the process group number.

11. Get the process group number from the process attribute object

```
#include <spawn.h>
int posix_spawnattr_getpgroup(const posix_spawnattr_t *attrp,
                              pid_t *pgroup);
```

- Prototype analysis of Function `posix_spawnattr_getpgroup`:
- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***pgroup*** is the process group number returned.

12. Set the scheduling policy in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attrp,
                                   int schedpolicy);
```

Prototype analysis of Function `posix_spawnattr_setschedpolicy`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***schedpolicy*** is the process scheduling policy. Parameter ***schedpolicy*** can have the following cases:

Table 8.2 Table of process scheduling policy

Macro name	Explanation
------------	-------------

LW_OPTION_SCHED_FIFO	SCHED_FIFO FCFS real-time scheduling policy
LW_OPTION_SCHED_RR	SCHED_RR round-robin real-time scheduling policy

13. Get the scheduling policy in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attrp,
                                   int *schedpolicy);
```

Prototype analysis of Function `posix_spawnattr_getschedpolicy`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***schedpolicy*** is the process scheduling policy got

14. Set the process priority in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_setschedparam(posix_spawnattr_t *attrp,
                                   const struct sched_param *schedparam);
```

Prototype analysis of Function `posix_spawnattr_setschedparam`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***schedparam*** is the scheduling parameter (see 6.7 POSIX thread scheduling).

15. Get the process priority in the process attribute object

```
#include <spawn.h>
int posix_spawnattr_getschedparam(const posix_spawnattr_t *attrp,
                                   struct sched_param *schedparam);
```

Prototype analysis of Function `posix_spawnattr_getschedparam`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***attrp*** is the process attribute object;
- Parameter ***schedparam*** is the process priority setting parameters got.

16. Initialize the file operation set object

```
#include <spawn.h>
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *file_actions);
```

Prototype analysis of Function `posix_spawn_file_actions_init`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***file_actions*** is the file operation set object.

17. Destroy the file operation set object

```
#include <spawn.h>
int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *file_actions);
```

Prototype analysis of Function `posix_spawn_file_actions_destroy`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***file_actions*** is the file operation set object.

18. Add open file operation to the file operation set object

```
#include <spawn.h>
int posix_spawn_file_actions_addopen(
    posix_spawn_file_actions_t *file_actions,
    int fd,
    const char *path,
    int oflag,
    mode_t mode
);
```

Prototype analysis of Function `posix_spawn_file_actions_addopen`:

- For success of the function, return 0. For failure, return the error code;
- Parameter ***file_actions*** is the file operation set object;
- Parameter ***fd*** is number of the file opened;
- Parameter ***path*** is the file path;
- Parameter ***oflag*** is the file open mode;
- Parameter ***mode*** is valid only when creating a new file, indicating the creation mode of the new file.

19. Add close file operation to the file operation set object

```
#include <spawn.h>
int posix_spawn_file_actions_addclose(
    posix_spawn_file_actions_t *file_actions,
    int fd
);
```

Prototype analysis of Function `posix_spawn_file_actions_addclose`:

For success of the

- function, return 0. For failure, return the error code;

- Parameter **file_actions** is the file operation set object;
- Parameter **fd** is the number of the file to be closed.

20. Add file descriptor dup operation to the file operation set object

```
#include <spawn.h>
int posix_spawn_file_actions_adddup2(
    posix_spawn_file_actions_t *file_actions,
    int fd,
    int newfd
);
```

Prototype analysis of Function `posix_spawn_file_actions_adddup2`:

- For success of the function, return 0. For failure, return the error code;
- Parameter **file_actions** is the file operation set object;
- Parameter **fd** is the number of the file to be copied;
- Parameter **newfd** is the new file number pointing to the file corresponding to Parameter **fd**.

The following instances indicate how to use POSIX API to create the process.

Program List 8.5 `posix_spawn` function instance

```
#include <stdio.h>
#include <spawn.h>

int main (int argc, char *argv[])
{
    posix_spawn_file_actions_t file_actions;
    struct sched_param schedparam;
    posix_spawnattr_t spawnattr;
    pid_t pid;

    char *cmd[] = { "child_process", "execve test", (char *)0 };
    char *env[] = { "PARENT=execve_demo", (char *)0 };
    char *log_file = "/tmp/child_process_output";
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

    /*
     * Initialize process attribute
     */
```

```

    if (posix_spawnattr_init(&spawnattr) != 0) {
        fprintf(stderr, "init posix_spawnattr_t failed\n");
        return (-1);
    }
    /*
     * Set the priority of the new process is NORMAL.
     */
    schedparam.sched_priority = PX_PRIORITY_CONVERT(LW_PRIO_NORMAL);
    posix_spawnattr_setschedparam(&spawnattr, &schedparam);
    posix_spawnattr_setflags(&spawnattr, POSIX_SPAWN_SETSCHEDPARAM);

    /*
     * Initialize file operation handler
     */
    if (posix_spawn_file_actions_init(&file_actions) != 0) {
        fprintf(stderr, "init posix_spawn_file_actions_t failed\n");
        return (-2);
    }

    /*
     * Turn off standard input, standard output, error output
     */
    if (posix_spawn_file_actions_addclose(&file_actions, STDIN_FILENO) != 0 ||
        posix_spawn_file_actions_addclose(&file_actions, STDOUT_FILENO) != 0 ||
        posix_spawn_file_actions_addclose(&file_actions, STDERR_FILENO) != 0) {
        fprintf(stderr, "close std file failed\n");
        return (-3);
    }

    /*
     * Redirect standard output to log file
     */
    if (posix_spawn_file_actions_addopen(&file_actions,
                                         STDOUT_FILENO,
                                         log_file,
                                         O_WRONLY | O_CREAT | O_TRUNC,
                                         mode) != 0) {
        fprintf(stderr, "redirection std output failed\n");
        return (-4);
    }

    if (posix_spawn(&pid, "./child_process",
                   &file_actions,

```

```

        &spawnattr, cmd, env) != 0) { /* 启动进程
*/
    posix_spawnattr_destroy(&spawnattr);
    posix_spawn_file_actions_destroy(&file_actions);

    return (-6);
}

posix_spawnattr_destroy(&spawnattr);
posix_spawn_file_actions_destroy(&file_actions);

return (0);
}

```

The instance uses the program in Program List 8.1 as the executable file of the child process, and the program is executed in SylixOS Shell:

```

# ./posix_spawn_demo
#

```

The standard output has been redirected when the program starts. Therefore, any program output cannot be seen in Shell, which can be seen by checking contents in the /tmp/child_process_output file.

```

# cat /tmp/child_process_output
child process, execve test
environment variable PARENT = execve_demo

```

8.3.3 Process scheduling

The process is the basic unit of system resource allocation (it can be seen as the container of resources), and the thread is the basic unit of scheduling. Therefore, SylixOS process scheduling refers to scheduling for the main thread of the process.

1. Set the process scheduling priority

Sets the SylixOS scheduling priority of all threads satisfying conditions.

```

#include <sys/resource.h>
int setpriority(int which, id_t who, int value);

```

Prototype analysis of Function setpriority:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *which* appoints the meaning of Parameter *who*;

- The meaning of Parameter **who** is appointed by Parameter **which**, as shown in Table 8.3:

Table 8.3 Corresponding relation between Parameter which and Parameter who

Macro name	Explanation
PRIO_PROCESS	The value of Parameter who is the process ID
PRIO_PGRP	The value of Parameter who is the group ID
PRIO_USER	The value of Parameter who is the user ID

- Parameter **value** is the SylixOS priority to be set.

2. Get the SylixOS scheduling priority

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
```

Prototype analysis of Function getpriority:

- Return the maximum value of the SylixOS priority in all threads satisfying conditions. Note: the priority value is the largest and therefore the priority is the lowest here;
- Parameter **which** appoints the meaning of Parameter **who**, as shown in Table 8.3;
- The meaning of Parameter **who** is appointed by Parameter **which**.

Function setpriority and Function getpriority set and get the SylixOS priority. Different from the priority indicated by the sched_priority member in the previous sched_param structure, the sched_priority member is the POSIX priority.

3. Adjust the SylixOS scheduling priority

The nice function can adjust the priority of current process.

```
#include <unistd.h>
int nice(int incr);
```

Prototype analysis of Function nice:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **incr** is the value to be adjusted. The process flow of the function to Parameter **incr** is as follows:

(1) Firstly, get the lowest priority among all threads in the current process, i.e., the priority with the maximum value;

(2) Then sum the value got and Parameter incr;

(3) Set the result of the previous sum to all threads in the current process.

The following instances indicate how to use the POSIX process to schedule API setting and get the process priority.

Program List 8.6 Scheduler parameter setting instance child process

```
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>

int main (int argc, char *argv[])
{
    int    pid;
    int    priority;

    pid = getpid();                /* Get process ID      */

    priority = getpriority(PRIO_PROCESS, pid); /* Get process priority*/
    fprintf(stdout, "child process priority: %d\n", priority);
    sleep(3);                      /* Waiting for the parent
process to modify the priority */

    priority = getpriority(PRIO_PROCESS, pid); /* Get modified priority
*/
    fprintf(stdout, "child process priority after sched_setscheduler: %d\n",
priority);

    nice(1);                        /* Call nice to adjust
priority */

    priority = getpriority(PRIO_PROCESS, pid); /* Get adjusted priority
*/
    fprintf(stdout, "child process priority after nice: %d\n", priority);

    return (0);
}
```

Program List 8.7 Scheduler parameter setting instance parent process

```
#include <stdio.h>
```

```

#include <spawn.h>
#include <sched.h>

int main (int argc, char *argv[])
{
    pid_t          pid;
    struct sched_param  param;
    int            policy;
    char           *policy_name[] = {"SCHED_RR",
                                     "LW_OPTION_SCHED_FIFO"};

    if (posix_spawn(&pid, "./sched_child_proc",
                   NULL, NULL, NULL, NULL) != 0) { /* Startup process */
        fprintf(stderr, "create child process failed\n");
        return (-1);
    }

    sleep(1); /* Waiting for the child
process to start */

    if (sched_getparam(pid, &param) != 0) { /* Get child process
priority */
        fprintf(stderr, "get sched_param failed\n");
        return (-3);
    }

    policy = sched_getscheduler(pid); /* Get process
scheduling policy */
    if (policy == -1) {
        fprintf(stderr, "get scheduler policy failed\n");
        return (-4);
    }

    fprintf(stdout, "child process pid:%d, posix priority:%d, policy:%s\n",
           pid, param.sched_priority, policy_name[policy]);
    param.sched_priority += 1;

    if (sched_setscheduler(pid,
                           policy,
                           &param) != 0) { /* Set process priority and
scheduling policy */
        fprintf(stderr, "get sched_param and scheduler policy failed\n");
        return (-5);
    }
}

```



```

    }
    fprintf(stdout, "set posix priority to %d\n", param.sched_priority);

    sleep(5); /* Wait for the child process
to end */
    return (0);
}

```

The parent process in the instance uses the `sched_getparam` function to get the child process priority, and the child process uses the `getpriority` function to get its own priority. Therefore, conversion relation between the POSIX priority and the SylixOS system priority can be seen. Run the program in the SylixOS Shell:

```

# ./posix_sched_demo
child process priority: 200
child process pid:18, posix priority:55, policy:SCHED_RR
set posix priority to 56
child process priority after sched_setscheduler: 199
child process priority after nice: 200

```

4. Set process affinity

The `sched_setaffinity` function locks all threads of the process running on the appointed cpu set, only for the multi-core situation.

```

#include <sys/resource.h>
int sched_setaffinity(pid_t pid, size_t setsize, const cpu_set_t *set);

```

Prototype analysis of Function `sched_setaffinity`:

- For success of the function, return 0. For failure, return -1 and set the error code.
- Parameter *pid* appoints the process ID.
- Parameter *set* appoints the processor core allowing the process to execute. It is a 2048-bit bit set, and each bit represents a processor core. 1 indicates that the process is allowed to execute on the core. Otherwise, it is not allowed.

5. Get the main thread affinity of the process

The `sched_getaffinity` function gets affinity setting of the main thread processor of the process.

```

#include <sys/resource.h>
int sched_getaffinity(pid_t pid, size_t setsize, cpu_set_t *set);

```

Prototype analysis of Function `sched_getaffinity`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pid*** appoints the process ID;
- Parameter ***set*** represents the processor core allowing the process to execute. It is a 2048-bit bit set, and each bit represents a processor core. 1 indicates that the process is allowed to execute on the core. Otherwise, it is not allowed.

8.3.4 Process relation

1. Get the process ID

```
#include <unistd.h>
pid_t getpid(void);
```

Prototype analysis of Function getpid:

- The function returns the calling process ID

2. Set process group ID

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Prototype analysis of Function setpgid:

- For success of the function, return 0. For failure, return -1 and set the error code.
- Parameter ***pid*** is the process ID;
- Parameter ***pgid*** is the process group ID to be set.

3. Get process group ID

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

Prototype analysis of Function getpgid:

- For success of the function, return the target process group ID. For failure, return -1 and set the error code;
- Parameter ***pid*** is the process ID;

4. Set the process as the session header

```
#include <unistd.h>
pid_t setpgrp(void);
```

Prototype analysis of Function setpgrp:

- This function will call the process group ID as this process ID, making this process the session header. For success of the function, return 0. For failure, return -1 and set the error code.

5. Get process group ID

```
#include <unistd.h>
pid_t getpgrp(void);
```

Prototype analysis of Function getpgrp:

- The function returns the calling process group ID

6. Get the parent process ID

```
#include <unistd.h>
pid_t getppid(void);
```

Prototype analysis of Function getppid:

- The function returns the parent process ID of the calling process.

Each SylixOS process includes three user IDs:

- **Actual user ID:** the actual user ID is the user ID of the start process, and the Shell login user ID of the start process;
- **Valid user ID:** the valid user ID is the user ID currently used by the process. If authority judgment is required, the kernel will only verify the valid user ID.
- **Saved set user ID:** the saved set user ID is the owning user ID of the executable file of the process, and the saved set user ID is valid only if the executable file sets the S_ISUID bit.

In the same way, the user group ID is also divided into the actual user group ID, the valid user group ID, and the saved set user group ID. When the process starts, if the file sets the S_ISUID attribute bit, the valid user ID of the process and the saved set user ID are set as the owner ID of the file. If the S_ISUID bit is not set, the saved set user ID is invalid, and the valid user ID is set as the actual user ID. The process group ID is also set likewise, and the difference is that the bit of the detected file attribute S_ISGID.

The following indicates how to set and get the process user ID.

7. S_ISUID or S_ISGID bit judgment

If the S_ISUID bit in the file attribute is 1, the S_ISUID bit is set when the process starts. If the S_ISGID bit in the file attribute is 1, the S_ISGID bit is set when the process starts.

```
#include <unistd.h>
int issetugid (void);
```

Prototype analysis of Function issetuid:

- If either S_ISUID or S_ISGID is set to 1 in the start process, return true. Otherwise, return false.

8. Set the actual process user ID

```
#include <unistd.h>
int setuid(uid_t uid);
```

Prototype analysis of Function setuid:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter **uid** is the process user ID to be set

If the current user is the superuser, i.e., the valid user ID is 0, and the setuid can set the user ID to any ID. Once setting is successful, the actual user ID, valid user ID, and saved set user ID of the process are all set as the new IDs. If the current user is the ordinary user, i.e., the user ID is not 0, only the valid user ID is modified, which can only be modified as the actual user ID or the saved set user ID.

9. Get the actual process user ID

```
#include <unistd.h>
uid_t getuid(void);
```

Prototype analysis of Function getuid:

- Return the actual user ID of the calling process.

10. Set the valid user ID of the process

```
#include <unistd.h>
int seteuid(uid_t euid);
```

Prototype analysis of Function seteuid:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter **euid** is the valid user ID of the process to be set.

If the current user is the superuser, the seteuid can change the valid user ID of the process to any ID. If the current user is the ordinary user, the valid user ID of the process can only be modified as the actual user ID or the saved set user ID.

11. Get the valid user ID of the process

```
#include <unistd.h>
uid_t geteuid(void);
```

Prototype analysis of Function geteuid:

- The function returns the valid user ID of the calling process.

12. Set the process user group ID

```
#include <unistd.h>
int setgid(gid_t gid);
```

Prototype analysis of Function setgid:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *gid* is the user group ID of the process to be set.

If the current user is the superuser, i.e., the user ID is 0, and the setgid can set the user group ID to any group ID. However, once setting is successful, the actual user group ID, valid user group ID, and saved set user group ID of the process are all set as the new group IDs. If the current user is the ordinary user, i.e., the user ID is not 0, only the valid user group ID can be modified, which can only be modified as the actual user group ID or the saved set user group ID.

13. Get the actual user ID of the process

```
#include <unistd.h>
gid_t getgid(void);
```

Prototype analysis of Function getgid:

- The function returns the actual user ID of the calling process.

14. Set the valid user ID of the process

```
#include <unistd.h>
int setegid(gid_t egid);
```

Prototype analysis of Function setegid:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *egid* is the valid user group ID of the process to be set.

If the current user is the superuser, the setegid can set the valid user group ID as any group ID. If the current user is the ordinary user, the valid user group ID can only be modified as the actual user group ID or the saved set user group ID.

15. Get the valid user ID of the process

```
#include <unistd.h>
gid_t getegid(void);
```

Prototype analysis of Function getegid:

- The function returns the valid user ID of the calling process

The following instances indicate how to set and get the process user ID and the user group ID.

Program List 8.8 Setting instances of user ID and user group ID

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    fprintf(stdout, "uid: %d, gid:%d, euid:%d, egid:%d\n",
            getuid(), getgid(), geteuid(), getegid()); /* 打印用户 ID 和组 ID */

    if (setuid(1) != 0) { /* Set user ID */
        fprintf(stderr, "setuid failed\n");
    }

    if (setgid(1) != 0) { /* Set user group ID */
        fprintf(stderr, "setgid failed\n");
    }

    fprintf(stdout, "uid: %d, gid:%d, euid:%d, egid:%d\n",
            getuid(), getgid(), geteuid(), getegid());

    return (0);
}
```

Run the program in the SylixOS Shell:

```
# ./setuid_demo
user and group id before set, uid: 0, gid:0, euid:0, egid:0
setgid failed
user and group id after set, uid: 1, gid:0, euid:1, egid:0
```

It can be seen that the initial valid user ID of the process is 0 (superuser), the setuid(1) call sets the actual user ID and valid user ID of the process to 1, and the valid user ID of the process is not the superuser ID at the moment. When the setuid(1) call is executed, the system finds that the target group ID is not the actual user group ID or the saved set user group ID (the program does not set S_ISUID and S_ISGID, and the saved set user group ID is invalid). Refuse execution, and return failure.

16. Set the extended user group ID of the current process

The superuser authority must be owned to call the function process. Otherwise, return failure.

```
#include <unistd.h>
int setgroups(int groupsun, const gid_t grlist[]);
```

Prototype analysis of Function setgroups:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **groupsun** is the size of Parameter **grlist** array;
- Parameter **grlist** is the array of the extended user group ID.

17. Extended user group ID got

```
#include <unistd.h>
int getgroups(int groupsize, gid_t grlist[]);
```

Prototype analysis of Function setgroups:

- The function returns the number of extended user group IDs of the process.
- Parameter **groupsize** is the size of Parameter **grlist** array. If **groupsize** is less than the number of extended user group IDs, only groupsize user group IDs are filled. 0 represents only statistics of quantity of extended user group ID.
- Parameter **grlist** is the buffer zone array used to save the extended user group ID. 0 represents only statistics of quantity of extended user group ID.

8.3.5 Process control

1. Process quit

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);

#include <unistd.h>
void _exit(int status);
```

Function prototype analysis:

- Parameter **status** is the process return code.

The above three functions are all used for process exit. The difference is that the exit function will use the hook function registered with the agetit function, but _Exit and _exit do not call it. Functions of the _Exit and _exit functions are the same.

2. Register process exit hook

```
#include <stdlib.h>
void atexit(void (*func)(void));
```

Prototype analysis of Function atexit:

- Parameter **func** is the process exit hook function. When the process exits normally (the main function return or call exit function), the hook function is called in reverse order of the atexit registration sequence.

The following example shows how to use the atexit function and the exit function.

Program List 8.9 Process exit instance

```
#include <stdio.h>
#include <stdlib.h>

void exit_hook1()
{
    fprintf(stdout, "in exit_hook1\n");
}

void exit_hook2()
{
    fprintf(stdout, "in exit_hook2\n");
}

int main (int argc, char *argv[])
{
    atexit(exit_hook1);                /* Register the hook
function                               */
    atexit(exit_hook2);

    fprintf(stdout, "this is exit hook demo.\n");

    exit(0);
}
```

Run the program in the SylixOS Shell:

```
# ./exit_demo
this is exit hook demo.
in exit_hook2
in exit_hook1
```

3. Wait for the child process to end

The following function waits for the end of a child process.

```
#include <wait.h>
pid_t wait(int *stat_loc);
```


Prototype analysis of Function wait;

- For success of the function, return child process ID. For failure, return -1 and set the error code.
- Parameter *stat_loc* is the child process exit code.

The following example shows how to wait for a child process with the wait function.

Program List 8.10 wait instance child process

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    int pid;

    sleep(2); /* Wait two seconds to
exit */

    pid = getpid();
    fprintf(stdout, "child process %d exit\n", pid);

    exit(1);
}
```

Program List 8.11 wait instance parent process

```
#include <stdio.h>
#include <spawn.h>
#include <sys/wait.h>

int main (int argc, char *argv[])
{
    pid_t pid;
    int status;

    if (posix_spawn(&pid, "./wait_demo_child",
        NULL, NULL, NULL, NULL) != 0) { /* Startup process
*/
        fprintf (stderr, "create child process failed\n");
        return (-1);
    }

    fprintf(stdout, "create child process %d\n", pid);
}
```

```

        pid = wait(&status);
        /* Wait for the
child process to exit */

    fprintf(stdout, "wait returned, child process: %d, status: %d\n",
           pid, status);

    return (0);
}

```

Run the program in the SylixOS Shell:

```

# ./wait_demo
create child process 25
child process 25 exit
wait returned, child process: 25, status: 0

```

4. Wait for child process state change

```

#include <wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infp, int options);

```

Prototype analysis of Function waitpid:

- If function return is caused due to child process state, return child process ID. If options is set with WNOHANG bit, and the child process state conforming to conditions is not changed, do not wait, and return 0. In other cases, return -1, and set the error code;
- Parameter *idtype* indicates the meaning of Parameter *id*, there are the following situations;

Table 8.4 id meaning

Macro name	Explanation
P_PID	Wait for the child process with process ID equal to Parameter <i>id</i>
P_PGID	Wait for the child process with process group ID equal to Parameter <i>id</i>
P_ALL	Wait for any child process

- The meaning of Parameter *id* is specified by *idtype*, and is used to specify the child process;
- Parameter *infp* returns the received child process signal, which records the information of child process with state changes;

- Parameter **option** is the function option consisting of bit mask, as shown in Table 8.5:

Table 8.5 **option** parameter bit mask

Macro name	Explanation
WNOHANG	If it is 1, the function does not wait, and returns directly when there is no child process state change
WUNTRACED	If it is 1, return when the child process enters the stopped state. Otherwise, return only when the process exits

5. Wait for appointed child process state change

```
#include <wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Prototype analysis of Function waitpid:

- If function return is caused due to child process state, return child process ID. If options is set with WNOHANG bit, and the child process state is not changed, do not wait, and return 0. In other cases, return -1, and set the error code;
- Parameter **pid** can have the following cases:
 - ◆ **pid** > 0: indicating wait for the child process with process number of **pid**;
 - ◆ **pid** == 0: indicating wait for the child process in the same group with the calling process;
 - ◆ **pid** < -1: indicating wait for the child process with the process group ID as the absolute value of pid.
- Parameter **stat_loc** is the child process exit code;
- Parameter **option** is the function option, consisting of the bit mask, as shown in Table 8.5.

The following function shows how to use the waitpid function. Create two child processes in the program in succession, set different group Ids for the child process respectively, and call waitpid function to wait for the child process.

Program List 8.12 waitpid instance

```
#include <stdio.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

int main (int argc, char *argv[])
```

```
{
pid_t    pid;
int      status;

if (posix_spawn(&pid, "./wait_demo_child",
                NULL, NULL, NULL, NULL) != 0) { /* Startup process
*/
    fprintf(stderr, "create child process failed\n");
    return (-1);
}

fprintf(stdout, "create child process %d\n", pid);
setpgid(pid, 10); /* Set process group ID
*/
fprintf(stdout, "waiting for group id 10...\n");
pid = waitpid(-10, &status, 0); /* Wait for the child
process to exit */
fprintf(stdout, "waitpid returned, child process: %d, status: %d\n",
        pid, status);

if (posix_spawn(&pid, "./wait_demo_child",
                NULL, NULL, NULL, NULL) != 0) { /* Startup process
*/
    fprintf("create child process failed\n");
    return (-1);
}

fprintf(stdout, "create child process %d\n", pid);

setpgid(pid, 11); /* Set process group ID
*/
fprintf(stdout, "waiting for group id 11...\n");
pid = waitpid(-11, &status, 0); /* Wait for the child
process to exit */

fprintf(stdout, "waitpid returned, child process: %d, status: %d\n",
        pid, status);

return (0);
}
```

Run the program under the SylixOS Shell: It can be seen that `waitpid` does not return when the child process group ID is different from the absolute value of **`pid`** parameter passed by the parent process by calling the `waitpid` function.

```
# ./waitpid_demo
create child process 9
waiting for group id 10...
child process 9 exit
waitpid returned, child process: 9, status: 0
create child process 10
waiting for group id 11...
child process 10 exit
```

6. Get process resource status

```
#include <sys/resource.h>
int getrusage(int who, struct rusage *r_usage);
```

Prototype analysis of Function `getrusage`:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter **`who`** is the object of resources obtained, and the value is shown in Table 8.6:

Table 8.6 Value of `who` parameter

Macro name	Explanation
RUSAGE_SELF	Get the process resource status
RUSAGE_CHILDREN	Get the child process resource status

- Parameter **`r_usage`** returns process resource status. The `rusage` structure is defined as follows:

```
struct rusage {
    struct timeval    ru_utime;          /* User time          */
    struct timeval    ru_stime;          /* System time        */
    long              ru_maxrss;
#define ru_first      ru_ixrss
    long              ru_ixrss;
    long              ru_idrss;
    long              ru_isrss;
    long              ru_minflt;
    long              ru_majflt;
    long              ru_nswap;
    long              ru_inblock;
```

```

long          ru_oublock;
long          ru_msgsnd;
long          ru_msgrcv;
long          ru_nsignals;
long          ru_nvcsw;
long          ru_nivcsw;
#define ru_last      ru_nivcsw
};

```

Note: currently, SylixOS only uses two fields of `ru_utime` and `ru_stime`. Other fields are reserved for subsequent expansion.

7. Get process time

```

#include <sys/times.h>
clock_t times(struct tms *ptms);

```

Prototype analysis of Function times:

- The function returns the current time of the system;
- Parameter *ptms* is the time status of the process and its child processes. The `tms` structure is defined as follows.

```

struct tms {
    clock_t tms_utime;           /* Process user time      */
    clock_t tms_stime;           /* Process system time    */
    clock_t tms_cutime;         /* Child process user time */
    clock_t tms_cstime;         /* Child process system time */
    /*
};

```

It shall be noted that if Parameter *ptms* is NULL, set `errno` to `EINVAL`, and return the system time.

The following pseudocode shows how to get the program running time:

```

clock_t      start, end, run;
struct tms   tm_start, tm_end;

start = times(&tm_start);

/*
 * Program
 */
...

end = times(&tm_end);

```

```
run = end - start;
```

8.3.6 Process environment

1. Get environment variable

```
#include <stdlib.h>
char *getenv(const char *name);
```

Prototype analysis of Function getenv:

- For success of the function, return the found character string character string pointer. For failure, return NULL and set the error code;
- Parameter *name* is the name of the environment variable.

Calling getenv can get the current system environment variable. It shall be noted that if the environment variable exists but has no associated value, the function will return an empty character string, that is to say, the first character of the character string is '\0'.

2. Set the environment variable

```
#include <stdlib.h>
int putenv(char *string);
```

Prototype analysis of Function putenv:

- For success of the function, return 0. For failure, return non-0 value and set the error number;
- Parameter *string* is an environment variable setting character string, of which the format is: name=value. If the name already exists, the original definition will be deleted.

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

Prototype analysis of Function setenv:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter *name* is the name of the environment variable set;
- Parameter *value* is the value of the environment variable;
- Parameter *overwrite* indicates whether the original environment variable is overwritten when the environment variable already exists. 1 indicates overwriting, and 0 indicates not overwriting.

Both the putenv function and the setenv function can be used to set the system environment variable. The difference is that setenv can set the environment variable in a more flexible way.

3. Clear environment variable


```
#include <stdlib.h>
int unsetenv(const char *name);
```

Prototype analysis of Function unsetenv:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **name** is the name of the environment variable to be cleared;

The following example shows how to use the environment variable function.

Program List 8.13 Environment variable instance

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    const char *env_value = 0;
    const char *env_name = "ENV_DEMO";

    env_value = getenv(env_name);
    if (env_value) {
        fprintf(stdout, "value of %s before setenv is %s\n",
                env_name, env_value);
    } else {
        fprintf(stderr, "value of %s is not setted before setenv\n",
                env_name);
    }

    env_value = "test_value";
    if (setenv(env_name, env_value, 0) != 0) {
        fprintf(stderr, "setenv failed\n");
    } else {
        fprintf(stdout, "set value of %s to %s\n", env_name, env_value);
    }

    env_value = getenv(env_name);
    if (env_value) {
        fprintf(stdout, "value of %s after setevn is %s\n",
                env_name, env_value);
    } else {
        fprintf(stderr, "value of %s is not setted after setenv\n",
                env_name);
    }
}
```

```

if (unsetenv(env_name) != 0) {
    fprintf(stderr, "unsetenv failed\n");
} else {
    fprintf(stdout, "unset value of %s\n", env_name);
}

env_value = getenv(env_name);
if (env_value) {
    fprintf(stdout, "value of %s after unsetenv is %s\n",
            env_name, env_value);
} else {
    fprintf(stderr, "value of %s is not setted after unsetenv\n",
            env_name);
}

return (0);
}

```

Run the program in SylixOS Shell, and the results are as follows:

```

# ./env_demo
value of ENV_DEMO is not setted before setenv
set value of ENV_DEMO to test_value
value of ENV_DEMO after setenv is test_value
unset value of ENV_DEMO
value of ENV_DEMO is not setted after unsetenv

```

8.4 SylixOS process API

In addition to the POSIX standard-compliant process API, SylixOS also provides the process operation function conforming to UNIX standard. The POSIX standard process API is recommended for writing programs, but it is more convenient to use this part of the function in some cases.

8.4.1 Create processes by using SylixOS API

1. Create process by using the spawn function

The difference between the spawn function and the exec function is that: the exec series function will not create the new process, and can only execute the new executable file in the existing process environment. The spawn series function can choose to execute the executable file in the existing process environment or choose to create the new child process.

```

#include <process.h>
int spawnl(int mode, const char *path, const char *argv0, ...);

```

```

int spawnle(int mode, const char *path, const char *argv0, ...);
int spawnlp(int mode, const char *file, const char *argv0, ...);
int spawnv(int mode, const char *path, char * const *argv);
int spawnve(int mode, const char *path, char * const *argv, char * const *envp);
int spawnvp(int mode, const char *file, char * const *argv);
int spawnvpe(int mode, const char *file, char * const *argv, char * const *envp);

```

Function prototype analysis:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter *mode* is the process creation mode, and the value is shown in Table 8.7;

Table 8.7 Table of process creation mode

Macro name	Explanation
P_WAIT	Create the new child process, and calling thread waits for the child process to exit and continues execution
P_NOWAIT	Create the new child process, and calling thread does not wait for the child process to exit
P_OVERLAY	Do not create the new child process, and run the new program in the current process space

- Parameter *path* is the executable file path;
- Parameter *argv0* is the first command line parameter, which is generally the command name;
- Parameter *file* is the executable file name. What's different from Parameter *path* is that it does not have the directory, and the application loader search the file at the specified path. The search paths of application dynamic library in SylixOS are as follows in sequence:
 - ◆ Shell current directory, but not the application directory;
 - ◆ Search path included in PATH environment.
- Parameter...is the variable parameter, which represents the remaining parameters in the command line, and the command line parameter is ended with 0. In the execl function, there is an array of environment variables behind the command line parameter which ends with 0. The array ends with 0. See instructions for the *envp* parameter;
- Parameter *argv* is the array of character strings consisting of command line parameter. The array starts with the executable file name and ends with 0.

- Parameter **envp** is the set of process environment variable character strings preset. The array ends with 0.

The following instance shows how to create the new process and execute the executable program with the spawn series function.

Program List 8.14 Instance of the spawn function

```
#include <stdio.h>
#include <process.h>

int main (int argc, char *argv[])
{
    char *cmd[] = { "child_process", "spawnve demo", (char *)0 };
    char *env[] = { "PARENT=spawnve_demo", (char *)0 };

    fprintf(stdout, "before spawnve\n");
    spawnve(P_WAIT, "./child_process", cmd, env);
    fprintf(stdout, "after spawnve\n");

    return (0);
}
```

Run the program in the SylixOS Shell:

```
# ./spawn_demo
before spawnve
child process, spawnve demo
environment variable PARENT = spawnve_demo
after spawnve
```

It can be seen from the instance that when the **mode** is not P_OVERLAY, the spawn function creates the new child process, and the parent process continues to execute. Therefore, the final print statement of the spawn function is valid.

8.4.2 SylixOS process control API

1. Set the current process as the daemon

```
#include <unistd.h>
int daemon(int nochdir, int noclose);
```

Prototype analysis of Function daemon:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter **nochdir** indicates whether to switch the current working directory of the process to the root directory "/". 0 indicates switching, while others indicate no switching;

- Parameter ***noclose*** indicates whether to redirect standard input, standard output and standard error output to "/dev/null" file. 0 means redirect, while others mean no redirect.

The following instance shows how to use the daemon function.

Program List 8.15 Daemon instance

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    fprintf(stdout, "before daemon\n");
    daemon(0, 0);
    fprintf(stdout, "after daemon\n");

    while (1) {
        sleep(1);
    }

    return (0);
}
```

Run the program in the SylixOS Shell:

```
# ./daemon_demo&
# before daemon
#
```

It can be seen from the running results that after the function daemon is called, the print statement is invalid, and the ps command is used to check the running status of the program.

```
# ps
```

NAME	FATHER	PID	GRP	MEMORY	UID	GID	USER
kernel	<orphan>	0	0	0	0	0	root
daemon_demo	<orphan>	33	33	106496	0	0	root

```
total vprocess : 2
```

2. Wait for appointed child process state change

```
#include <wait.h>
```

```
pid_t wait3(int *stat_loc, int options, struct rusage *prusage);
```

Prototype analysis of Function wait3;

- If changes in child process state cause function return, return 0. If **options** is set with WNOHANG bit, and the child process state is not changed, do not wait, and return 0. In other cases, return -1, and set the error number;
- Parameter **stat_loc** is the child process exit code;
- Parameter **option** is the function option consisting of bit mask, as shown in Table 8.5:
- Parameter **prusage** is the resource service conditions of the child process.

3. Wait for appointed child process state change

```
#include <wait.h>
pid_t wait4(pid_t pid, int *stat_loc, int options, struct rusage *prusage);
```

Prototype analysis of Function wait4;

- If function return is caused due to child process state, return child process ID. If **options** is set with WNOHANG bit, and the child process state is not changed, do not wait, and return 0. In other cases, return -1, and set the error code;
- Parameter **pid** can have the following cases:
 - ◆ **pid > 0**: indicating wait for the child process with process number of pid;
 - ◆ **pid == 0**: indicating wait for the process in the same group with the calling process;
 - ◆ **pid < -1**: indicating wait for the process with the process group ID as the absolute value of pid.
- Parameter **stat_loc** is the child process exit code;
- Parameter **option** is the function option, consisting of the bit mask, as shown in Table 8.5;
- Parameter **prusage** is the resource service conditions of the child process.

The following example shows how to use the wait4 function;

Program List 8.16 Use of the wait4 function

```
#include <stdio.h>
#include <unistd.h>
#include <spawn.h>
#include <wait.h>

int main (int argc, char *argv[])
```

```

{
pid_t      pid;
int        status;
struct rusage  rusage;

if (posix_spawn(&pid, "/apps/wait_demo_child/wait_demo_child",
                NULL, NULL, NULL, NULL) != 0) { /* Startup process */
    fprintf("create child process failed\n");
    return (-1);
}

fprintf(stdout, "create child process %d\n", pid);
setpgid(pid, 10); /* Set process group ID
*/
fprintf(stdout, "waiting for group id 10...\n");
pid = wait4(-10, &status, 0, &rusage); /* Wait for the child
process to exit */
fprintf(stdout, "waitpid returned, child process: %d, status: %d\n"
        "utime: %dus, stime: %dus\n",
        pid, status,
        (int)(rusage.ru_utime.tv_sec * 1000000 + rusage.ru_utime.tv_usec),
        (int)(rusage.ru_stime.tv_sec * 1000000 + rusage.ru_stime.tv_usec));

if (posix_spawn(&pid, "/apps/wait_demo_child/wait_demo_child",
                NULL, NULL, NULL, NULL) != 0) { /* Startup process */
    printf ("create child process failed\n");
    return (-1);
}

fprintf(stdout, "create child process %d\n", pid);
setpgid(pid, 11); /* Set process group ID */
fprintf(stdout, "waiting for group id 11...\n");
pid = wait4(-10, &status, 0, &rusage); /* Wait for the child process to
exit */
fprintf(stdout, "waitpid returned, child process: %d, status: %d\n"
        "utime: %dus, stime: %dus\n",
        pid, status,
        (int)(rusage.ru_utime.tv_sec * 1000000 + rusage.ru_utime.tv_usec),
        (int)(rusage.ru_stime.tv_sec * 1000000 + rusage.ru_stime.tv_usec));

return (0);
}

```

Run the program in SylixOS Shell, it can be seen that the wait4 function does not return when the child process group ID is different from the absolute value of *pid* parameter passed by the parent process by calling the wait4 function.

```
# ./wait4_demo
create child process 8
waiting for group id 10...
child process 8 exit
waitpid returned, child process: 8, status: 0
utime: 40000us, stime: 0us
create child process 9
waiting for group id 11...
child process 9 exit
```


Chapter 9 Inter-Process Communication

9.1 Definition of IPC

The Inter-Process Communication (IPC) refers to the technique or method of transferring data or signals between two or more processes. The process is the smallest unit of resource allocation in a computer system. Each process has its own part of a separate system resource that is isolated from each other. The IPC mechanism is required to enable different processes to access resources and coordinate work.

Common interprocess communication methods include: pipe, named message queue, named semaphore, shared memory, and signals etc.

9.2 Anonymous Pipe

The pipe is a mode of SylixOS IPC. Similar to the transmission pipe in the real world, the pipe has two ports: the read port and the write port, and only allows the data to flow from the write port to the read port. Therefore, the pipe is a streaming device.

The pipes are classified as the anonymous pipe and the named pipe fifo.

The pipe function is needed for creating an anonymous pipe. The output parameters of the pipe function are two file descriptors: one for the read port file and the other for the write port file. After creating the anonymous pipe, usually create a child process using the `posix_spawn` family or `spawn` family function. As the child process inherits the file descriptor of the parent process, both the child process and the parent process can read and write the anonymous pipe by means of the read and write functions.

Both the child and parent processes have two file descriptors for the anonymous pipe: the read port file and the write port file, but this does not mean that the anonymous pipe can carry out the full-duplex communication between the parent and child processes. The anonymous pipe has only two ports: read and write, and only allows the data to flow from the write port to the read port, and thus the anonymous pipe can only carry out the half-duplex communication. If the full duplex communication is required, two anonymous pipe are needed creating.

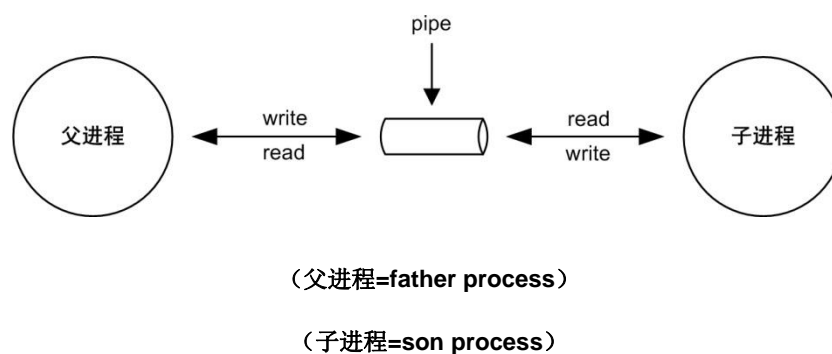


Figure 9.1 Anonymous Pipe

9.2.1 Operate Anonymous Pipe

1. Create Anonymous Pipe

```
#include <unistd.h>
int  pipe(int  iFd[2]);
int  pipe2(int  iFd[2], int  iFlag);
```

Function prototype analysis:

- The above function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The output parameter *iFd* is used to record the two file descriptors for the anonymous pipe: *iFd*[0] is the read port file descriptor and *iFd*[1] is the write port file descriptor.
- The parameter *iFlag* is the file flag for the anonymous pipe that can be combined with a macro of 0 or less.

Table 9.1 Parameter *iFlag*

Macro Name	Meaning
O_NONBLOCK	The read-write pipe is a non-blocking operation.
O_CLOEXEC	The pipe will be closed when exec occurs.

2. Read and Write Anonymous Pipe

As the output parameter of the pipe function has two file descriptors: the first is the read port file descriptor and the second is the write port file descriptor. Therefore, the read and write operations of the anonymous pipe can use the standard file read and write functions.

3. Wait for Anonymous Pipe

When the anonymous pipe is empty or full, the read or write anonymous pipe operation will be blocked (unless the parameter *iFlag* specifies the O_NONBLOCK option when creating the anonymous pipe using the pipe2 function); and call the select function when Wait for the anonymous pipe to read or write.

4. Close Anonymous Pipe

Use the standard file close function to close an anonymous pipe. As the anonymous pipe has two file descriptors, it is necessary to close the anonymous pipe using the close function to close the two file descriptors of the anonymous pipe.

Meanwhile as the child process (if existing) inherits the two file descriptors of the anonymous pipe, the child process also needs to close the two file descriptors of the anonymous pipe.

The following program shows how to use the anonymous pipe. The parent process calls the `posix_spawn` function to create the child process, closes the standard input of the child process and the pipe write port of the child process, and copies the read port file descriptor of the pipe to the standard input of the child process. The child process reads data from the pipe read port every 1 second.

Program List 9.1 Child Process of Anonymous Pipe

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    char    buf[64] = {0};
    int     i;

    for (i = 0; i < 10; i++) {
        read(STDIN_FILENO, buf, sizeof(buf));
        fprintf(stdout, "buf:%s\n", buf);
        sleep(1);
    }
    close(STDIN_FILENO);

    return (0);
}
```

Program List 9.2 Parent Process of Anonymous Pipe

```
#include <stdio.h>
#include <spawn.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>

#define SEND_STR    "From parent."

int main (int argc, char *argv[])
{
    posix_spawn_file_actions_t  file_actions;
    posix_spawnattr_t          spawnattr;
    pid_t                      pid;
    int                        fd[2];
    int                        ret, i;

    char    *cmd[]      = {"child_process", (char *)0};
```

```
ret = pipe(fd);
if (ret < 0) {
    fprintf(stderr, "pipe error.\n");
    return (-1);
}

/*
 * Initialize process attributes
 */
if (posix_spawnattr_init(&spawnattr) != 0) {
    fprintf(stderr, "init posix_spawnattr_t failed\n");
    return (-1);
}

/*
 * Initialize file operations
 */
if (posix_spawn_file_actions_init(&file_actions) != 0) {
    fprintf(stderr, "init posix_spawn_file_actions_t failed\n");
    return (-2);
}

/*
 * Close the standard input of the child process and the write end of the
pipe
 */
posix_spawn_file_actions_addclose(&file_actions, STDIN_FILENO);
posix_spawn_file_actions_addclose(&file_actions, fd[1]);
/*
 * Copy the reading end to the standard input of the child process
 */
posix_spawn_file_actions_adddup2(&file_actions, fd[0], STDIN_FILENO);

if (posix_spawn(&pid, "./child_process",
                &file_actions, &spawnattr, cmd, NULL) != 0) {
    posix_spawnattr_destroy(&spawnattr);
    posix_spawn_file_actions_destroy(&file_actions);

    return (-6);
}

close(fd[0]);
```

```

    for (i = 0; i < 10; i++) {
        write(fd[1], SEND_STR, strlen(SEND_STR));
        sleep(1);
    }

    close(fd[1]);
    wait(NULL);
    posix_spawnattr_destroy(&spawnattr);
    posix_spawn_file_actions_destroy(&file_actions);

    return (0);
}

```

9.3 Named Pipe

The anonymous pipe is a file but it does not exist in the file system, and thus the anonymous pipe can only be used for the communication between parent and child processes. For the irrelevant process, there is no way to communicate using any anonymous pipe because there is no inheritance of file descriptors, but you can use the named pipe to communicate.

Use the `mkfifo` function to create a named pipe and the `mkfifo` function specifies the device file path for the named pipe, and other processes can use the `open` function of standard file to open the named pipe. You can read and write the named pipe by means of the read-write functions.

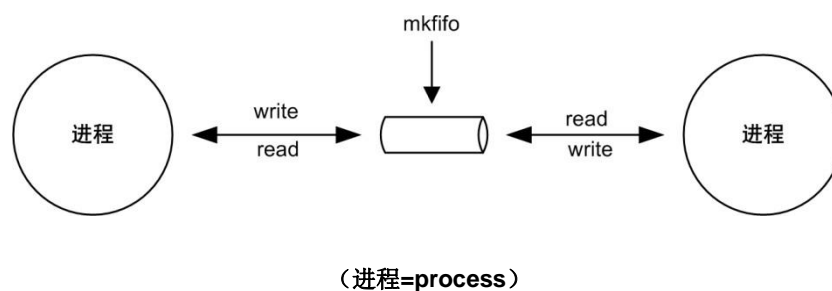


Figure 9.2 Named Pipe

9.3.1 Operate Named Pipe

1. Create Named Pipe

```

#include <unistd.h>
int  mkfifo(const char  *pcFifoName, mode_t  mode);

```

Prototype analysis on function `mkfifo`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *pcFifoName* specifies the device file path for the named pipe;
- The parameter *mode* specifies the device file mode for the named pipe, which is the same as the open function mode.

Creating a named pipe is similar to creating a file, and thus the pathname of the named pipe exists in the file system.

2. Open Named Pipe

As the named pipe exists in the file system, you can open the function using the standard file to open the named pipe. When the open function is called, the parameter *iFlag* is the open flag of the named pipe. In addition to the file flags that can use the anonymous pipe, the parameter *iFlag* can also use the following flags.

Table 9.2 File Flags

Macro Name	Meaning
O_RDONLY	Open the pipe in the read-only mode
O_WRONLY	Open the pipe in the write-only mode
O_RDWR	Open the pipe in the read and write modes

3. Read and Write Named Pipe

The read and write operations of the named pipe use the read and write functions of standard file respectively.

4. Wait for Named Pipe

When the named pipe is empty or full, the read or write named pipe operation will be blocked (unless the parameter *iFlag* specifies the O_NONBLOCK option when creating the named pipe by means of the open function); and call the select function when waiting for the named pipe to read or write.

5. Close Named Pipe

Close a named pipe by using the close function of standard file.

6. Unlink Named Pipe

Use the unlink function of standard file to unlink the named pipe.

The following program shows how to use the named pipe. The program consists of client programs and server programs. Create a named pipe file "/dev/fifo" on the server program and write data to the pipe. The client-side program opens the file and reads the data from the pipe.

Program List 9.3 Named Pipe Client-side

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define FIFO_NAME "/dev/fifo"
#define SEND_STR "From server."

int main (int argc, char *argv[])
{
    int    fd;
    char   buf[64] = {0};
    int    i;

    fd = open(FIFO_NAME, O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "open fifo error.\n");
        return (-1);
    }

    for (i = 0; i < 10; i++) {
        read(fd, buf, strlen(SEND_STR));
        fprintf(stdout, "read \"%s\" from fifo.\n", buf);
        sleep(1);
    }

    close(fd);

    return (0);
}
```

Program List 9.4 Named Pipe Server

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define FIFO_NAME "/dev/fifo"
#define SEND_STR "From server."

int main (int argc, char *argv[])
{
    int    ret, i;
    int    fd;
```

```
ret = mkfifo(FIFO_NAME, 0777);
if (ret < 0) {
    perror("mkfifo");
    fprintf(stderr, "mkfifo error.\n");
    return (-1);
}

fd = open(FIFO_NAME, O_RDWR);
if (fd < 0) {
    fprintf(stderr, "open fifo error.\n");
    return (-1);
}

for (i = 0; i < 10; i++) {
    write(fd, SEND_STR, strlen(SEND_STR));
    fprintf(stdout, "write \"%s\" to fifo.\n", SEND_STR);
    sleep(1);
}

sleep(3);
close(fd);
unlink(FIFO_NAME);

return (0);
}
```

9.4 POSIX Named Semaphore

We have introduced the use of POSIX anonymous semaphore in Section 7.4. POSIX anonymous semaphore can only be used for inter-thread communication within the same process. To achieve synchronization between processes, you can use the POSIX named semaphore.

A POSIX named semaphore must be created or opened after the `sem_open` function is called.

When a POSIX named semaphore is used, the `sem_close` function should be called to close it; when a POSIX named semaphore is no longer useful, the `sem_unlink` function should be called to unlink it, and the SylixOS will recycle the kernel resources that the semaphore has consumed.

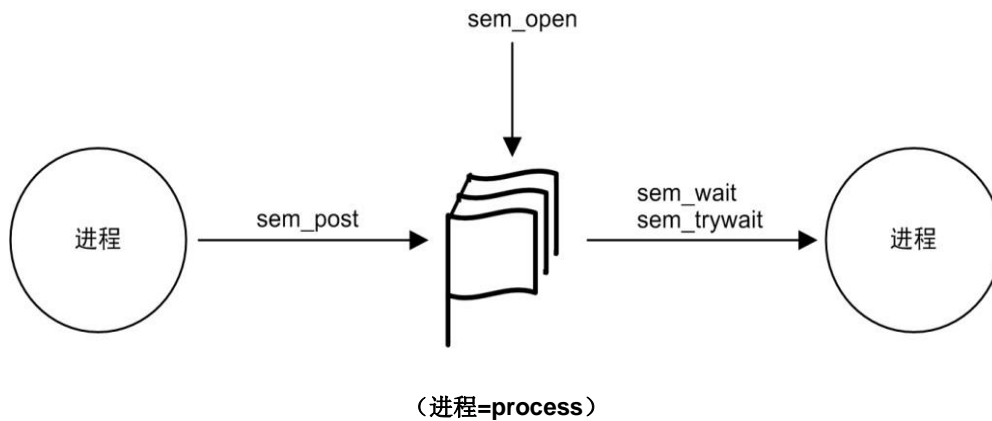


Figure 9.3 POSIX Named Semaphore

9.4.1 Named Semaphore

1. Create and Open Named Semaphore

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int flag, ...);
```

Prototype analysis on function `sem_open`:

- This function returns a pointer to a `sem_t` type when it succeeds, returns `NULL` when it fails, and sets the error number.
- The parameter ***name*** is the name of the POSIX named semaphore;
- The parameter ***flag*** is the open option for the POSIX named semaphore (`O_CREAT` and `O_EXCL...`);
- The parameter `...` is a variable parameter that can normally specify the open mode (mode and value).

If you need to create a POSIX named semaphore, open the option and add `O_CREAT`, and the variable parameter should specify the value of mode and value.

If you need to open an existing POSIX named semaphore, the open option cannot contain the `O_CREAT` option flag.

2. Close Named Semaphore

```
#include <semaphore.h>
int sem_close(sem_t *psem);
```

Prototype analysis on function `sem_close`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***psem*** is a pointer to the POSIX named semaphore.

Calling the `sem_close` function reduces the use count of a named semaphore, but does not unlink a named semaphore. It is important to note that if the `sem_close` function is called to close an anonymous semaphore, it will return `-1` and set `errno` as `EINVAL`.

3. Unlink Named Semaphore

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

Prototype analysis on function `sem_unlink`:

- The function returns `0` when succeeds and returns `-1` when fails, and sets the error number;
- The parameter ***name*** is the name of the POSIX named semaphore.

The `sem_unlink` function unlinks a named semaphore that is no longer used, and frees system resources. The `sem_unlink` function firstly determines the use count of semaphore. If the count reaches `0`, then the semaphore is unlinked, and if no `0` is reached, the error returns, and `errno` is set to `EBUSY`.

The following program realizes the IPC through named semaphore. The server program waits for the semaphore `sem`. When the semaphore waiting is unlinked, new data can be read in the named pipe. After reading the data, the server program sends another semaphore `sem1` to the client-side indicating that the reading data is complete. The client program firstly writes the new data to the pipe, then sends the semaphore `sem` to the server and waits for the semaphore `sem1` (the server read operation completed).

Program List 9.5 Named Semaphore Client-side

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>

#define FIFO_NAME "/dev/fifo"
#define SEND_STR "server."
#define SEM_FILE "sem_named"
#define SEM_FILE1 "sem_named1"

int main (int argc, char *argv[])
{
    int fd;
    int i;
    sem_t *sem, *sem1;
```

```

    fd = open(FIFO_NAME, O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "open fifo error.\n");
        return (-1);
    }

    sem = sem_open(SEM_FILE, 0);
    if (sem == SEM_FAILED) {
        fprintf(stderr, "sem_open error.\n");
        return (-1);
    }

    sem1 = sem_open(SEM_FILE, 0);
    if (sem1 == SEM_FAILED) {
        fprintf(stderr, "sem_open error.\n");
        return (-1);
    }

    for (i = 0; i < 10; i++) {
        write(fd, SEND_STR, strlen(SEND_STR));
        fprintf(stdout, "write \"%s\" to fifo.\n", SEND_STR);
        sem_post(sem);
        sem_wait(sem1);
    }

    close(fd);
    sem_close(sem);
    sem_close(sem1);

    return (0);
}

```

Program List 9.6 Named Semaphore Server

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>

#define FIFO_NAME "/dev/fifo"
#define SEND_STR "server."
#define SEM_FILE "sem_named"
#define SEM_FILE1 "sem_named1"

```

```
int main (int argc, char *argv[])
{
    int      ret, i;
    int      fd;
    sem_t    *sem, *sem1;
    char     buf[64] = {0};

    ret = mkfifo(FIFO_NAME, 0777);
    if (ret < 0) {
        perror("mkfifo");
        fprintf(stderr, "mkfifo error.\n");
        return (-1);
    }

    fd = open(FIFO_NAME, O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "open fifo error.\n");
        return (-1);
    }

    sem = sem_open(SEM_FILE, O_CREAT, 0644, 0);
    if (sem == SEM_FAILED) {
        fprintf(stderr, "sem_open error.\n");
        return (-1);
    }

    sem1 = sem_open(SEM_FILE, O_CREAT, 0644, 0);
    if (sem1 == SEM_FAILED) {
        fprintf(stderr, "sem_open error.\n");
        return (-1);
    }

    for (i = 0; i < 10; i++) {
        sem_wait(sem);
        read(fd, buf, strlen(SEND_STR));
        fprintf(stdout, "read \"%s\" from fifo.\n", buf);
        sem_post(sem1);
    }

    close(fd);
    unlink(FIFO_NAME);
    sem_close(sem);
    sem_close(sem1);
}
```

```

sem_unlink(SEM_FILE);
sem_unlink(SEM_FILE1);

return (0);
}

```

9.5 POSIX Named Message Queue

The type of handle to the POSIX named message queue is `mqd_t`. When it is used, you need to define a variable of type `mqd_t`: `mqd_t mqd`:

```
mqd_t mqd;
```

A POSIX named message queue must be created or opened after the `mq_open` function is called. The receive message can call the `mq_receive` function and the message can be sent using the `mq_send` function.

When a POSIX named message queue completes using, the `mq_close` function should be called to close it. When a POSIX named message queue no longer has any purpose, it should be unlinked by calling the `mq_unlink` function, and the SylixOS will recycle the kernel resources that the message queue has consumed.

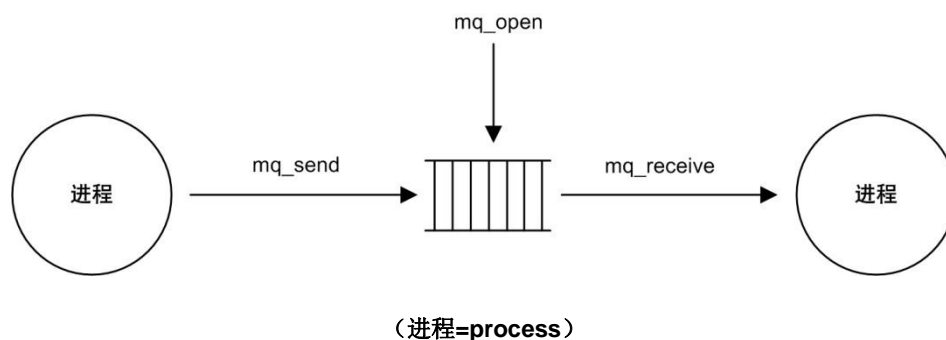


Figure 9.4 POSIX Named Message Queue

9.5.1 Attribute Block of Named Message Queue

Creating a POSIX named message queue requires an attribute block of POSIX named message queue. The attribute block type of the POSIX named message queue is `struct mq_attr`, which is defined as follows:

```

typedef struct mq_attr {
    long          mq_flags;           /* Message queue file flag */
    long          mq_maxmsg;         /* The maximum number of messages
that the message queue can hold */
    long          mq_msgsize;        /* The maximum length of a message
message queue's single message */
}

```

```

long          mq_curmsgs;          /* The number of messages in
the current message queue        */
} mq_attr_t;

```

When using, you need to define a struct `mq_attr` structure variable, such as:

```
struct mq_attr mqattr;
```

Because the POSIX does not define the operation function of the attribute block of POSIX named message queue, the member of the struct `mq_attr` structure needs to be assigned to use, and the sample code is as follows:

```
struct mq_attr mqattr = {O_RDWR, 128, 64, 0};
```

9.5.2 Named Message Queue

1. Create and Open Named Message Queue

```

#include <mqqueue.h>
mqd_t  mq_open(const char *name, int flag, ...);

```

Prototype analysis on function `mq_open`:

- This function returns a handle to a named message queue when it succeeds, returns `MQ_FAILED` when it fails and sets the error number;
- The parameter ***name*** is the name of the named message queue;
- The parameter ***flag*** is the open option for the named message queues (`O_CREAT` and `O_EXCL...`);
- The parameter `...` is a variable parameter that specifies the mode and attribute block (mode and `pmqattr`) of the message queue.

If you need to create a POSIX named message queue, open the option and add `O_CREAT`, and the variable parameter should specify mode and `pmqattr`. When the `pmqattr` is `NULL`, the default attribute is used. Create a command message queue as follows:

```

mqd_t  mq;
mq = mq_open("mq_test", O_RDWR | O_CREAT, 0666, NULL);

```

If you open an existing POSIX named message queue, open the option but not add `O_CREAT`.

The default attributes are defined as follows:

```
mq_attr_t mq_attr_default = {O_RDWR, 128, 64, 0};
```

The message queue can hold 128 messages, and the maximum length of a single message is 64 bytes.

2. Get and Set Named Message Queue Attributes

```
#include <mqueue.h>
int mq_getattr(mqd_t mqd, struct mq_attr *pmqattr);
```

Prototype analysis on function `mq_getattr`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *mqd* is the handle to the POSIX named message queue;
- The output parameter *pmqattr* is used to receive the attributes of the POSIX named message queue.

```
#include <mqueue.h>
int mq_setattr(mqd_t mqd, const struct mq_attr *pmqattrNew,
               struct mq_attr *pmqattrOld);
```

Prototype analysis on function `mq_setattr`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *mqd* is the handle to the POSIX named message queue;
- The parameter *pmqattrNew* points to an attribute block of POSIX named message queue, which is a new attribute that needs to be set.
- The output parameter *pmqattrOld* is used to receive the current attribute of the POSIX named message queue, which can be NULL.

3. Send Message to Named Message Queue

```
#include <mqueue.h>
int mq_send(mqd_t mqd, const char *msg, size_t msglen,
            unsigned msgprio);
int mq_timedsend(mqd_t mqd, const char *msg, size_t msglen,
                 unsigned msgprio, const struct timespec *abs_timeout);
int mq_reltimedsend_np(mqd_t mqd, const char *msg, size_t msglen,
                      unsigned msgprio, const struct timespec *rel_timeout);
```

Prototype analysis on the above three functions:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *mqd* is the handle to the POSIX named message queue;
- The parameter *msg* points to the message buffer that needs to be sent (a const char type pointer);

- The parameter *msglen* is the length of the message to be sent;
- The parameter *msgprio* is the priority of the message to be sent;
- The parameter *abs_timeout* is the absolute timeout period that the sender thread needs to wait when the message queue is full.
- The parameter *rel_timeout* is the relative timeout period that the sender thread needs to wait when the message queue is full.

The `mq_timedsend` function is the version of the `mq_send` function waiting for timeout. The *abs_timeout* is the absolute timeout waiting period (the absolute time refers to some point in the future).

The `mq_reltimedsend_np` function is the non-POSIX standard version of the `mq_timedsend` function, with the parameter *rel_timeout* as the waiting relative timeout (the relative time refers to a time interval starting at the current time).

4. Receive Message of Named Message Queue

```
#include <mqueue.h>
ssize_t mq_receive(mqd_t mqd, char *msg, size_t msglen,
                  unsigned *pmsgprio);
ssize_t mq_timedreceive(mqd_t mqd, char *msg, size_t msglen,
                       unsigned *pmsgprio,
                       const struct timespec *abs_timeout);
ssize_t mq_reltimedreceive_np(mqd_t mqd, char *msg, size_t msglen,
                              unsigned *pmsgprio,
                              const struct timespec *rel_timeout);
```

Prototype analysis on the above three functions:

- The above function returns the length of the received message when it succeeds, and returns -1 and sets the error number when fails;
- The parameter *mqd* is the handle to the POSIX named message queue;
- The parameter *msg* points to the message buffer used to receive messages (a char type pointer);
- The parameter *msglen* is the length of message buffer;
- The output parameter *pmsgprio* is used to receive message priority;
- The parameter *abs_timeout* is the absolute timeout period that the receiver thread needs to wait when the message queue is empty.
- The parameter *rel_timeout* is the relative timeout that the receiver thread needs to wait when the message queue is empty.

The `mq_timedreceive` is the version of the `mq_receive` that waits for timeout, and the ***abs_timeout*** is the absolute waiting timeout.

The `mq_timedreceive_np` is the non-POSIX standard version of `mq_timedreceive`, and the parameter ***rel_timeout*** is the relative timeout waiting period.

5. Notification Signal of Registering Named Message Queue for Reading

```
#include <mqueue.h>
int mq_notify(mqd_t mqd, const struct sigevent *pnotify);
```

Prototype analysis on function `mq_notify`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***mqd*** is the handle to the POSIX named message queue;
- The parameter ***pnotify*** points to the variable of a struct `sigevent` signal event type (see the signal system in Chapter 10).

6. Close Named Message Queue

```
#include <mqueue.h>
int mq_close(mqd_t mqd);
```

Prototype analysis on function `mq_close`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***mqd*** is the handle to the POSIX named message queue.

7. Unlink Named Message Queue

```
#include <mqueue.h>
int mq_unlink(const char *name);
```

Prototype analysis on function `mq_close`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***name*** is the name of the POSIX named message queue.

Here is the producer and consumer instances implemented through the POSIX named message queue. The producer program sends a message to the message queue every 1 second. The consumer gets a message from the queue every 2 seconds. After the producer has finished the production, the delay is 1 minute to wait for the consumer to exit normally. After 1 minute, the producer calls the `mq_unlink` function to unlink the message queue file.

Program List 9.7 Producer Program

```
#include <stdio.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

#define MAXSIZE      (10)
#define BUFFER      (8192)
#define FILE_NAME    "/posix"

struct msg_type {
    int    len;
    char   buf[MAXSIZE];
};

int main (int argc, char *argv[])
{
    mqd_t      msgq_id;
    struct msg_type  msg;
    unsigned int  prio = 1;
    struct mq_attr  msgq_attr;
    int           ret;
    int           i;

    msgq_id = mq_open(FILE_NAME, O_RDWR | O_CREAT, S_IRWXU | S_IRWXG, NULL);
    if(msgq_id == (mqd_t)-1) {
        perror("mq_open");
        return (-1);
    }

    ret = mq_getattr(msgq_id, &msgq_attr);
    if(ret < 0) {
        perror("mq_getattr");
        return (-1);
    }

    ret = mq_setattr(msgq_id, &msgq_attr, NULL);
    if(ret < 0) {
        perror("mq_setattr");
    }
}
```

```
        return (-1);
    }

    for (i = 0; i < 10; ++i) {
        memset(msg.buf, 0, MAXSIZE);
        sprintf(msg.buf, "%c", i + 'a');
        msg.len = 1;

        fprintf(stdout, "msg.buf = %s\n", msg.buf);

        ret = mq_send(msgq_id, (char*)&msg, sizeof(struct msg_type), prio);
        if(ret < 0) {
            perror("mq_send");
            return (-1);
        }
        sleep(1);
    }
    sleep(60);

    ret = mq_close(msgq_id);
    if(ret < 0) {
        perror("mq_close");
        return (-1);
    }

    ret = mq_unlink(FILE_NAME);
    if(ret < 0) {
        perror("mq_unlink");
        return (-1);
    }

    return (0);
}
```

Program List 9.8 Consumer Program

```
#include <stdio.h>
#include <mqqueue.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
```

```
#define MAXSIZE      (10)
#define BUFFER      (8192)
#define FILE_NAME   "/posix"

struct msg_type {
    int    len;
    char   buf[MAXSIZE];
};

int main (int argc, char *argv[])
{
    mqd_t      msgq_id;
    unsigned int sender;
    struct msg_type msg;
    struct mq_attr msgq_attr;
    long       recv_size = BUFFER;
    int        ret;
    int        i;

    msgq_id = mq_open(FILE_NAME, O_RDWR);
    if(msgq_id < 0) {
        perror("mq_open");
        return (-1);
    }

    ret = mq_getattr(msgq_id, &msgq_attr);
    if(ret < 0) {
        perror("mq_getattr");
        return (-1);
    }

    if(recv_size < msgq_attr.mq_msgsize) {
        recv_size = msgq_attr.mq_msgsize;
    }

    for (i = 0; i < 10; ++i) {
        msg.len = -1;
        memset(msg.buf, 0, MAXSIZE);

        ret = mq_receive(msgq_id, (char*)&msg, recv_size, &sender);
        if (ret < 0) {
            perror("mq_receive");
            return (-1);
        }
    }
}
```

```
    }

    fprintf(stdout, "msg.len = %d, msg.buf = %s\n", msg.len, msg.buf);
    sleep(2);
}

ret = mq_close(msgq_id);
if(ret < 0) {
    perror("mq_close");
    return (-1);
}

return (0);
}
```

9.6 POSIX Shared Memory

Though both the pipe and POSIX named message queue can achieve the inter-process data communication, the efficiency of the pipe and POSIX named message queue is a bit low when the data volume is large. The POSIX shared memory is recommended for direct data communication.

In order to avoid having multiple writer processes write operations on the same shared memory, it is usually necessary to use a named semaphore as the write lock for that shared memory.

Meanwhile, in order to let the reader process know that the writer process has modified the contents of shared memory, a named semaphore is usually used as the read notification signal for the shared memory.

You can use the `shm_open` function to create a POSIX shared memory. The `shm_open` function specifies the device file path for the POSIX shared memory, and other processes can open the shared memory using the `shm_open` function. The `shm_open` function returns a file descriptor, and then uses the `mmap` function to map the shared memory into the process virtual space. The `mmap` function returns a virtual address, which can then be read and written to the shared memory via this virtual address, thus achieving the purpose of direct and large data volume communication between processes.

When a POSIX shared memory is used, the close function should be called to close it. When a POSIX shared memory no longer has any purpose, the `shm_unlink` function should be called to unlink it. The SylixOS will recycle the kernel resources that the shared memory has consumed.

The detailed usage of POSIX shared memory is shown in Section 12.4 Virtual Memory Management.

9.7 XSI IPC

There are three IPC mechanisms called XSI IPC: message queue, semaphore, and shared memory. They have a lot in common and this section starts with a description of their similar features.

9.7.1 XSI Identifiers and Keys

The IPC structure in each kernel (message queue, semaphore, or shared memory) is referenced with a non-negative integer identifier. For example, to send a message to a message queue or to fetch a message from a message queue, you only need to know its queue identifier. Unlike the file descriptors, the IPC identifier is not a small integer. When an IPC structure is created and then unlinked, the identifiers associated with this structure are added 1 consecutively so as to reach the maximum positive value of an integer, and then back to 0.

The identifier is the internal name of the IPC object. To enable multiple collaboration processes to converge on the same IPC object, an external naming scheme is required. To do this, each IPC object is associated with a key and this action is taken as the external name of the object.

Whenever you create an IPC structure (call `msgget`, `semget`, or `create shmget`), you should specify a key whose data type is the basic system data type `key_t`.

There are several ways to make the client process and server process converge on the same IPC structure:

- The server process can specify the key `IPC_PRIVATE` to create a new IPC structure and store the returned identifiers somewhere (such as a file) for the client process to fetch. The key `IPC_PRIVATE` ensures that the server process creates a new IPC structure;
- A key can be defined in a common header file for both a client process and a server process. The server process then specifies this key to create a new IPC structure;
- The client process and server process identify a path name and item ID (the item ID is the character value between 0 and 255). Next, the call function `ftok` transforms the two values into a single key. The key is then used in the above method.

```
#include <sys/ipc.h>
key_t  ftok(const char *path, int id);
```

Prototype analysis on function `ftok`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;

- The parameter **path** refers to an existing file;
- The parameter **id** is the item id (the parameter only uses the low 8 bits).

The keys created by the `ftok` function usually get the `stat` structure using **path**, and then combine the `st_dev` and `st_ino` member values in the `stat` structure with the item ID. In SylixOS, `st_dev` and `st_ino` are device identifiers, and thus the keys to different files in the same file system may be the same (when the item ID is identical).

`Msgget`, `semget`, and `shmget` have two similar parameters, a **key** and an integer **flag**. When a new queue structure is created, if the key is `IPC_PRIVATE` or is not related to the current type of IPC structure, it needs to specify the flag bit of `flag` `IPC_CREAT`. To reference an existing queue, the key must equal the value of the key specified by the queue creation, and `IPC_CREAT` cannot be specified.

Table 9.3 XSI IPC Flag Bit

Flag	Meaning
<code>IPC_CREAT</code>	Create if the key does not exist
<code>IPC_EXCL</code>	Error if the key exists
<code>IPC_NOWAIT</code>	Non-blocking

It is important to note that you cannot specify `IPC_PRIVATE` as a key to reference an existing queue, because this particular key is always used to create a new queue.

If you want to create a new IPC structure, and make sure that you don't reference an existing IPC structure with the same identifier, you must specify both `IPC_CREAT` and `IPC_EXCL` bits in **flag**. After this setting, if the IPC structure already exists, the error is returned and `errno` is set to `EEXIST`.

9.7.2 XSI Permission Structure

XSI IPC has an `ipc_perm` structure associated with each IPC structure. The structure defines permissions and owners, which are defined in SylixOS as follows:

```
struct ipc_perm {
    uid_t    uid;           /* owner's effective user ID    */
    gid_t    gid;           /* owner's effective group ID   */
    uid_t    cuid;         /* creator's effective user ID  */
    gid_t    cgid;         /* creator's effective group ID */
    mode_t   mode;         /* read/write permission       */
};
```

When you create an IPC structure, you need to assign initial values to all members, and then you can call `MSGCTL`, `semctl`, or `SHMCTL` functions to modify `uid`, `gid`, and `mode` members. To modify these values, the calling process must be the creator or superuser of the IPC structure. Modifying these member values is similar to files calling `chown` functions and `chmod` functions.

9.7.3 XSI IPC Semaphore

The XSI IPC semaphore is different from pipe, named pipe, and message queue; instead, it is a counter that provides access to shared data objects for multiple processes.

To obtain shared resources, the process needs to do the following:

- The test controls the semaphore of the resource;
- If the value of this semaphore is positive, the process can use the resource. In this case, the process reduces the signal value by 1, indicating that it uses a resource unit;
- Otherwise, if the value of this semaphore is 0, the process goes into hibernation until the signal value is greater than 0, and the process is awakened and returns to the first step.

When a process is no longer using a shared resource controlled by a semaphore, the signal is incremented by 1. If a process is dormant waiting for this semaphore, wake it up.

In order to correctly realize the semaphore, the test of the semaphore value and the reduction of 1 operation should be the atomic operation. For this, the semaphore is usually implemented in the kernel.

The commonly used semaphore form is called a binary semaphore, which controls a single resource with an initial value of 1. However, in general, the initial value of the semaphore can be any positive value, indicating how many shared resource units are available for sharing applications.

The kernel provides a `semid_ds` structure for each semaphore set maintainer.

```
struct semid_ds {
    struct ipc_perm    sem_perm;        /* operation permission structure */
    u_short            sem_nsems;       /* number of semaphores in set    */
    time_t             sem_otime;       /* last semop^() time             */
    time_t             sem_ctime;       /* last time changed by semctl()  */
    .....
};
```

Under SylixOS, each semaphore is defined by the following structure:

```
struct sem {
    unsigned short    semval;           /* semaphore value                 */
    pid_t             sempid;          /* pid of last operation           */
    unsigned short    semncnt;         /* # awaiting semval > cval       */
    unsigned short    semzcnt;         /* # awaiting semval == 0         */
};
```

When using a XSI IPC semaphore, the `semget` function is called firstly:

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

Prototype analysis on function semget:

- The semaphore ID returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **key** is the key returned by the ftok function;
- The parameter **nsems** is the number of semaphore in the set.
- The parameter **flag** is the semaphore flag, as shown in Table 9.3.

The semctl function includes a variety of semaphore operations:

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

Prototype analysis on function semctl:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **semid** is the semaphore ID;
- The parameter **semnum** is the semaphore number;
- The parameter **cmd** is the command;
- The parameter... is a variable parameter.

The variable parameter of semctl function is optional according to **cmd**, and its type is union semun, which is a combination of specific parameters under multiple commands:

```
union semun {
    int          val;          /* value for SETVAL          */
    struct semid_ds *buf;     /* buffer for IPC_STAT & IPC_SET */
    unsigned short *array;    /* array for GETALL & SETALL  */
};
```

The dunction semop automatically performs an array of operations on the semaphore set:

```
#include <sys/sem.h>
int semop(int semid, struct sembuf *semoparray, size_t nops);
```

Prototype analysis on function semop:

- The parameter **semid** is the semaphore ID;
- The parameter **semoparray** points to an array of semaphore operations represented by sembuf structure;

- The parameter *nops* is the number of semaphore manipulation arrays.

The following is the type definition of struct sembuf structure:

```
struct sembuf {
    u_short      sem_num;          /* semaphore                */
    short        sem_op;          /* semaphore operation      */
    short        sem_flg;        /* operation flags          */
};
```

9.7.4 XSI IPC Message Queue

The message queues are linked lists of messages, stored in the kernel and identified by message queue identifiers. The msgget function is used to create a new queue or open an existing queue. The msgsnd function adds new messages to the end of the queue. Each message contains a positive long integer type field, a non-negative length, and actual data bytes. All these are passed to the msgsnd function when the message is added to the queue. The msgsnd function is used to fetch messages from the queue, and we do not have to fetch the message in the first-in first-out order, or fetch messages based on the message type.

Each queue has an msqid_ds structure associated with it:

```
struct msqid_ds {
    struct ipc_perm  msg_perm;      /* msg queue permission bits */
    msgqnum_t        msg_qnum;      /* number of msgs in the queue */
    msglen_t         msg_qbytes;    /* max # of bytes on the queue */
    pid_t            msg_lspid;     /* pid of last msgsnd()      */
    pid_t            msg_lrpid;     /* pid of last msgrcv()      */
    time_t           msg_stime;     /* time of last msgsnd()     */
    time_t           msg_rtime;     /* time of last msgrcv()     */
    time_t           msg_ctime;     /* time of last msgctl()     */
    .....
};
```

This structure defines the current state of the queue, and different systems may contain different members.

The first function of the XSI IPC message queue call is the msgget function, which opens an existing queue or creates a new queue.

```
#include <sys/msg.h>
int  msgget(key_t key, int flag);
```

Prototype analysis on function msgget:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;

- The parameter **key** is the key created by the ftok function or the IPC_PRIVATE specified to create the new IPC structure;
- The parameter **flag** is the message creation flag, as shown in Table 9.3.

The msgctl function performs a variety of operations on the queue, similar to the ioctl function.

```
#include <sys/msg.h>
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

Prototype analysis on function msgctl:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **msgid** is the message ID returned by the msgget function;
- The parameter **cmd** is the command as shown in Table 9.4;
- The parameter **buf** is the msgid_ds structure pointer.

Table 9.4 XSI IPC Command

Command	Meaning
IPC_STAT	Take the msgid_ds structure of this queue and store it in buf.
IPC_SET	The members such as msg_perm.uid, msg_perm.gid, and msg_perm.mode in buf are assigned to the msgid_ds structure associated with this queue.
IPC_RMID	Unlink the message queue from the system and all the data that is still in the queue.

Call the MSGSND function to place the data in the message queue:

```
#include <sys/msg.h>
int msgsnd(int msgid, const void *ptr, size_t nbytes, int flag);
```

Prototype analysis on function msgsnd:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **msgid** is the message ID returned by the msgget function;
- The parameter **ptr** is the message pointer;
- The parameter **nbytes** is the number of messages in the message body;
- The parameter **flag** is the message flag.

As previously mentioned, every message is composed of 3 parts: the long integer type of field, the length of a nonnegative (**nbytes**) and the actual data bytes (corresponding to the length), and the message is always at the end of a queue.

The parameter ***ptr*** is a pointer to the `mymesg` structure, which contains the long integer message type and message data, and defines a 512-byte message structure as follows:

```
struct mymesg {
    long    mtype;                /* Message type          */
    char    mtest[512];          /* Message body          */
};
```

The `MSGRCV` function takes messages from the queue.

```
#include <sys/msg.h>
ssize_t msgrcv(int msgid, void *ptr, size_t nbytes, long type, int flag);
```

Prototype analysis on function `msgrcv`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***msgid*** is the message ID returned by the `msgget` function;
- The parameter ***ptr*** is the message pointer;
- The parameter ***nbytes*** is the message buffer length;
- The parameter ***type*** is the message type;
- The parameter ***flag*** is the message flag.

Like the `msgsnd` function, the ***ptr*** parameter points to the long integer (where the returned message type is stored), followed by a buffer that stores the actual message data. The parameter ***type*** can specify which message you want:

- `type == 0` returns the first message in the queue;
- `type > 0` returns the first message of type in the queue;
- `type < 0` the message type value in the return queue is less than or equal to the absolute value of `type`, and if there are several of these messages, the message with the lowest type value is taken.

The type value not 0 is used for reading messages not in the first-in first-out order. For example, if the application gives priority to messages, then the type can be a priority value. If a message queue is used by multiple client processes and a service process, and then the type field can be used to contain the client process ID (as long as the process ID can be stored in a long integer).

When the `msgrcv` is successfully executed, the kernel updates the `msgid_ds` structure associated with the message queue to indicate the caller's process ID and call time, and indicates that the number of messages in the queue has been reduced by 1.

9.7.5 XSI IPC Shared Memory

The shared storage allows two or more processes to share a given storage area. Because the data does not need to be replicated between the client process and the server process, this is the fastest type of IPC. The only skill to master when using the shared storage is to synchronize access to a given storage area between multiple processes. If the server process is putting data into a shared storage area, the client process should not fetch the data before it completes the operation. The semaphore is usually used for synchronizing the shared storage access. The XSI shared memory differs from the memory mapping file that the former has no relevant files. The XSI shared memory segment is an anonymous segment of memory.

The kernel maintains a structure for each shared memory segment, which is implemented in SylixOS as follows:

```
struct shmid_ds {
    struct ipc_perm    shm_perm; /* operation permission structure */
    size_t            shm_segsz; /* size of segment in bytes */
    pid_t             shm_lpid; /* process ID of last shared memory op */
    pid_t             shm_cpid; /* process ID of creator */
    shmatt_t          shm_nattch; /* number of current attaches */
    time_t            shm_atime; /* time of last shmat() */
    time_t            shm_dtime; /* time of last shmdt() */
    time_t            shm_ctime; /* time of last change by shmctl() */
    void              *shm_internal;
};
```

The first function that XSI IPC calls is the `shmget` function, which gets a shared memory identifier:

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

Prototype analysis on function `shmget`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **key** is the return key of `ftok` function;
- The parameter **size** is the length of the shared memory area;
- The parameter **flag** is the shared memory flag.

The parameter **size** is the length of the shared memory area in bytes, and the implementation usually takes it up as an integer multiple of the system page length. However, if the specified size value is not an integer multiple of the system page length,

the remainder of the last page is not available. If you are creating a new segment, you must specify its **size**. If an existing segment is being referenced, the **size** is specified as 0. When a new segment is created, the contents of the segment are initialized to 0.

The SHMCTL function performs a variety of operations on the shared storage segments:

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmctl_ds *buf);
```

Prototype analysis on function shmctl:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **shmid** is the shared memory ID;
- The parameter **cmd** is the command;
- The parameter **buf** is the structure pointer of the structure shmctl_ds.

Once a shared memory segment is created, the process can call the shmat function to connect it to its address space.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

Prototype analysis on function shmat:

- This function returns the mapped memory address if succeeds, and returns MAP_FAILED and sets the error number if fails;
- The parameter **shmid** is the shared memory ID;
- The parameter **addr** must be NULL;
- The parameter **flag** is the shared memory flag.

If the SHM_RDONLY bit is specified in the **flag**, then the segment is connected read-only. Otherwise, connect this section with read and write mode.

The return value of the shmat function is the actual address that the segment is connected to, and MAP_FAILED is returned if an error occurs. If the shmat function is executed successfully, the kernel will add 1 to the shm_nattach counter in the shmctl_ds structure of the shared memory segment. When the operation on the shared storage segment is finished, the shmdt function is called to disconnect the segment. Note that this will not unlink its identifier and data structure from the system. The identifier still exists until a process calls the shmctl (with the command IPC_RMID) to unlink it.

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

Prototype analysis on function `shmdt`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***addr*** is the memory address to be disconnected.

The parameter ***addr*** is the return value when the `shmat` function was called before. If successful, the `shmdt` function will subtract 1 from the `shm_nattach` counter value in the associated `shmid_ds` structure.

Where the kernel places the shared memory area is closely related to the system. The following program prints the information of each data memory location.

Program List 9.9 Shared Memory Distribution

```
#include <stdlib.h>
#include <mman.h>
#include <sys/shm.h>

#define ARRAY_SIZE      (4096)
#define MALLOC_SIZE     (4096)
#define SHM_SIZE        (4096)
#define SHM_MODE         (0666)

static char  array[ARRAY_SIZE];

int main (int argc, char *argv[])
{
    int      shmid;
    char    *ptr, *shmptr;

    fprintf(stdout, "global data area from [0x%lx] to [0x%lx]\n",
            (unsigned long)&array[0],
            (unsigned long)&array[ARRAY_SIZE]);
    fprintf(stdout, "stack area [0x%lx]\n", (unsigned long)&shmid);

    ptr = malloc(MALLOC_SIZE);
    if(ptr == NULL) {
        fprintf(stderr, "malloc error");
        return (-1);
    }

    fprintf(stdout, "heap area from [0x%lx] to [0x%lx]\n", (unsigned long)ptr,
            (unsigned long)ptr + MALLOC_SIZE);
```



```
if((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0) {
    fprintf(stderr, "shmget error");
    return (-1);
}

if((shmptr = shmat(shmid, 0, 0)) == MAP_FAILED) {
    fprintf(stderr, "shmat error");
    return (-1);
}

fprintf(stdout, "shared memory area from [0x%x] to [0x%x]\n",
        (unsigned long)shmptr, (unsigned long)shmptr + SHM_SIZE);

if(shmctl(shmid, IPC_RMID, 0) < 0) {
    fprintf(stderr, "shmctl error");
    return (-1);
}

return (0);
}
```

Operation results under SylixOS:

```
# ./shm_test
global data area from [0xc00107cc] to [0xc00117cc]
stack area [0x30915dc4]
heap area from [0xc0032400] to [0xc0033400]
shared memory area from [0xc0006000] to [0xc0007000]
```

As you can see from the operation results that the shared memory area of SylixOS is under the heap memory area. Note that the mapped memory of XSI IPC shared memory is not associated with a specific file, whereas the mapped memory of mmap function is associated with a specific file (see Section 12.4).

Chapter 10 Signal System

10.1 Signal System

A signal is an interrupt that is simulated at the software level^①. The signal processing flow is shown in Figure 10.1. Many of the more important applications need to process signals that provide a way to handle asynchronous events. For example, the end user types the interrupt key and stops a program through the signal mechanism.

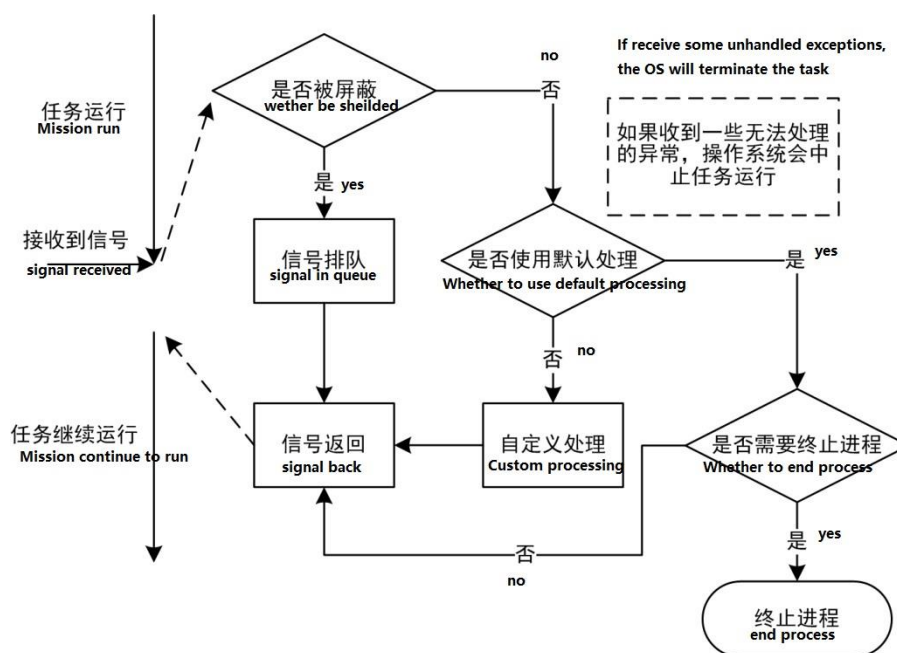


Figure 10.1 Signal Processing Flow

Each signal has its own name and the signal name begins with “SIG”. For example, SIGTERM is a termination signal that sends this signal to the process to terminate a process. Currently, SylixOS can support 63 different signals, including the standard and real-time signals.

Many conditions can generate signals:

- When a user presses a key, this will cause the terminal to generate a signal, for example, Ctrl+C generates the SIGINT signal;
- The alarm function sets the timer timeout to generate the SIGALRM signal;
- The SIGCHLD signal is generated after the child process exits or is abnormally terminated.
- Access to illegal memory generates the SIGSEGV signals;
- The user can call the kill command to send the signal to other processes, and often uses this command to terminate an out-of-control background process.

Signal asynchrony means that the application does not have to wait for events to occur. When the signal occurs, the application is automatically caught in the corresponding signal processing function. Events that generate signals are random in the process. The process cannot simply test a variable to determine whether a signal has occurred, but must tell the kernel that “when this signal occurs, do the following.”

When a signal occurs, you can tell the kernel to handle in one of the following three ways:

- **Ignore signal:** Most of the signals can be processed in this way, and there is a signal in SylixOS that cannot be ignored. The reason that this signal cannot be ignored is that it provides the kernel with a reliable way to terminate the process. In addition, if you ignore certain signals generated by hardware exceptions (such as illegal memory access), the operation behavior of the process is undefined;
- **Capture signal:** To do this, notify the kernel that a user function is called when a signal occurs. In the user function, the action that the user wants is executable. For example, after capturing the SIGALRM signal, the user can control a thread in the corresponding processing function. If the SIGCHLD signal is captured, it indicates that a subprocess has been terminated, and thus the capture function of this signal can call the waitpid function to get the exit status of the child process. Again for example, if the process creates temporary files, it may possibly write a signal capture function for the SIGTERM signal to clear the temporary files. It is important to note that the SIGSTOP signal cannot be captured (the debugging server in SylixOS will capture the SIGKILL signal, and thus the SIGKILL signal in the SylixOS implementation can be captured or ignored).
- **Execute system default action:** The default action for most signals is to terminate the process.

Below we detail the signals supported in SylixOS, as shown in Table 10.1.

Table 10.1 Signals

Signal Name	Description
SIGHUP	Hang up the control terminal or process. This notification daemon is usually used to read their configuration files again, as the daemon does not have a control terminal and will not normally receive such a signal.
SIGINT	Interruption from the keyboard. Ctrl + c is generally used to generate this signal. When a process gets out of control during operation, especially if it is generating a large amount of unwanted output on the screen, this signal will terminate.
SIGQUIT	Exit from the keyboard
SIGILL	Disable instruction
SIGTRAP	Track breakpoint
SIGABRT	Abnormal termination
SIGUNUSED	Unused
SIGFPE	Co-processing error, such as dividing by 0, and floating point overflow etc.
SIGKILL	The forced process terminates.
SIGBUS	A bus error, usually indicating the implemented defined hardware failure
SIGSEGV	Invalid memory reference
SIGUNUSED2	Unused 2
SIGPIPE	Pipe write error, no reader
SIGALRM	Real-time timer alarm
SIGTERM	The process terminates. This is the default action for the kill command, because the signal is captured by the application, and the usage of SIGTERM also gives the program an opportunity to clean up before exiting, and thus gracefully terminates.
SIGCNCL	Thread cancels
SIGSTOP	Stop process execution. This signal cannot be captured and ignored.
SIGTSTP	Tty sends stop process
SIGCONT	The recovery process continues.
SIGCHLD	The child process stops or is terminated. The system default is to ignore this signal.
SIGTTIN	Background process requests input
SIGTTOU	Background process requests output
SIGCANCEL	Identical to sigterm
SIGIO	Asynchronous i/o event
SIGXCPU	The process exceeded the soft cpu event limit.
SIGXFSZ	The process exceeded the soft file length limit.
SIGVTALRM	The virtual interval timer set by the function setitimer has timed out.
SIGPROF	The outline interval timer set by the function setitimer has timed out.
SIGWINCH	The window size is changed.
SIGINFO	Information request
SIGPOLL	Identical to sigio
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2
SIGPWR	The power fails and restarts

SIGSYS	Wrong system call
SIGURG	This signal can be selectively generated when the network is connected to the external data.
SIGRTMIN-SI GRTMAX	SylixOS realizes SIGRTMIN = 48 and SIGRTMAX = 63, the system does not specify a clear meaning, it is user-defined and should not use a value.

10.1.1 Unreliable Signals and Reliable Signals

The signals were unreliable in the early version of UNIX. In other words, the signal might be lost, which was usually expressed as a signal that occurred, but the process might not know it. In the earlier version, the signal was reset to the default value every time the process received a signal to handle it (we will expand on this when we introduce the signal function).

As mentioned above, signals can come from different sources. In SylixOS, the signal source contains several types as shown in Table 10.2. When a signal is generated, the kernel usually sets a flag in some form as specified in the process table. When the signal performs the corresponding action, this represents a signal **delivered** to the process. The signal is **pending** in the interval between signal generation and delivery.

The process can shield (or block) signals. If the signal is generated and the signal action is system default or captured during the signal blocking period, the signal will remain pending until the process unblocks the signal or sets the signal action to be ignored.

The SylixOS kernel will have two ways to handle the signal that occurs several times before the process unblocks the signal: one is that the signal generated by the SI_KILL mode will be delivered only once, that is, the signal will not be queued (see Section 10.4.1); and the other signal produced by the non-SI_KILL method will be delivered multiple times, that is, the signal generates a queue.

In the SylixOS kernel implementation, if multiple different signals are delivered to one process, the signal with a small signal number is preferred to be delivered.

Table 10.2 Source of Signal Generation

Source of Signal Generation	Description
SI_KILL/SI_USER	Signal sent by using kill function
SI_QUEUE	Signal sent by using sigqueue function
SI_TIMER	Signal sent by Posix timer
SI_ASYNCIO	Signal completed and sent by asynchronous I/O system
SI_MSGQ	Signal generated by receiving a message
SI_KERNEL	Internal use of sylixos kernel

As it can be seen, the signal mechanism of SylixOS eliminates the previously unreliable signal mechanism. All the signals generated by the non-SI_KILL mode will be queued.

Because a thread is a unit of SylixOS dispatching, each signal that needs processing will be embedded into the thread to execute, it is more consistent with the characteristics of SylixOS to introduce the signal by means of thread. In fact, the delivery of signals to the process in SylixOS is the main thread of delivery to the process^①.

10.2 Signal Installation

10.2.1 Function signal

The simplest interface in the SylixOS signal mechanism is the signal function:

```
#include <signal.h>
void (*signal(int iSigNo, void (*pfuncHandler)(int)))(int);
```

Prototype analysis on function signal:

- This function returns a function pointer when it succeeds and returns SIG_ERR when it fails, as shown in Table 10.3;
- The function pointer to the function has no return value;
 - ◆ The parameter is an integer value.
- The parameter *iSigNo* is any signal name as shown in Table 10.1;
- The parameter *pfuncHandler* is the signal function or constant SIG_IGN, and the constant SIG_DFL to be installed.

We check <system/signal/signal.h> and find out the definitions as shown in Table 10.3:

Table 10.3 Signal Macros

Macro Name	Value
SIG_ERR	(PSIGNAL_HANDLE)-1
SIG_DFL	(PSIGNAL_HANDLE)0
SIG_IGN	(PSIGNAL_HANDLE)1
SIG_CATCH	(PSIGNAL_HANDLE)2
SIG_HOLD	(PSIGNAL_HANDLE)3

Note: The macro PSIGNAL_HANDLE can be found out in <kernel/include/k_ptype.h>:

```
typedef VOID (*PSIGNAL_HANDLE)(INT);
```

In the previous UNIX system implementation, the signal function installed was unreliable because the installed signal was not permanent^①. As long as the signal is delivered, the signal action will revert to the default action. Fortunately, the SylixOS signal mechanism supports the POSIX real-time extension, ensuring that the signal function will permanently install a signal.

10.2.2 Function sigaction

The `sigaction` function checks or modifies the processing actions associated with the specified signal. This function replaces the signal function, and the signal function is implemented by calling the `sigaction` function in the SylixOS system.

```
#include <signal.h>
int sigaction(int iSigNo,
              const struct sigaction *psigactionNew,
              struct sigaction *psigactionOld);
```

Prototype analysis on function `sigaction`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter `iSigNo` is any signal name as shown in Table 10.1;
- The parameter ***psigactionNew*** is the new signal processing structure;
- The output parameter ***psigactionOld*** saves the previously processed structure.

The `sigaction` function uses the following structure to check or modify the processing actions associated with the specified signal:

```
struct sigaction {
    union {
        PSIGNAL_HANDLE _sa_handler;
        PSIGNAL_HANDLE_ACT _sa_sigaction;
    } _u; /* Signal service function
handler */
    sigset_t sa_mask; /* Signal mask code when
executed */
    INT sa_flags; /* The handle handler flag
*/
    PSIGNAL_HANDLE sa_restorer; /* Resume processing
function pointer */
};

#define sa_handler _u._sa_handler
#define sa_sigaction _u._sa_sigaction
```

When changing the signal action, if the `sa_handler` member contains the address of a signal capture function (not a constant `SIG_IGN` or `SIG_DFL`), the `sa_mask` member contains a signal set (see Section 10.3 for the operation of signal set). Before calling the signal capture function, the signal set is added to the thread signal mask. The `sa_flags` member specifies various options for processing signal. The significance of `sa_flags` member flags is shown in Table 10.4.

Table 10.4 Flag Options of `sa_flags`

Option	Description
SA_NOCLDSTOP	Do not generate signal when the child process is deleted.
SA_NOCLDWAIT	No zombie process.
SA_SIGINFO	The signal handle requires the siginfo parameter.
SA_ONSTACK	Self-defined stack
SA_RESTART	Restart the call after the signal handle is executed.
SA_INTERRUPT	Do not restart the call after the signal handle is executed.
SA_NOMASK	Do not prevent receiving the signal in the specified signal processing handle.
SA_RESETHAND	After the handle is executed, set the signal handle as the default action.

If the `sa_flags` contains the `SA_NOCLDSTOP` flag, the parent process does not receive a pause signal from the child process. The `SIGCHLD` signal is ignored, and the `SIGCHLD` signal is introduced in detail in Section 10.6.

Specify the `SA_NOCLDWAIT` flag that the system will take over the resources of the recovery child process, and thus there is no zombie process.

Specify the `SA_NOMASK` flag, and when the signal processing function is executed, if the same signal is received, it will be interrupted so that the recursion is formed.

Specify the `SA_RESETHAND` flag, and once the signal processing function is executed, the signal action will be set to the default action, which is compatible with the previously unreliable signal mechanism.

The `sa_sigaction` member is an alternative signal handler. If the `SA_SIGINFO` flag is used in the `sigaction` structure, the signal handler is used. In SylixOS, members of `sa_sigaction` and `sa_handler` use the same storage area, and all applications can only use one of these two members at once.

Typically, if you use the `sa_handler` member, call the signal handler as follows:

```
void handler(int signo);
```

- If you use the `sa_sigaction` member, which sets the `SA_SIGINFO` flag, then the signal handler is called as follows:

```
void handler(int signo, siginfo_t *siginfo, void *arg);
```

- The `siginfo_t` structure contains information about the cause of signal generation, which is defined as follows in SylixOS:

```
typedef struct siginfo {
    INT          si_signo;
    INT          si_errno;
    INT          si_code;
    union {
        struct {
            INT    _si_pid;
            INT    _si_uid;
        }
    }
};
```



```

        } _kill;
    struct {
        INT          _si_tid;
        INT          _si_overrun;
    } _timer;
    struct {
        INT          _si_pid;
        INT          _si_uid;
    } _rt;
    struct {
        INT          _si_pid;
        INT          _si_uid;
        INT          _si_status;
        clock_t      _si_utime;
        clock_t      _si_stime;
    } _sigchld;
    struct {
        INT          _si_band;
        INT          _si_fd;
    } _sigpoll;
} _sifields;
#define si_pid          _sifields._kill._si_pid
#define si_uid          _sifields._kill._si_uid
#define si_timerid      _sifields._timer._si_tid
#define si_overrun      _sifields._timer._si_overrun
#define si_status       _sifields._sigchld._si_status
#define si_utime        _sifields._sigchld._si_utime
#define si_stime        _sifields._sigchld._si_stime
#define si_band         _sifields._sigpoll._si_band
#define si_fd           _sifields._sigpoll._si_fd
    union sigval
        si_value;
#define si_addr        si_value.sival_ptr /* Faulting insn/memory ref */
#define si_int         si_value.sival_int
#define si_ptr         si_value.sival_ptr
    .....
} siginfo_t;

```

Union `sigval` will be detailed in Section 10.4.2 and the member `si_code` indicates the causes of signal generation. The `si_code` value definition for various signals in SylixOS is shown in Table 10.5. The third parameter of the signal processing function returns the stack address or NULL in the SylixOS.

The `sa_restorer` members are discarded and should not be used.

Table 10.5 Value Definition of si_code

Signal	Code	Description
ANY	SI_KILL	Signal sent by using kill ()
	SI_USER	Identical to SI_KILL
	SI_QUEUE	Signal sent by using sigqueue
	SI_TIMER	Signal sent by Posix timer
	SI_ASYNCIO	Signal completed and sent by asynchronous I/O system
	SI_MSGQ	Signal generated by receiving a message
	SI_KERNEL	Internal use of sylixos kernel
SIGILL	ILL_ILLOPC	Illegal opcode
	ILL_ILLOPN	Illegal operand
	ILL_ILLADR	Illegal address mode
	ILL_ILLTRP	Illegal trap
	ILL_PRVOPC	Privileged opcode
	ILL_PRVREG	Privileged register
	ILL_COPROC	Coprocessor error
	ILL_BADSTK	Internal stack error
SIGFPE	FPE_INTDIV	Integer divided by 0
	FPE_INTOVF	Integer overflow
	FPE_FLTDIV	Floating point divided by 0
	FPE_FLTOVF	Floating point overflow upwards
	FPE_FLTUND	Floating point overflow downwards
	FPE_FLTRES	Imprecise floating point result
	FPE_FLTINV	Invalid floating point operation
	FPE_FLTSUB	Subscript out of range
SIGSEGV	SEGV_MAPERR	Address not mapped to object
	SEGV_ACCERR	Invalid permissions for mapped object
SIGBUS	BUS_ADRALN	Invalid address alignment
	BUS_ADRERR	Inexistent physical address
	BUS_OBJERR	Specific object hardware error
SIGTRAP	TRAP_BRKPT	Process breakpoint trap
	TRAP_TRACE	Process tracing trap
SIGCHLD	CLD_EXITED	Subprocess termination
	CLD_KILLED	Subprocess abnormally terminated (without core)
	CLD_DUMPED	Subprocess abnormally terminated (with core, currently
	CLD_TRAPPED	SylixOS does not support core files)
	CLD_STOPPED	Tracked child process trapped
	CLD_CONTINUED	Child process stopped The stopped child process has continued

SIGPOLL	POLL_IN	Available data entry
	POLL_OUT	Available output buffer
	POLL_MSG	Available input message
	POLL_ERR	I/O error
	POLL_PRI	Available high priority input
	POLL_HUP	Equipment disconnected

The following examples show how the sigaction function is used, and the alarm function in the program is described in detail in the following sections.

Program List 10.1 Usage of Sigaction

```
#include <stdio.h>
#include <signal.h>

void handler (int signum, siginfo_t *siginfo, void *arg)
{
    fprintf(stdout, "alarm signal.\n");
}

int main (int argc, char *argv[])
{
    struct sigaction newact, oldact;
    int ret;

    newact.sa_sigaction = handler;
    newact.sa_flags = SA_SIGINFO;
    sigemptyset(&newact.sa_mask);

    ret = sigaction(SIGALRM, &newact, &oldact);
    if (ret < 0) {
        fprintf(stderr, "sigaction error.\n");
        return (-1);
    }

    alarm(2);
    sleep(5);
    sigaction(SIGALRM, &oldact, NULL); /*restore the signal behavior
before */
    return (0);
}
```

10.3 Signal Set

In SylixOS, a signal set that can represent multiple signals is needed to tell the kernel that it is not allowed to deliver the signal in the signal set. Different signal numbers may

exceed the number of digits contained in an integer quantity. Therefore, in general, a signal set cannot be represented by an integer quantity. POSIX.1 defines the data type `sigset_t` to define the corresponding signal set. Different system `sigset_t` may have different definition methods, and thus it should not assume what type `sigset_t` should be. The SylixOS defines the following five functions to operate on the signal set.

```
#include <signal.h>
int sigemptyset(sigset_t *psigset);
int sigfillset(sigset_t *psigset);
int sigaddset(sigset_t *psigset, int iSigNo);
int sigdelset(sigset_t *psigset, int iSigNo);
int sigismember(const sigset_t *psigset, int iSigNo);
```

Prototype analysis on function `sigemptyset`:

- This function returns 0;
- The parameter ***psigset*** is the signal set to operate.

Prototype analysis on function `sigfillset`:

- This function returns 0;
- The parameter ***psigset*** is the signal set to operate.

Prototype analysis on function `sigaddset`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***psigset*** is the signal set to add signal;
- The parameter ***iSigNo*** is the signal we added to the signal set.

Prototype analysis on function `sigdelset`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***psigset*** is the signal set to delete the signal;
- The parameter ***iSigNo*** is the signal to be deleted.

Prototype analysis on function `sigismember`:

- The function returning 1 represents belonging to the specified signal set, returning 0 represents not belonging to the specified signal set, and returning -1 represents error and sets error number;
- The parameter ***psigset*** is the signal set to be judged;
- The parameter ***iSigNo*** is the judged signal.

The `sigemptyset` function initializes a signal set to clear all the signals. The `sigfillset` function initializes a signal set to contain all the signals. All the applications will call a `sigemptyset` function or the `sigfillset` function before operating the signal set. The `sigaddset` function adds the specified signal to the existing signal set. Note that the existing signal set has been initialized. The function `sigdelset` deletes the specified signal from the existing signal set. The function `sigismember` determines whether a signal is contained in the specified signal set.

A thread signal mask (or signal mask code) is the signal set that is currently blocked and cannot be delivered to the process. Calling the `sigprocmask` function can detect, change, or simultaneously detect and change the signal mask.

```
#include <signal.h>
int sigprocmask(int          iHow,
                const sigset_t *sigset,
                sigset_t      *sigsetOld);
```

Prototype analysis on function `sigprocmask`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***iHow*** is the command of signal set operation as shown in Table 10.6;

Table 10.6 Command Value of *iHow*

Macro Name	Value
SIG_BLOCK	The new signal set (sigset) is added to the current signal mask.
SIG_UNBLOCK	Delete the signal contained in the new signal set (sigset) from the current signal mask.
SIG_SETMASK	Assign the new signal set (sigset) to the current signal mask.

- The parameter ***sigset*** is the new signal set;
- The output parameter ***sigsetOld*** saves the previous signal set.

If the ***sigset*** is NULL, the signal mask for the thread is not changed (specifically, if the ***sigsetOlds*** is not empty at this time, then the current signal mask of the thread is returned), and the value of ***iHow*** is meaningless. If the ***sigsetOld*** is NULL, the previous signal set is not saved.

After calling the sigprocmask function, if there are any pending signals that are no longer masked, at least one of them will be delivered to the process before the sigprocmask returns.

As an early BSD compatible interface, the SylixOS provides the following set of functions to operate on the signal mask.

```
#include <signal.h>
int sigmask(int iSigNo);
int siggetmask(VOID);
int sigsetmask(int iMask);
int sigblock(int iBlock);
```

Prototype analysis on function sigmask:

- This function returns the signal mask when it succeeds, returns 0 when it fails and sets the error number;
- The parameter ***iSigNo*** is the signal value.

Prototype analysis on function siggetmask:

- This function returns the current thread signal mask.

Prototype analysis on function sigsetmask:

- This function returns the pre-set signal mask;
- The parameter ***iMask*** is the new signal mask.

Prototype analysis on function sigblock:

- This function returns the pre-set signal mask;
- The parameter ***iBlock*** is a new signal set to be added.

The sigmask function gets the mask bit (mask code bit) corresponding to this signal through the signal value. The siggetmask function is called to get the signal mask for the current thread. The sigsetmask function is called to set the specified signal set as the signal mask for the current thread. The sigblock function is called to add the specified signal set to the current thread signal mask. Note that unlike the sigsetmask function, this function does not replace the previous signal mask, and the sigsetmask function replaces the current thread signal mask with the new signal set.

The sigpending function returns a pending signal set for the current thread, where the signal is blocked and cannot be delivered.

```
#include <signal.h>
int  sigpending(sigset_t  *sigset);
```

Prototype analysis on function sigpending:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The output parameter ***sigset*** returns a pending signal set.

The following examples show the usage of signal set function. The program firstly adds the SIGALRM signal to the thread (main thread of process) signal mask. After 2 seconds to generate the SIGALRM signal, the sigpending function is called to get the pending signal set of the thread and determine whether to include the SIGALRM signal, and finally restore the previous signal mask.

Program List 10.2 Usage of Signal Set Function

```
#include <stdio.h>
#include <signal.h>

void int_handler (int  signum)
{
    fprintf(stdout, "signal SIGALRM\n");
    if (signal(SIGALRM, SIG_DFL) == SIG_ERR) {
        fprintf(stderr, "Reset SIGALRM error.\n");
    }
}

int main (int  argc, char *argv[])
{
    sigset_t newmask, oldmask, pendmask;

    if (signal(SIGALRM, int_handler) == SIG_ERR) {
```

```
        fprintf(stderr, "Signal error.\n");
        return (-1);
    }

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        fprintf(stderr, "Sigprocmask error.\n");
        return (-1);
    }

    alarm(2);
    sleep(5);
    sigpending(&pendmask);
    if (sigismember(&pendmask, SIGALRM) == 1) {
        fprintf(stdout, "Signal SIGALRM pending.\n");
    } else {
        fprintf(stdout, "SIGALRM no pending.\n");
    }

    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        fprintf(stderr, "Resume mask error.\n");
        return (-1);
    }

    sleep(5);
    return (0);
}
```

Run the program under the SylixOS Shell, and the results are stated as follows:

```
#!/sigset_test
Signal SIGALRM pending.
Signal SIGALRM
```

As can be seen from the operation results, the SIGALRM signal is shielded. But when the non-shielding state is restored, the SIGALRM signal processing function has been executed. Therefore, the signal is shielded but not discarded. When the non-shielding state is restored, the signal will continue to be delivered.

10.4 Signal Transmission

Signal events are generated from two sources: hardware source (such as we press a key on the keyboard or other hardware failure occurs) and software source including

illegal arithmetic operation, and calling sending signal function etc. The software sources are shown in Table 10.2.

The usage of signal sending function is introduced according to different signal sources.

10.4.1 Non-queued Signal

The SylixOS can send the non-queued signal by the following functions. This means that if the sent signal is in the thread signal mask (the signal is shielded), the signal has been sent multiple times; when this signal is unshielded, it will only be delivered once:

```
#include <signal.h>
int  raise(int  iSigNo);
int  kill(LW_HANDLE  uLId, int  iSigNo);①
```

Prototype analysis on function raise:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *iSigNo* is the signal value.

Prototype analysis on function kill:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *uLId* is the thread handle;
- The parameter *iSigNo* is the signal value.

The raise function allows the thread to send a signal to itself meanwhile the kill function sends the signal to the specified thread. If it is a process, the signal is sent to the main thread.

The pthread_kill function is the sent signal function in the POSIX thread, which is implemented by calling the kill function in SylixOS.

```
#include <pthread.h>
int  pthread_kill(pthread_t  thread, int  signo);
```

Prototype analysis on function pthread_kill:

- This function returns 0 when it succeeds and returns the corresponding error number when it fails.
- The parameter *thread* is the thread handle;
- The parameter *signo* is the signal value.

^① It is worth noting that this function sends a signal to a process or a process group in Linux.

It is important to note that the **thread** parameter handle is returned by the `pthread_create` function. The generation source type of this type of signal is `SI_KILL`.

10.4.2 Queued Signal

As mentioned above, the SylixOS supports the real-time extension of POSIX, and thus the SylixOS signal mechanism implements signal queuing.

```
#include <signal.h>
int sigqueue(LW_HANDLE uId, int iSigNo, const union sigval sigvalue);
```

Prototype analysis on function `sigqueue`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***uId*** is the thread handle;
- The parameter ***iSigNo*** is the signal value.
- The parameter ***sigvalue*** is the parameter delivered by the signal.

Calling the `sigqueue` function will proactively send a queue type signal. This means that the same signal is sent multiple times and the signal will be queued. If the signal is shielded, after the signal is unshielded, it will be delivered the same times of being sent.

Usually a signal contains only one digital message: the signal itself. In addition to queuing for signals, the real-time extensions of POSIX allow applications to deliver more information while delivering the signal. The information is embedded in union `sigval`. In addition to the information provided by the system, the application can either pass an integer to the signal handler or point to a buffer pointer that contains more information.

The third parameter of the `sigqueue` function is the information delivered to the signal handler by the application. The union `sigval` information is as follows:

```
typedef union sigval {
    INT      sival_int;
    PVOID    sival_ptr;
} sigval_t;
```

- `sival_int`: will deliver an integer value;
- `sival_ptr`: points to a buffer structure containing more information.

This is a consortium type, that is, the application can only deliver one of these two types at a time.

To use the queued signal, you must do the following:

- Specify the `SA_SIGINFO` flag when using the `sigaction` function to install the signal handler. If this flag is not given, then in the SylixOS, the application information will not be delivered to the signal processing function.

- The signal handlers are provided in the `sa_sigaction` member of the `sigaction` structure, not the usual `sa_handler`. If the application uses the `sa_handler` member, you cannot get additional information delivered by the `sigqueue` function.

The `sigqueue` function is similar to the `kill` function except that it can use the parameter ***sigvalue*** to pass the integer and pointer values to the signal handler. The signal cannot be queued indefinitely. In the POSIX definition, the `_POSIX_SIGQUEUE_MAX` limits the maximum value of signal queue, and when it reaches the corresponding limit, the `sigqueue` fails, and the corresponding `errno` value is set. The generation source type of this type of signal is: `SI_QUEUE`.

The following examples show the usage methods of `sigqueue` function. The program defines a struct `CLS` with more information. When the signal is installed, specify the `sa_flags` flag as `SA_SIGINFO` and use the `sa_sigaction` member, and then call the `sigqueue` function to send the signal `SIGUSR1` and attach our additional information in `sival_ptr`.

Program List 10.3 Usage of `sigqueue` Function

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

typedef struct {
    int    num;
    char  name[64];
} cls;

void sig_handler (int signum, struct siginfo *info, void *arg)
{
    cls  *name = (cls *)info->si_value.sival_ptr;

    fprintf(stdout, "num: %d, name: %s\n", name->num, name->name);
}

int main (int argc, char *argv[])
{
    struct sigaction  act;
    int              error;
    cls              name = {1, "sylvixos"};
    union sigval     val;

    act.sa_sigaction = sig_handler;
    act.sa_flags     = SA_SIGINFO;
    sigemptyset(&act.sa_mask);
```

```

error = sigaction(SIGUSR1, &act, NULL);
if (error != 0) {
    fprintf(stderr, "sigaction install signal SIGUSR1 failed.\n");
    return (-1);
}

val.sival_ptr = (void *)&name;

error = sigqueue(getpid(), SIGUSR1, val);
if (error != 0) {
    fprintf(stderr, "sigqueue send signal failed.\n");
    return (-1);
}
sleep(5);
return (0);
}

```

Run the program under the SylixOS Shell, you can see from the operation results that the signal processing function successfully received the information we delivered.

```

# ./sigqueue_test
num: 1, name: sylixos

```

10.4.3 Timer Signal

In the SylixOS, the timer allows the work to be processed at the specified time point. The timer provides a way to delay processing.

1. Process Timer Signal

The SylixOS system provides three types of timers for the process. Each timer decreases its value in a different time domain. When the timer runs out, the corresponding signal is sent to the process and then the timer reloads.

Table 10.7 is the type supported by SylixOS process timer, and its definition can be found in <sys/time.h>.

Table 10.7 Process Timer Type

Timer Type	Description
ITIMER_REAL	Reduce based on the system real time and send the SIGALRM signal
ITIMER_VIRTUAL	Reduce based on the process in user state time and send the SIGVTALRM signal
ITIMER_PROF	Reduce based on the process in kernel state and user state time and send the SIGPROF signal

For the timer of ITIMER_REAL type, each system TICK updates the ITIMER_REAL type time of all processes in the system; if a timeout occurs, the SIGALRM signal is sent. For the timer of ITIMER_VIRTUAL type, only update the operation time of the current process in the user state; if a timeout occurs, the SIGVTALRM signal is sent. For the timer of ITIMER_PROF type, update the operation time of the current process in the user state and kernel state; if a timeout occurs, the SIGPROF signal is sent. The generation source type of this type of signal is: SI_TIMER.

The SylixOS provides the following functions to operate on three types of timers:

```
#include <sys/time.h>
int  setitimer(int          iWhich,
               const struct itimerval *pitValue,
               struct itimerval  *pitOld);
int  getitimer(int iWhich, struct itimerval *pitValue);
```

Prototype analysis on function setitimer:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *iWhich* is the timer type as shown in Table 10.7;
- The parameter *pitValue* is the timer parameter pointer;
- The output parameter *pitOld* saves the previous timer parameter pointer.

Prototype analysis on function getitimer:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *iWhich* is the timer type as shown in Table 10.7;
- The output parameter *pitValue* gets the current timer information pointer.

The setitimer function can set a timer in the context of the process. After the specified time timeout, the corresponding signal can be generated, and the getitimer function can get the timing information of the specified timer. The setitimer function sets the timer expiration time and reload time through the itimerval structure. The time precision of this timer is microsecond.

```
struct itimerval {
    struct timeval    it_interval;
    struct timeval    it_value;
};
```

The following examples show how process timers are used. The program is installed with SIGALRM, SIGVTALRM and SIGPROF. The three signals are spaced by 4 seconds

apart. The program finally uses the circulation mode to wait for the signal. Note that you cannot use the pause function or the sleep function to suspend the process, because the timers of ITIMER_VIRTUAL and ITIMER_PROF types decrease their value only when the process is operating.

Program List 10.4 Process Timer

```
#include <signal.h>
#include <stdio.h>
#include <sys/time.h>

void sig_handler (int signum)
{
    switch (signum) {
        case SIGALRM:
            fprintf(stdout, "catch SIGALRM sinal.\n");
            break;

        case SIGVTALRM:
            fprintf(stdout, "catch SIGVTALRM sinal.\n");
            break;

        case SIGPROF:
            fprintf(stdout, "catch SIGPROF sinal.\n");
            break;

        default:
            fprintf(stdout, "catch other signal.\n");
    }
}

int main (int argc, char *argv[])
{
    struct itimerval    newtime0, newtime1, newtime2, oldtime;
    struct timeval      value;

    signal(SIGALRM, sig_handler);
    signal(SIGVTALRM, sig_handler);
    signal(SIGPROF, sig_handler);

    value.tv_sec        = 4;
    value.tv_usec       = 0;
    newtime0.it_interval = value;
    newtime0.it_value    = value;
```

```

newtime1.it_interval = value;
newtime2.it_interval = value;

value.tv_sec        = 2;
value.tv_usec       = 0;
newtime1.it_value   = value;

value.tv_sec        = 3;
value.tv_usec       = 0;
newtime2.it_value   = value;

setitimer(ITIMER_REAL, &newtime0, &oldtime);
setitimer(ITIMER_VIRTUAL, &newtime1, &oldtime);
setitimer(ITIMER_PROF, &newtime2, &oldtime);

while (1) {
    ; /* Cannot call pause function or sleep
function */
}

return (0);
}

```

Run the program under the SylixOS Shell, and the operation results can be viewed as the operation effects of the timer.

```

# ./setitimer_test
catch SIGVTALRM sinal.
catch SIGPROF sinal.
catch SIGALRM sinal.
catch SIGVTALRM sinal.
catch SIGPROF sinal.
catch SIGALRM sinal.
.....

```

The SylixOS also provides a simpler timer function -- an alarm function in seconds or microseconds.

```

#include <unistd.h>
unsigned int alarm(UINT uiSeconds);
useconds_t ualarm(useconds_t usec, useconds_t usecInterval);

```

Prototype analysis on function alarm:

- When this function is successful, return the remaining seconds of the previous alarm clock, return 0 when it fails, and set the error number;

- The parameter ***uiSeconds*** specifies how many seconds to generate the alarm signal.

Prototype analysis on function `ualarm`:

- When this function is successful, return the remaining seconds of the previous alarm clock, return 0 when it fails, and set the error number;
- The parameter ***usec*** is the initial microseconds;
- The parameter ***usecInterval*** is the interval microseconds.

Set a timer with an alarm or `ualarm` function. At some point in the future, the timer will time out and generate a SIGALRM signal. If this signal is not captured, the default action is to terminate the process.

Each process can only have one alarm clock time. If `alarm` or `ualarm` is called, the alarm clock registered for the process has not timed out, the remaining value of the alarm clock time will be returned as the value of the call, and the previously registered alarm clock will be replaced by the new value.

Although the default action of SIGALRM is to terminate the process, most processes that use an alarm clock will capture this signal. If the process terminates at this point, it can perform the required cleanup operations before terminating. To capture the SIGALRM signal, we need to register our signal function before calling `alarm` or `ualarm`. Here is an example of the usage of alarm clock function.

Program List 10.5 Usage of Alarm Clock Function

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void alarm_handler (int signum)
{
    fprintf(stdout, "alarm signal.\n");
}

int main (int argc, char *argv[])
{
    if (signal(SIGALRM, alarm_handler) == SIG_ERR) {
        fprintf(stderr, "install signal handler failed.\n");
        return -1;
    }

    alarm(1);
    pause();
}
```



```

return (0);
}

```

Run the program under the SylixOS Shell:

```

# ./alarm_test
alarm signal.

```

2. POSIX Timer Signal

In the SylixOS, you can create a specific timer by calling `timer_create`. Unlike the process timer, this timer can send a signal to either a specified thread or a specified function, not just the main thread of the process.

```

#include <sys/time.h>
int timer_create(clockid_t clockid, struct sigevent *sigeventT,
                timer_t *ptimer);
int timer_delete(timer_t timer);
int timer_gettime(timer_t timer, struct itimerspec *ptvTime);
int timer_getoverrun(timer_t timer);
int timer_settime(timer_t timer, int iFlag,
                  const struct itimerspec *ptvNew,
                  struct itimerspec *ptvOld);

```

Prototype analysis on function `timer_create`:

- The function returns 0 when it succeeds and returns -1 when it fails, and sets the error number;
- The parameter *clockid* is the clock source type as shown in Table 10.8;
- The parameter *sigeventT* is a signal event.
- The output parameter *ptimer* returns the timer handle.

Prototype analysis on function `timer_delete`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *timer* is the timer handle.

Prototype analysis on function `timer_gettime`:

- The function returns 0 when it succeeds and returns -1 when it fails, and sets the error number;
- The output parameter *ptvTime* returns the time parameter of the timer.

Prototype analysis on function `timer_getoverrun`:

- This function returns the timer timeout number successfully, fails to return -1 and sets the error number;

- The parameter ***timer*** is the timer handle.

Prototype analysis on function `timer_settime`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***timer*** is the timer handle.
- The parameter ***iFlag*** is the timer flag;
- The parameter ***ptvNew*** is the new time information of the timer;
- The output parameter ***ptvOlds*** saves previous timer time information.

Call the `timer_create` function to create a POSIX timer, and we need to specify the clock source type for the created timer. If the ***sigeventT*** is NULL, then the default signal event is set (timeout sends the SIGALRM signal). If the ***sigeventT*** is not NULL, then the application-specific signal event is set. The application needs to specify the ***ptimer*** buffer address to hold the created timer handle. If the ***ptimer*** is NULL, return -1 and set `errno` to `EINVAL`. The following is the definition of clock source type, and see the detailed introduction in Chapter 10 Clock Management.

Table 10.8 Clock Source

Clock Source Name	Description
CLOCK_REALTIME	Represent actual physical time
CLOCK_MONOTONIC	Monotone growth time

The properties and behaviors of the timer are contained in struct sigevent. The member sigev_signo is the signal that we want to send by the timer timeout. The notification type of sigev_notify signal is shown in Table 10.9. The sigev_notify_function is the function that needs to be notified, and the sigev_notify_thread_id is the thread ID that needs to be notified. The application needs to cooperate with the sigev_notify type to select different signal notification modes.

```
typedef struct sigevent {
    INT                sigev_signo;
    union sigval       sigev_value;
    INT                sigev_notify;
    void               (*sigev_notify_function)(union sigval);
#ifdef LW_CFG_POSIX_EN > 0
    pthread_attr_t     *sigev_notify_attributes;
#else
    PVOID              sigev_notify_attributes;
#endif
    LW_OBJECT_HANDLE   sigev_notify_thread_id; /* Linux-specific */
    /* equ pthread_t */
    .....
} sigevent_t;
```

Table 10.9 Signal Notification Type

Macro Name	Description
SIGEV_NONE	No signal notification
SIGEV_SIGNAL	Send signal notification
SIGEV_THREAD	Notify the sigev_notify_function and the system creates a new thread
SIGEV_THREAD_ID	Notify the sigev_notify_thread_id thread to apply your own thread creation.

Calling the `timer_delete` function will delete an already created POSIX timer. If the deleted timer does not exist, return -1 and set `errno` to `EINVAL`. Calling the `timer_gettime` function will return the time information of the timer. It is important to note that if the timer exists but does not run, it returns successfully and the time value is 0. Calling the `timer_getoverrun` function will get the number of timer timeout. In SylixOS, this value can also be got by the `si_overrun` member of the `siginfo_t` struct (for example, call the `sigwaitinfo` function, see Section 10.5 Signal Block). POSIX provides that if the timeout value returned is greater than `DELAYTIMER_MAX`, the `DELAYTIMER_MAX` value will be returned. In fact, the SylixOS provides the following functions to return values greater than `DELAYTIMER_MAX`. It is important to note that the usage of this function is limited, and there is no header file to which the function belongs. In other words, the application cannot be used directly. In fact, this function is the extension provided by SylixOS for `timerfd` (see Chapter 13 Standard I/O Equipment).

```
INT timer_getoverrun_64(timer_t timer, UINT64 *pu64Overruns, BOOL bClear);
```

The timer created by calling the `timer_create` function does not start. The `timer_settime` function is called to associate the created timer to an expiration time and start the timer. The timer uses the `itimerspec` structure to set the expiration time value (`it_value`) and the reload time value (`it_interval`). If the reload time value is 0 and the expiration time value is not 0, then the timer will not automatically reload, and the timer will stop automatically once the timer expires. The timer stops if the expiration time value and reload time value are 0 at the same time. The POSIX timer provides nanosecond time precision.

```
struct itimerspec {
    struct timespec    it_interval;           /* Timer reload          */
    struct timespec    it_value;             /* The remaining time until the
next expiration      */
};
```

The timer flag *iFlag* indicates the time type of timer. POSIX defines the absolute clock as follows. The absolute clock time refers to a time point greater than the current time point, and the non-absolute clock is also called the relative clock. The time type POSIX of this clock does not specify a value; in other words, any non-0x1 value is considered by SylixOS to be a relative clock time, and the relative clock time refers to a time length.

```
#include <sys/time.h>
```

```

#define TIMER_ABSTIME    0x1    /* Absolute clock
*/

```

The following program shows how the thread timer is used:

Program List 10.6 Timer Usage

```

#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <pthread.h>

void *timer_thread (void *arg)
{
    sigset_t    sigset;
    int        sig;

    sigemptyset(&sigset);
    sigaddset(&sigset, SIGUSR1);
    sigprocmask(SIG_BLOCK, &sigset, NULL);

    for (;;) {
        sigwait(&sigset, &sig);
        if (sig == SIGUSR1) {
            fprintf(stdout, "Find signal SIGUSR1.\n");
        }
    }
    return NULL;
}

int main (int argc, char *argv[])
{
    struct sigevent    sigev;
    timer_t            timerid;
    pthread_t          tid;
    struct timespec    interval;
    struct itimerspec  tspec;
    int                ret;

    ret = pthread_create(&tid, NULL, timer_thread, NULL);
    if (ret) {
        fprintf(stderr, "pthread create failed.\n");
        return -1;
    }

    sigev.sigev_notify    = SIGEV_THREAD_ID | SIGEV_SIGNAL;
    sigev.sigev_signo    = SIGUSR1;

```

```

sigev.sigev_notify_thread_id    = tid;

ret = timer_create(CLOCK_MONOTONIC, &sigev, &timerid);
if (ret) {
    fprintf(stderr, "timer create failed.\n");
    return -1;
}
/*
 * 我们以秒为单位
 */
interval.tv_sec    = 2;
interval.tv_nsec   = 0;
tspec.it_interval  = interval;
tspec.it_value     = interval;

ret = timer_settime(timerid, 0, &tspec, LW_NULL);
if (ret) {
    fprintf(stderr, "timer settime failed.\n");
    return -1;
}
return 0;
}

```

Run the program under the SylixOS Shell:

```

#./timer_test
Find signal SIGUSR1.

```

The program sends a SIGUSR1 signal to the user thread every two seconds from timer. The thread is blocking the arrival of the SIGUSR1 signal by calling sigwait (see Section 10.5 Signal Blocking). Here the timer clock source uses CLOCK_MONOTONIC and the signal notification parameter sigev_notify uses the options of SIGEV_THREAD_ID and SIGEV_SIGNAL.

10.5 Signal Blocking

```

#include <signal.h>
int  sigsuspend(const sigset_t *sigsetMask);
int  pause(void);

```

Prototype analysis on function sigsuspend:

- This function returns -1;
- The parameter *sigsetMask* is the specified signal mask.

Prototype analysis on function pause:

- This function returns -1;

The `sigsuspend` function sets the current signal mask of the process to the value specified by ***sigsetMask***, and causes the current process to suspend. When the signal specified by ***sigsetMask*** arrives, it will not be processed because it is shielded meanwhile it will not affect the suspended state of the process. When the signal outside ***sigsetMask*** occurs, the signal will be executed and after the signal processing function is returned, the suspended process state will be lifted and the `sigsuspend` function will set the signal mask of the process to the previous value. The return value is -1 and `errno` is set to `EINTR`.

The `pause` function suspends the call process until it catches any signal. The `pause` function returns only if a signal processing function is executed and returned from it. The return value is -1 and `errno` is set to `EINTR`.

Modifying the signal mask can shield or remove the signal selected by shielding. Using this technique can protect the critical area of code that does not want to be interrupted by a signal. Here is a way to protect critical area code.

```
.....
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
.....                               /* Critical area code          */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();
.....
```

The above program fragment uses the `sigprocmask` function to shield the selected signal. When the critical area code is executed, the shielded signal is removed and the `pause` function is called to wait for the shielded signal to be delivered. This process seems to be very protective for the critical area, but there is a very serious problem. If there is a signal between the unshielding moment and the `pause` function in the `sigprocmask` function, then the `pause` function may be blocked forever. In other words, the signal will be lost during this time period. The `sigsuspend` function can be seen as an atomic operation of this process, and thus calling the `sigsuspend` function will not have such a period.

The following functions will synchronously wait for the pending signals while removing the shielding state. If there are more than one signal, return from small to large in a serial manner.

```
#include <signal.h>
int  sigwait(const sigset_t *sigset, int *piSig);
int  sigwaitinfo(const sigset_t *sigset, struct siginfo *psiginfo);
```

Prototype analysis on function `sigwait`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter ***sigset*** is the specified signal set;
- The output parameter ***piSig*** returns a pending signal.

Prototype analysis on function sigtimedwait:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **sigset** is the specified signal set;
- The output parameter **psiginfo** returns the pending signal information;
- The parameter **ptv** is the equal timeout value.

The sigwait function causes the call process or thread to suspend until the signal contained in **sigset** is pending, and the pending signal is returned via **piSig**. This signal will be removed from the mask. Note that the signal in **sigset** is shielded.

The sigwaitinfo function causes the calling process or thread to suspend until the signal contained in **sigset** is pending, and the pending signal is returned via **psiginfo**. Unlike the sigwait function, the sigwaitinfo function returns the signal information as **siginfo_t** (see Section 10.2.2), meaning that more information is returned.

If there is no pending signal, the sigwait function and the sigwaitinfo function will always be blocked. Sometimes this is not allowed by the program. The sigtimedwait function can be called to set a waiting time, and the other functions are the same as the sigwaitinfo function. It is important to note that if **ptv** is NULL, wait forever until a pending signal is generated.

The following functions provide a timeout mechanism for signal waiting. The function returns and sets **errno** to **EAGAIN** when the specified time timeout occurs. In particular, if the parameter **ptv** is NULL, wait until the signal is pending.

```
#include <signal.h>
int sigtimedwait(const sigset_t *sigset,
                 struct siginfo *psiginfo,
                 const struct timespec *ptv);
```

Prototype analysis on function sigtimedwait:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **sigset** is the specified signal set;
- The output parameter **psiginfo** returns the pending signal information;
- The parameter **ptv** is the waiting time information.

The following program shows the usage methods of sigwait function.

Program List 10.7 Sigwait Usage

```
#include <stdio.h>
#include <signal.h>
```



```
int main (int argc, char *argv[])
{
    sigset_t newmask;
    int      sig;
    int      ret;

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    sigprocmask(SIG_BLOCK, &newmask, NULL);
    alarm(1);

    ret = sigwait(&newmask, &sig);
    if (!ret) {
        if (sig == SIGALRM) {
            fprintf(stdout, "Signal SIGALRM.\n");
        }
    }
    return (0);
}
```

Operate this program under the SylixOS Shell, and you can see from the program results that the sigwait function returns the shielded SIGALRM signal via sig.

```
#!/sigwait_test
Signal SIGALRM.
```

10.6 Process and Signal

The termination of child process is an asynchronous event, and the parent process cannot predict when the child process will terminate. The parent process can call the wait function to prevent the accumulation of zombie processes, and usually the parent process can use the following two methods:

- The parent process calls the wait function or the waitpid function without the WNOHANG flag, and if the child process has not been terminated, the calling will be blocked.
- The parent process periodically calls the waitpid function with the WNOHANG flag and performs a non-blocking check on the specified child process.

For the first method, sometimes you may not want the parent process to wait for the child process to terminate in a blocked way. The second method, repeatedly polling, causes waste of CPU resources and increases the complexity of application design. Therefore, in order to avoid these problems, we can use the processing program for the SIGCHLD signal.

Whenever the child process terminates, the SIGCHLD signal is sent to the parent process (this is the default of SylixOS), and SylixOS defaults to the signal. There are two scenarios, one is default ignoring and the other is that if the sa_flags flag of sigaction is set to contain SA_NOCLDWAIT, the system will automatically recycle the child process resources (which will be recovered by the system thread "t_reclaim"). The application can also install signal processing functions to capture the SIGCHLD signals, and the recover the child process resources in the signal processing functions.

As we mentioned above, except for the SI_KILL (kill function sending) type signal, all the other types of signals can be queued. Therefore, we can know that the SA_NOMASK flag is not specified even when the SIGCHLD signal is installed. The SIGCHLD signal sent by the child process will not be lost because the signal will be queued.

If you want to install the SIGCHLD signal processing function, you have to consider reentrancy in the program implementation (details in Section 10.7.2). For example, calling a system function in a signal processing function may change the value of the global variable errno. In this case, when the signal processing function attempts to explicitly set the errno value or when the system function returns a failure to check the errno value, there may be a conflict. Therefore, when you write a signal processing function, you firstly save the errno value with a local variable, and then restore it.

As mentioned above, when the child process exits, the SIGCHLD signal is sent to the parent process. However, if the SA_NOCLDSTOP flag is specified when the sigaction function is called, the child process is forbidden to send the SIGCHLD signal. It is important to note that the SA_NOCLDSTOP flag only works for the SIGCHLD signals. In SylixOS, when the signal SIGCONT causes the stopped child process to recover, it sends the SIGCHLD signal to its parent process, which is allowed in SUSv3.

10.7 Influence of Signal

As mentioned above, the signal is soft interrupt, which is mainly because the signal transmission and the interrupt is the same as the characteristics of asynchrony and randomness.

10.7.1 System Call Interrupt[®]

If the thread catches a signal during the blocking of some slow system call, then the system call will be interrupted, and the error number will be returned and errno will be set to EINTR.

System calls under this category include:

- POSIX message queue call: mq_receive function, and mq_send function;
- POSIX AIO call: aio_suspend function;

- Signal call: sigsuspend function, pause function, sigtimedwait function, and sigwaitinfo function;
- Timer call: nanosleep function, and sleep function;

The function of the signal to the above system call may be exactly what the design desired or the design must avoid. In either case, a sound system should consider this effect adequately. If you want to avoid the effect of the signal on the system call, take certain steps to restart the system call. In 4.2BSD, the program can choose to automatically recover the system call interrupted by the signal, and SylixOS supports this feature. As long as the SA_RESTART flag is set when the signal processing function is installed, the system will automatically determine and restore the interrupted system call.

Table 10.10 shows partial system calls that can be interrupted by signal in SylixOS:

Table 10.10 System Calls Interrupted by Signal

Function Name	Description
nanosleep	Cause thread to sleep at a specified time (nanosecond)
usleep	Cause thread to sleep at a specified time (microsecond)
sleep	Cause thread to sleep at a specified time (second)
mq_send	POSIX message queue sending function
mq_timedsend	POSIX message queue sending function with timeout (absolute time)
mq_reltimedsend_np	POSIX message queue sending function with timeout (relative time)
mq_receive	POSIX message queue receiving function
mq_timedreceive	POSIX message queue receiving function with timeout (absolute time)
mq_reltimedreceive_np	POSIX message queue receiving function with timeout (relative time)
sem_wait	POSIX semaphore blocking function
sem_timedwait	POSIX semaphore blocking function with timeout (absolute time)
sem_reltimedwait_np	POSIX semaphore blocking function with timeout (relative time)

10.7.2 Reentrancy Effect of Function

When a thread catches the signal and processes it, the normal sequence of instructions being executed is temporarily interrupted by the signal handler, which first lyexecutes the instructions in the signal processing function. If the signal handler is returned, the normal sequence of instructions that is being executed when the signal is captured is continuously executed (similar to what happens when the hardware is interrupted). However, in the signal processing function, it is not possible to determine where the thread is executed when the signal is captured. If malloc is being executed (see Chapter 12 Memory Management) to allocate additional storage space in its heap, and then the signal handler is inserted because of the captured signal, it also calls the malloc function, which may cause damage to the context being executed.

The Single UNIX Specification describes functions that guarantee the call of security in a signal handler. These functions are reentrant. In addition to the reentrancy, during the

signal processing operation, it will block any signal that will cause a consistent transmission. These asynchronous signal security functions are listed in Table 10.11.

Table 10.11 Signal Security Function

Function Name	Function Name	Function Name	Function Name	Function Name	Function Name
abort	dup2	getpid	recv	sigfillset	times
accept	execl	getppid	recvfrom	sigismember	umask
access	execle	getsockname	recvmsg	signal	uname
aio_error	execv	getsockopt	rename	sigpending	unlink
aio_return	execve	getuid	rmdir	sigprocmask	utime
aio_suspend	_Exit	kill	select	sigqueue	utimes
alarm	_exit	listen	sem_post	sigsuspend	wait
bind	fchmod	lseek	send	sleep	waitpid
cfgetispeed	fchown	lstat	sendmsg	socket	write
cfgetospeed	fcntl	mkdir	sendto	socketpair	
cfsetispeed	fdatasync	mkfifo	setgid	stat	
cfsetospeed	fstat	mknod	setpgid	symlink	
chdir	fsync	open	setsid	tcdrain	
chmod	ftruncate	pause	setsockopt	tclflush	
chown	getegid	pipe	setuid	tcgetattr	
clock_gettime	geteuid	poll	shutdown	tcsetattr	
close	getgid	pselect	sigaction	time	
connect	getgroups	raise	sigaddset	timer_getoverrun	
creat	getpeername	read	sigdelset	timer_gettime	
dup	getpgrp	readlink	sigemptyset	timer_settime	

Let's look at an example. The `getpwnam` function is called in the signal processing function `int_handler` to get the username, and `int_handler` is called once every second.

Program List 10.8 Call Non-Reentrant Function in Signal Processing Function

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <pwd.h>

void int_handler (int signum)
{
    struct passwd *ptr;

    fprintf(stdout, "Alarm signal!\n");
    if ((ptr = getpwnam("root")) == NULL) {
```

```
        fprintf(stderr, "getpwnam error.\n");
    }
    alarm(1);
}

int main (int argc, char *argv[])
{
    struct passwd *ptr;

    signal(SIGALRM, int_handler);
    alarm(1);

    for (;;) {
        if ((ptr = getpwnam("sylixos")) == NULL) {
            fprintf(stderr, "getpwnam error.\n");
        }
        if (strcmp(ptr->pw_name, "sylixos") != 0) {
            fprintf(stderr, "ptr->pw_name: %s\n", ptr->pw_name);
        }
    }
    return (0);
}
```

You will find that the program structure is random during the program operation. Generally, after the signal processing function is called a few times, the program will probably have an exception and the signal SIGSEGV will terminate it, or the main function will operate properly but the system Shell has an exception.

As you can see from these examples, if you call a non-reentrant function in the signal processing function, the result is unpredictable.



Chapter 11 Time Management

11.1 SylixOS Time Management

11.1.1 System Time

SylixOS records the clock ticks produced since the system started (we call them TICK) counts, which represent the system time. The clock ticks is produced at a fixed frequency. There are three important functions related to this:

```
#include <SylixOS.h>
ULONG Lw_Time_GetFrequency(VOID);
ULONG Lw_Time_Get(VOID);
INT64 Lw_Time_Get64(VOID);
```

Call the `Lw_Time_GetFrequency` function to get the SylixOS clock tick frequency (the number of ticks per second). The `Lw_Time_Get` function will return the current clock tick counts of SylixOS, and the `Lw_Time_Get64` function will return a wider range of clock tick counts.

The following examples show how long the system has been running through the above functions.

Program List 11.1 Get System Runtime

```
#include <SylixOS.h>
int main (int argc, char *argv[])
{
    ULONG    ulUsecPerTick;
    ULONG    ulMsecTotal;

    ulUsecPerTick = 1000000 / Lw_Time_GetFrequency();
    ulMsecTotal   = Lw_Time_Get () * ulUsecPerTick / 1000;
    fprintf(stdout, "system has run for %lu milliseconds.\n", ulMsecTotal);

    return (0);
}
```

In practice, we are used to adopting unit of time of second or millisecond, and there are many APIs in SylixOS that take the clock ticks as parameters. Therefore, the system provides the following two operation macros for the conversion of clock ticks:

```
#include <SylixOS.h>
```

```

ULONG  LW_MSECOND_TO_TICK_0 (ULONG  u1Ms);
ULONG  LW_MSECOND_TO_TICK_1 (ULONG  u1Ms);

```

LW_MSECOND_TO_TICK_0 converts milliseconds to clock ticks, and the millisecond value less than one clock tick are discarded. **LW_MSECOND_TO_TICK_1** takes the millisecond value less than one clock tick as one clock tick.

11.1.2 RTC Time

RTC time comes from a hardware device that is independent of CPU time. The biggest difference between it and system time is that the RTC time can continue to be counted after the system is switched off. Therefore, it can represent the actual physical time.

```

#include <SylixOS.h>
INT  Lw_Rtc_Set (time_t  time);
INT  Lw_Rtc_Get (time_t  *ptime);

```

Prototype analysis on function `Lw_Rtc_Set`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *time* is the time that needs to be set.

Prototype analysis on function `Lw_Rtc_Get`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The output parameter *ptime* is the got RTC time.

`Time_t` is the time type defined by POSIX, and the more detailed information is shown in Section 11.2.

SylixOS provides 3 functions of RTC time synchronized with system time:

```

#include <SylixOS.h>
INT  Lw_Rtc_SysToRtc (VOID);
INT  Lw_Rtc_RtcToSys (VOID);
INT  Lw_Rtc_RtcToRoot (VOID);

```

`Lw_Rtc_SysToRtc` function is used to synchronize the system time to the RTC time. `Lw_Rtc_RtcToSys` function is used to synchronize the RTC time to the system time. The `Lw_Rtc_RtcToRoot` function is used to synchronize the RTC time to the root file system time.

Note: The system will automatically call `Lw_Rtc_RtcToSys` and `Lw_Rtc_RtcToRoot` to ensure that their time is consistent.



11.2 POSIX Time Management

11.2.1 UTC Time and Local Time

UTC refers to Universal Time Coordinated. In practice, it is equivalent to Greenwich Mean Time (GMT). The UTC time takes 00:00:00 on 01/01/1970 as the base time and takes second as the smallest unit. From the GMT (prime meridian) of Greenwich, England, the earth is divided into 12 time zones with one hour difference between each time zone. This is the local time of each time zone. As the earth rotates from west to east, the eastern time zone is earlier than the GMT, and the western time zone is later than the GMT.

```
#include <time.h>
time_t  time(time_t *time);
time_t  timelocal(time_t *time);
```

Prototype analysis on function time:

- This function returns the UTC time of type time_t;
- The output parameter **time** is the UTC time got, which is the same as the return value. The parameter can be NULL.

Prototype analysis on function timelocal:

- This function returns the local time of type time_t;
- The output parameter **time** is the local time got, which is the same as the return value. The parameter can be NULL.

Time_t is defined as a 32-bit signed integer number in some Unix-like systems, with the maximum positive second being 2147483647 seconds, which can be expressed as the latest time at 03:14:07 on January 19, 2038. This means that time will overflow if more than this. In SylixOS, time_t is defined as a 64-bit signed integer number, and thus no such problem exists. Use the following function to handle the higher precision time.

```
#include <time.h>
int  gettimeofday(struct timeval *tv, struct timezone *tz);
int  settimeofday(const struct timeval *tv, const struct timezone *tz);
```

函数 gettimeofday 原型分析:

Prototype analysis on function gettimeofday:

- This function returns 0 and no error value is returned;
- The output parameter **tv** is a timeval structure pointer which saves the acquired time information;
- The output parameter **tz** is a timezone structure pointer which saves the acquired information of timezone.

The structure `timeval` is defined as follows:

```
struct timeval {
    time_t    tv_sec;           /* seconds */
    LONG     tv_usec;         /* microseconds */
};
```

The value of `tv_usec` is 0-999999, that is, no more than 1 second, and `tv_sec` and `tv_usec` make up the current time. Note that this time is UTC time. The structure `timezone` is defined as follows:

```
struct timezone {
    int    tz_minuteswest;    /* Time difference from Greenwich
time to the West */
    int    tz_dsttime;       /* In minutes (East 8 District)
-60*8 */
    int    tz_dsttime;       /* The time correction method must
be 0 */
};
```

The definition of `tz_minuteswest` is different from the previous one, which is defined as the time difference between Greenwich time and the west, and thus the time zone value of the eastern time zone is negative.

```
#include <time.h>
void    tzset(void);
```

The `tzset` function sets the time zone of the system, which has no parameters. In fact, it gets an environment variable named `TZ`, which is a description of the time zone. In SylixOS, the `tzset` function uses the current value of the environment variable `TZ` to be assigned to the global variable `timezone` and `tzname` (currently SylixOS does not support daylight). The `TZ` is described as follows:

```
# echo $TZ
CST-8:00:00
```

For the CST (China Standard Time), the time difference from Greenwich to the west is negative 8 hours, 0 minute and 0 second, which is actually GMT+8. You can use the following program to set the current time zone:

Program List 11.2 Set System Time Zone

```
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[])
{
    system("TZ=CST-6:0:0");
    tzset();
}
```

```

return (0);
}

```

The above program sets the current time zone to GMT+6, and the following two Shell commands have exactly the same effect.

```

# TZ=CST-6:0:0
# tzsync

```

11.2.2 Time Form Transformation

The time taken in the last Section is expressed in a single second, which is not consistent with normal usage. Therefore, there is a function that converts this time into a time format that we are usually familiar with.

```

#include <time.h>
struct tm *gmtime(const time_t *time);
struct tm *gmtime_r(const time_t *time, struct tm *ptmBuffer);

```

Prototype analysis on function mtime:

- This function returns the tm structure pointer when it succeeds and returns NULL when it fails;
- The parameter *time* is the local time.

Prototype analysis on function gmtime_r:

- This function returns the tm structure pointer when it succeeds and returns NULL when it fails;
- The parameter *time* is the local time;
- The output parameter *ptmBuffer* is the tm structure buffer.

It is important to note that the gmtime function is non-reentrant and therefore it is non-thread and safe. You can see this by Program List 11.3 and Program List 11.4.

The struct tm describes the usage we usually get used to, which is defined as follows:

```

struct tm {
    INT    tm_sec;           /* seconds after the minute - [0, 59] */
    INT    tm_min;         /* minutes after the hour   - [0, 59] */
    INT    tm_hour;        /* hours after midnight     - [0, 23] */
    INT    tm_mday;        /* day of the month         - [1, 31] */
    INT    tm_mon;         /* months since January     - [0, 11] */
    INT    tm_year;        /* years since 1900         */
    INT    tm_wday;        /* days since Sunday        - [0, 6] */
    INT    tm_yday;        /* days since January 1    - [0, 365] */

```

```

#define tm_day      tm_yday
INT  tm_isdst;      /* Daylight Saving Time flag      */
                          /* must zero                                */
};

```

Gmtime converts `time_t` type time to `tm` type time. It is known from the function name that it takes the input parameter time as UTC time (also known as GMT) and the time zone is not converted internally. Therefore, when we use the UTC time as a parameter, the returned `tm` pointer represents the UTC time. When we use the local time as a parameter, the returned `tm` pointer represents the local time.

Note that the pointer returned by `gmtime` actually points to an internal global variable. Therefore, successive calls to this function to get different time will get the same time value, as shown in Program List 11.3.

Program List 11.3 Gmtime Test

```

#include <time.h>
int main (int argc, char *argv[])
{
    struct tm    *tm_old;
    struct tm    *tm_new;
    time_t       time_old;
    time_t       time_new;

    time_old = time(NULL);
    sleep(10);
    time_new = time(NULL);

    tm_old = gmtime(&time_old);
    tm_new = gmtime(&time_new);

    fprintf(stdout, "old: %d:%d\n", tm_old->tm_min, tm_old->tm_sec);
    fprintf(stdout, "new: %d:%d\n", tm_new->tm_min, tm_new->tm_sec);

    return (0);
}

```

After run the program, the results are shown as follows, and you can see that the two time is the same. In fact, `tm_old` and `tm_new` both point to the same object, which is the result of the last call, `tm_new`.

```

# ./gm0_test
old: 33:21
new: 33:21

```

To solve the above problem, there is the corresponding `gmtime_r` function. The suffix `_r` indicates that this is a reentrant version, and it has an output parameter ***ptmBuffer***,

which will change the internal global variable previously used to give the user a buffer object that outputs results. This allows users to get different results if they use different buffer objects. The Program List 11.4 uses `gmtime_r` to handle the same problem.

Program List 11.4 `Gmtime_r Test`

```
#include <time.h>
int main (int argc, char *argv[])
{
    struct tm    tm_old;
    struct tm    tm_new;
    time_t      time_old;
    time_t      time_new;

    time_old = time(NULL);
    sleep(10);
    time_new = time(NULL);

    gmtime_r(&time_old, &tm_old);
    gmtime_r(&time_new, &tm_new);

    fprintf(stdout, "old: %d:%d\n", tm_old.tm_min, tm_old.tm_sec);
    fprintf(stdout, "new: %d:%d\n", tm_new.tm_min, tm_new.tm_sec);

    return (0);
}
```

The program operation result is as follows: the time difference between the two is 10 seconds as expected.

```
# ./gm1_test
old: 56:18
new: 56:28
```

In the above program, we must define two data objects that save the results, instead of just defining two pointers as before. These two objects save different data and achieve the original purpose of the program. The example above shows the problem of continuously calling `gmtime` in a single thread. In multithreading, we must use the `gmtime_r`. There are several other functions that have the same problem and the same solution. This book will no longer elaborate on their reentrancy version functions, just list their function prototypes.

```
#include <time.h>
struct tm *localtime(const time_t *time);
struct tm *localtime_r(const time_t *time, struct tm *ptmBuffer);
```

Localtime has the same performance as gmtime, but it will have UTC time to convert to localtime. Therefore, the correct usage is to pass in the parameter of UTC time.

```
#include <time.h>
char *asctime(const struct tm *ptm);
char *asctime_r(const struct tm *ptm, char *pcBuffer);
```

Prototype analysis on function asctime:

- This function returns the formatted time string pointer when it succeeds, and returns NULL when it fails.
- The parameter *ptm* is a tm structure pointer.

Note that when the asctime function handles the parameter *ptm*, it will not make any time zone conversions. Like the gmtime function, the program should pass in the required tm data objects as needed. The format of the time string returned by the asctime function is "Tue May 21 13:46:22 1991\n". Therefore, when using the reentrant version asctime_r of this function, its parameter *pcBuffer* must ensure that the length is no less than 26 bytes.

```
#include <time.h>
char *ctime(const time_t *time);
char *ctime_r(const time_t *time, char *pcBuffer);
```

Prototype analysis on function ctime:

- This function returns the formatted time string pointer when it succeeds, and returns NULL when it fails.
- The input parameter *time* is a time_t type pointer.

The ctime function converts local time into a string format that matches with human habit. The string format is similar to the converted asctime function. Therefore, the parameter *pcBuffer* length in the reentrant function ctime_r must be guaranteed no less than 26 bytes. Note that the ctime will internally convert UTC time to local time.

```
#include <time.h>
time_t mktime(struct tm *ptm);
time_t timgm(struct tm *ptm);
```

Prototype analysis on function mktime:

- This function returns the converted UTC time when it succeeds.
- The input parameter *ptm* is the tm type pointer.

The performance of mktime function is to convert local time to UTC time. It internally converts the local time to UTC time, and thus the correct input parameter should be local time. The timgm with the same performance is just a conversion from tm to time_t data type, and the correct input parameter should be the UTC time.

You can call the following function to calculate the difference between the two `time_t` types of time.

```
#include <time.h>
double difftime(time_t time1, time_t time2);
```

11.2.3 High-precision Time

```
#include <time.h>
clock_t clock(void);
```

The function `clock` returns the clock counts that have elapsed since the system started, and the count type is `clock_t`. The clock here refers to the clock ticks, and thus the performance of the clock function is the same as the `Lw_Time_Get` function.

The accuracy of the system high-precision time can be got by using the `clock_getres`.

```
#include <time.h>
int clock_getres(clockid_t clockid, struct timespec *res);
```

Prototype analysis on function `clock_getres`:

- The function returns 0 when it succeeds and returns error number when it fails;
- The input parameter ***clockid*** is the clock source ID, and its type `clockid_t` has the following definition:

Table 11.1 Clock Source Definition

Clock Source Name	Description
CLOCK_REALTIME	Represent actual physical time
CLOCK_MONOTONIC	Monotone growth time
CLOCK_PROCESS_CPUTIME_ID	The CPU time consumed by the process from start
CLOCK_THREAD_CPUTIME_ID	The CPU time consumed by the thread from start

- The time accuracy got by the output parameter ***res*** can reach 1 nanosecond precision, and its type `timespec` is defined as follows:

```
struct timespec {
    time_t    tv_sec;           /* seconds          */
    LONG     tv_nsec;         /* nanoseconds     */
};
```

Because `CLOCK_REALTIME` represents the actual physical time, changes to the system time will affect it. The `CLOCK_MONOTONIC` has been growing since the system started and it will not be affected by any operation. The time difference between the two operations is usually calculated by using the `CLOCK_MONOTONIC` clock source.

```
#include <time.h>
```



```
int clock_gettime(clockid_t clockid, struct timespec *tv);
```

Prototype analysis on function `clock_gettime`:

- The function returns 0 when it succeeds, returns -1 when it fails, and sets the error code;
- The parameter *clockid* is the clock source as shown in Table 11.1;
- The output parameter *tv* saves the got high-precision time.

The `clock_gettime` function gets the time value of the struct `timespec` type according to different clock source types. Note that `clock_gettime` gets the UTC time.

```
#include <time.h>
int clock_settime(clockid_t clockid, const struct timespec *tv);
```

Prototype analysis on function `clock_settime`:

- The function returns 0 when it succeeds, returns -1 when it fails, and sets the error code;
- The parameter *clockid* specifies the clock source (only `CLOCK_REALTIME`);
- The parameter *tv* is the high-precision time that needs to be set.

The `CLOCK_MONOTONIC` is not affected by any operation and the `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID` are only updated within the system, therefore the clock source of `clock_settime` can only be `CLOCK_REALTIME`.

```
#include <time.h>
int clock_nanosleep(clockid_t clockid,
                    int flags,
                    const struct timespec *rqtp,
                    struct timespec *rmtp);
```

Prototype analysis on function `clock_nanosleep`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *clockid* specifies the clock source;
- The parameter *flags* is the time type (such as `TIMER_ABSTIME`);
- The parameter *rqtp* specifies the sleep duration;
- The output parameter *rmtp* returns the remaining sleep duration.

`clock_nanosleep` is similar to `nanosleep`, which can make a process sleep specify nanoseconds. The difference is that if the parameter *flags* is specified as absolute time (`TIMER_ABSTIME`), *rmtp* is no longer meaningful. The meaning of the remaining sleep duration usually refers to the remaining duration after the sleep is interrupted by signal.

11.2.4 Get Process or Thread Clock Source

```
#include <time.h>
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

Prototype analysis on function `clock_getcpuclockid`:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter *pid* is the process ID;
- The output parameter *clock_id* returns the clock source type as shown in Table 11.1.

Call the `clock_getcpuclockid` function to get the clock source type for the specified process *pid*, and SylixOS always returns `CLOCK_PROCESS_CPUTIME_ID`.

```
#include <pthread.h>
int pthread_getcpuclockid(pthread_t thread, clockid_t *clock_id);
```

Prototype analysis on function `pthread_getcpuclockid`:

- This function returns 0 when it succeeds and returns the error number when it fails;
- The parameter *thread* is the thread ID;
- The output parameter *clock_id* returns the clock type.

Call the `pthread_getcpuclockid` function to get the clock source type of the specified *thread*, and SylixOS always returns `CLOCK_THREAD_CPUTIME_ID`.

11.2.5 Time-related Extension Operations

For the `timeval` structure (see Section 11.2.1), the system provides several useful macros conveniently to operate the structure object. Note these operations are not defined in POSIX and they exist in Linux and most Unix-like systems. They are defined as follows (though they are both macro definitions, they are defined here in the form of function based on their actual usage):

```
#include <sys/time.h>
void timeradd(struct timeval *a, struct timeval *b, struct timeval *result);
void timersub(struct timeval *a, struct timeval *b, struct timeval *result);
void timerclear(struct timeval *tvp);
int timerisset(struct timeval *tvp);
int timercmp(struct timeval *a, struct timeval *b, CMP);
```

timeradd adds the time of **a** and **b**, and the structure is saved in **result**. The internal process will automatically handle the carry problem from microseconds to seconds. **Timersub** subtracts time **a** from time **b**, the structure is saved in **result**, and the internal process will automatically handle the carry problem from microseconds to seconds. Note that there will be no overflow or size of any security test inside these two operations, and thus the result may not be in line with the expected time value, and these problems need to be processed by the application.

timerclear zeros a time value and **timerisset** detects whether the time value is 0.

timercmp compares the two time and **CMP** is an operator, such as >, ==, <, !=, <= etc.

The following program shows how these macros are used.

Program List 11.5 Operation Example of Timeval Extension

```
#include <stdio.h>
#include <sys/time.h>

#define timeval_show(des, tv) \
    fprintf(stdout, des "sec = %llu, usec = %lu.\n", \
            tv.tv_sec, tv.tv_usec)

int main(int argc, char *argv[])
{
    struct timeval a;
    struct timeval b;
    struct timeval result;

    a.tv_sec    = 100;
    a.tv_usec   = 800000;
    b.tv_sec    = 80;
    b.tv_usec   = 330000;

    timeval_show("time a: ", a);
    timeval_show("time b: ", b);

    timeradd(&a, &b, &result);
    timeval_show("time a + b: ", result);
    timersub(&a, &b, &result);
    timeval_show("time a - b: ", result);

    timerclear(&result);
    if (timerisset(&result)) {
        timeval_show("time is set: ", result);
    }
}
```

```
    } else {
        timeval_show("time is zero: ", result);
    }

    if (timercmp(&a, &b, >)) {
        fprintf(stdout, "time a is larger than time b.\n");
    } else {
        fprintf(stderr, "time a is not larger than time b.\n");
    }

    return (0);
}
```

After the program operation, output the following results:

```
# ./timeext_test
time a: sec = 100, usec = 800000.
time b: sec = 80, usec = 330000.
time a + b: sec = 181, usec = 130000.
time a - b: sec = 20, usec = 470000.
time is zero: sec = 0, usec = 0.
time a is larger than time b.
```

Chapter 12 Memory Management

12.1 Fixed Length Memory Management

Fixed length memory refers to the got same size of memory in each allocation, that is, we use the memory block with the fixed length. Meanwhile the total number of memory blocks is also determined, that is, the total memory size is also determined. This is the same concept as the memory pool we usually understand. There are two advantages in using this memory: one is that the stability of critical applications can be greatly improved by allocating enough memory in advance; and the other is that the management of fixed length memory usually has a simpler algorithm, which is more efficient to allocate/free. In SylixOS, the managed fixed length memory is called PARTITION, or memory partition.

12.1.1 Create Memory Partition

```
#include <SylixOS.h>
LW_HANDLE Lw_Partition_Create(CPCHAR      pcName,
                              PVOID      pvLowAddr,
                              ULONG      ulBlockCounter,
                              size_t     stBlockByteSize,
                              ULONG      ulOption,
                              LW_OBJECT_ID *puId)
```

Prototype analysis on function `Lw_Partition_Create`:

- When this function succeeds, it returns a memory partition handle, returns `LW_HANDLE_INVALID` and sets the error number when it fails.
- The parameter *pcName* specifies the name of memory partition, which can be `LW_NULL`;
- The parameter *pvLowAddr* defines a low address for a user-defined piece of memory, that is, the starting address. The address must satisfy the alignment of a CPU word length, such as in a 32-bit system, which must be 4-byte aligned;
- The parameter *ulBlockCounter* is the number of fixed length memory blocks in the memory partition;
- The parameter *stBlockByteSize* is the size of memory block, which must be no less than the length of a pointer and have 4 bytes in the 32-bit system;
- The parameter *ulOption* is an option to create a memory partition, as shown in Table 12.1;

Table 12.1 Create Options of Memory Partition

Option Name	Explanation
LW_OPTION_OBJECT_GLOBAL	Represent the object as a kernel global object
LW_OPTION_OBJECT_LOCAL	Indicate the object is owned only by one process, that is the local object
LW_OPTION_DEFAULT	Default option

- The output parameter *pullId* saves the ID of memory partition, the same as the return value. It can be LW_NULL.

Note: The LW_OPTION_OBJECT_GLOBAL option is used for the driver or kernel module, and the corresponding LW_OPTION_OBJECT_LOCAL option is used for the application. To make the application more compatible, the LW_OPTION_DEFAULT option is recommended, which contains the properties of LW_OPTION_OBJECT_LOCAL.

It is important to note that the maximum length is LW_CFG_OBJECT_NAME_SIZE for the SylixOS object name, the macro definition is located in the <SylixOS/config/kernel/kernel_cfg.h>, and its value is 32. If the length of the parameter *pcName* exceeds this value, the memory partition creation fails.

12.1.2 Delete Memory Partition

```
#include <SylixOS.h>
ULONG Lw_Partition_DeleteEx(LW_HANDLE *pullId, BOOL bForce);
ULONG Lw_Partition_Delete(LW_HANDLE *pullId);
```

Prototype analysis on function Lw_Partition_DeleteEx:

- This function returns ERROR_NONE when it succeeds and returns an error code when it fails;
- The parameter *pullId* is the memory partition handle pointer, and the successful operation will invalidate the handle.
- The parameter *bForce* indicates whether to force the memory partition to be deleted.

If a memory block is still being used in a memory partition, it should theoretically not be deleted immediately. If *bForce* is LW_TRUE, Lw_Partition_DeleteEx ignores this condition and deletes the partition directly. Typically, the application should not use this method, which can lead to memory errors. It is recommended to use the Lw_Partition_Delete function in general, which corresponds to the following call.

```
Lw_Partition_DeleteEx(Id, LW_FALSE);
```

12.1.3 Get/return Memory Block

```
#include <SylixOS.h>
PVOID Lw_Partition_Get(LW_HANDLE    uId);
PVOID Lw_Partition_Put(LW_HANDLE    uId, PVOID pvBlock);
```

Prototype analysis on function Lw_Partition_Get:

- This function returns the memory block pointer when it succeeds, returns LW_NULL and sets the error number when it fails.
- The parameter *uId* is the memory partition handle.

Prototype analysis on function Lw_Partition_Put:

- This function returns LW_NULL when it succeeds, and returns the currently returned memory block pointer when it fails.
- The parameter *uId* is the memory partition handle;
- The parameter *pvBlock* is the memory block pointer that needs to be returned.

The Lw_Partition_Get function is called to get the memory block of a memory partition, and its size is specified when the memory partition is created. Call the Lw_Partition_Put function to get the memory block (by Lw_Partition_Get) returned to the memory partition. Note that if *pvBlock* is NULL, the error number is set to ERROR_PARTITION_NULL.

In order to meet the needs of different users, SylixOS also provides the following two sets of APIs, with the same performance as above.

```
#include <SylixOS.h>
PVOID Lw_Partition_Take(LW_HANDLE    uId);
PVOID Lw_Partition_Give(LW_HANDLE    uId, PVOID pvBlock);
PVOID Lw_Partition_Allocate(LW_HANDLE uId);
PVOID Lw_Partition_Free(LW_HANDLE    uId, PVOID pvBlock);
```

12.1.4 Get Current State of Memory Partition

```
#include <SylixOS.h>
ULONG Lw_Partition_Status(LW_HANDLE    uId,
                          ULONG        *pulBlockCounter,
                          ULONG        *pulFreeBlockCounter,
                          size_t      *pstBlockSize);
```

Prototype analysis on function Lw_Partition_Status:

- This function returns ERROR_NONE when it succeeds and returns an error code when it fails;
- The parameter *uId* is the memory partition handle;

- The output parameter ***pulBlockCounter*** saves the total memory block number of the memory partition;
- The output parameter ***pulFreeBlockCounter*** saves the number of memory blocks that are not used by the memory partition;
- The output parameter ***pstBlockByteSize*** saves the memory block size of the memory partition.

The `Lw_Partition_Status` function is called to get the status information of the memory partition, such as the total number of memory partitions, the size of memory block, and the number of memory blocks currently available. In particular, if the parameters ***pulBlockCounter***, ***pulFreeBlockCounter***, and ***pstBlockByteSize*** are NULL, this function will be returned calmly.

12.1.5 Get Memory Partition Name

```
#include <SylixOS.h>
ULONG Lw_Partition_GetName(LW_HANDLE uId, PCHAR pcName);
```

Prototype analysis on function `Lw_Partition_GetName`:

- This function returns `ERROR_NONE` when it succeeds and returns an error code when it fails;
- The parameter ***uId*** is the memory partition handle;
- The output parameter ***pcName*** saves the memory partition name.

As mentioned above, the ***pcName*** buffer length should be no less than `LW_CFG_OBJECT_NAME_SIZE` to ensure that the operation buffer is safe.

The following example shows how to use memory partitions.

Program List 12.1 Usage Method of Memory Partition

```
#include <SylixOS.h>

typedef struct my_element {
    INT    iValue;
} MY_ELEMENT;

#define ELEMENT_MAX    (8)
UINT8    _G_pucMyElementPool[sizeof(MY_ELEMENT) * ELEMENT_MAX];
LW_HANDLE _G_hMyPartition;

int main (int argc, char *argv[])
{
    MY_ELEMENT    *peleTable[ELEMENT_MAX] = {LW_NULL};
```



```
MY_ELEMETNET    *peleTmp;
ULONG           ulError;
INT             i = 0;

_G_hMyPartition = Lw_Partition_Create("my_partition",
                                     _G_pucMyElementPool,
                                     ELEMENT_MAX,
                                     sizeof(MY_ELEMETNET),
                                     LW_OPTION_DEFAULT,
                                     LW_NULL);

if (_G_hMyPartition == LW_HANDLE_INVALID) {
    fprintf(stderr, "create partition failed.\n");
    return (-1);
}

/*
 * How many element memories can be obtained
 */
while (1) {
    peleTmp = (MY_ELEMETNET *)Lw_Partition_Get(_G_hMyPartition);
    if (peleTmp != LW_NULL) {
        peleTable[i] = peleTmp;
        peleTmp->iValue = i;
        fprintf(stdout, "get element successfully, count = %d.\n", i);
    } else {
        fprintf(stderr, "get element failed, count = %d.\n", i);
        break;
    }

    i++;
}

/*
 * Delete memory partitions without all reclaiming element memory
 */
ulError = Lw_Partition_Delete(&_G_hMyPartition);
if (ulError != ERROR_NONE) {
    fprintf(stderr, "delete partition error.\n");
} else {
    return (0);
}

for (i = 0; i < ELEMENT_MAX; i++) {
    peleTmp = peleTable[i];
}
```

```
        if (peleTmp != LW_NULL) {
            fprintf(stdout, "element%d value = %d.\n", i, peleTmp->iValue);

            peleTmp = Lw_Partition_Put(_G_hMyPartition, peleTmp);
            if (peleTmp != LW_NULL) {
                fprintf(stderr, "element%d put failed.\n", i);
            }

        } else {
            break;
        }
    }

    /*
     * Delete memory partitions after all memory elements are reclaimed */
    ulError = Lw_Partition_Delete(&_G_hMyPartition);
    if (ulError != ERROR_NONE) {
        fprintf(stderr, "delete partition error.\n");
        return (-1);
    } else {
        fprintf(stderr, "delete partition successfully.\n");
    }

    return (0);
}
```

The memory partition does not allocate memory directly to us, it just provides us with a way to manage memory. Therefore, when creating a memory partition, we need to specify the memory that needs to be managed, which is determined by the size of the element used (that is, the memory block described above) and the maximum number of elements. In Program List 12.1, a memory partition with the maximum of 8 types of MY_ELEMENT objects is created. The usage of the SylixOS memory partition is then shown by getting the element object, using the element object, and deleting the memory partition. After the program operation, the results are stated as follows:

```
get element successfully, count = 0.
get element successfully, count = 1.
get element successfully, count = 2.
get element successfully, count = 3.
get element successfully, count = 4.
get element successfully, count = 5.
get element successfully, count = 6.
get element successfully, count = 7.
get element failed,          count = 8.
delete partition error.
```

```

element0 value = 0.
element1 value = 1.
element2 value = 2.
element3 value = 3.
element4 value = 4.
element5 value = 5.
element6 value = 6.
element7 value = 7.
delete partition successfully.

```

Based on the operation results, the maximum number of elements is 8, and thus the ninth time you get the element will fail. The memory partition is then deleted using the `Lw_Partition_Delete` function. The deletion fails as the element has not yet been recovered, and it can be deleted successfully only when the element is fully recovered.

The maximum size of memory we define can accommodate up to 8 elements, and the maximum number of elements got through the memory partition is also 8. Therefore, it can be speculated that the memory of `_G_pucMyElementPool` is completely used by the program. The address of element 0 is the address of `_G_pucMyElementPool`, the address of element 1 is `_G_pucMyElementPool` plus `MY_ELEMENT` structure size. Therefore, there is a security hazard in the program above. That is, when the address of `_G_pucMyElementPool` does not meet the alignment requirements of the struct `MY_ELEMENT`, on some hardware, access to the member variable `iValue` will result in errors in multiple bytes of misaligned access (typical hardware platform such as ARM). Normally, the compiler will assign aligned address for the variables (global variables or local variables), but does not rule out that there are different ways of handling it under different compilers. At this point, you should define the type of `_G_pucMyElementPool` as `UINT8`, namely single-byte access, and logically its starting address can be any alignment value. For security, the following methods are recommended to define the memory buffer (for example, the Program List 12.1):

```

MY_ELEMENT _G_pmyelementPool[ELEMENT_MAX];
LW_STACK   _G_pstackMyElementPool[sizeof(MY_ELEMENT) * ELEMENT_MAX /
sizeof(LW_STACK)];

```

In Method 1, change the type of `_G_pucMyElementPool` to the type of element to be accessed. The compiler guarantees that the address of `_G_pucMyElementPool` must satisfy the condition of the `MY_ELEMENT` structure alignment access, which is the most common method. In fact, as long as the initial address of the structure satisfies the CPU word length alignment, the the alignment of access to all member variables will not cause the problem. In SylixOS, a data type of `LW_STACK` is defined, which means “stack access type of word alignment”, and the size of this type is actually the CPU word length. Method 2 gives a way to define the memory buffer using `LW_STACK`.

12.2 Variable Length Memory Management

The biggest difference between the variable long memory and the fixed length memory mentioned above is that the size of memory allocated each time may be different. In usage, it is similar to malloc/free, and the only difference is that the memory used is provided by the user. In SylixOS, the variable long memory is called REGION, that is, the memory area. Because the usage of functions like malloc/free is the same memory heap in the system, when a component in an application has an operation that frequently allocates/frees memory, there may be a lot of memory fragments, meanwhile it also affects the efficiency of other applications using memory heaps. In this case, you should consider creating a separate memory area for this component to effectively avoid the above problem.

12.2.1 Create Memory Area

```
#include <SylixOS.h>
LW_HANDLE  Lw_Region_Create(CPCHAR      pcName,
                           PVOID       pvLowAddr,
                           size_t      stRegionByteSize,
                           ULONG       ulOption,
                           LW_OBJECT_ID *pulId);
```

Prototype analysis on function Lw_Region_Create:

- This function returns a memory area handle when it succeeds, returns LW_HANDLE_INVALID and sets the error number when it fails.
- The parameter *pcName* specifies the memory area name and can be LW_NULL;
- The parameter *pvLowAddr* defines a low address for a user-defined piece of memory, that is, the starting address. The address must satisfy the alignment of a CPU word length, such as in a 32-bit system, which must be 4-byte aligned;
- The parameter *stRegionByteSize* is the memory area size with byte as unit;
- The parameter *ulOption* is the option to create the memory area, which is defined in the same way as Table 12.1;
- The output parameter *pulId* returns the ID of memory area, the same as the return value. It can be LW_NULL.

The Lw_Region_Create function can be called to create a variable memory partition. In contrast to the creation of memory partition, the memory area does not require the parameter of data block size. Note that the *pcName* length here should be no less than LW_CFG_OBJECT_NAME_SIZE.

12.2.2 Delete Memory Area

```
#include <SylixOS.h>
ULONG Lw_Region_DeleteEx(LW_HANDLE *pulId, BOOL bForce);
ULONG Lw_Region_Delete(LW_HANDLE *pulId);
```

Prototype analysis on function Lw_Region_DeleteEx:

- This function returns `ERROR_NONE` when it succeeds and returns an error code when it fails;
- The parameter ***pulId*** is the memory area handle pointer, and the successful operation will invalidate the handle.
- The parameter ***bForce*** indicates whether to force the memory area to be deleted.

As with the fixed length memory management, deleting the memory area is not theoretically allowed when the memory area is already in use. Use the `Lw_Region_DeleteEx` function to forcibly delete the memory area. But we should usually use the `Lw_Region_Delete` function to delete a memory area and the function is equivalent to the following call.

```
Lw_Region_DeleteEx(id, LW_FALSE);
```

12.2.3 Memory Area Increases Memory Space

```
#include <SylixOS.h>
ULONG Lw_Region_AddMem(LW_HANDLE ulId,
                      PVOID pvMem,
                      size_t stByteSize);
```

Prototype analysis on function Lw_Region_AddMem:

- This function returns `ERROR_NONE` when it succeeds and returns an error code when it fails;
- The parameter ***ulId*** is the memory area handle;
- The parameter ***pvMem*** is the increased memory pointer;
- The parameter ***stByteSize*** is the increased memory size with byte as unit.

When a memory area has insufficient memory space, it can dynamically increase its memory space. This means that the memory area can be managed with discontinuous memory in multiple addresses, meanwhile the memory partition can only manage one address which must have contiguous memory.

12.2.4 Memory Allocation

```
#include <SylixOS.h>
```

```
PVOID Lw_Region_Allocate(LW_HANDLE uId, size_t stByteSize);
```

Prototype analysis on function `Lw_Region_Allocate`:

- This function returns the allocated memory pointer when it succeeds, returns `LW_NULL` and sets the error number when it fails;
- The parameter *uId* is the memory area handle;
- The parameter *stByteSize* is the memory size that needs to be allocated with byte as unit.

To meet the usage habits of different users, the following two functions have exactly the same performance as that of `Lw_Region_Allocate`:

```
#include <SylixOS.h>
PVOID Lw_Region_Take(LW_HANDLE uId, size_t stByteSize);
PVOID Lw_Region_Get(LW_HANDLE uId, size_t stByteSize);
```

12.2.5 Allocate Memory of Address Aligned

```
#include <SylixOS.h>
PVOID Lw_Region_AllocateAlign(LW_HANDLE uId,
                             size_t stByteSize,
                             size_t stAlign);
```

Prototype analysis on function `Lw_Region_AllocateAlign`:

- This function returns the allocated memory pointer when it succeeds, returns `LW_NULL` and sets the error number when it fails;
- The parameter *uId* is the memory area handle;
- The parameter *stByteSize* is the memory size that needs to be allocated with byte as unit.
- The parameter *stAlign* is the aligned value and must be the power of 2, and it must be greater than or equal to the CPU word length.

The `Lw_Region_AllocateAlign` function is called to get a buffer that specifies the memory alignment relationship. There are two functions that have the same performance:

```
#include <SylixOS.h>
PVOID Lw_Region_TakeAlign(LW_HANDLE uId,
                          size_t stByteSize,
                          size_t stAlign);
PVOID Lw_Region_GetAlign(LW_HANDLE uId,
                          size_t stByteSize,
                          size_t stAlign);
```

12.2.6 Dynamic Memory Adjustment

```
#include <SylixOS.h>
PVOID Lw_Region_Realloc(LW_HANDLE uId,
                       PVOID pvOldMem,
                       size_t stNewByteSize);
```

Prototype analysis on function `Lw_Region_Realloc`:

- This function returns the newly allocated memory pointer when it succeeds, returns `LW_NULL` and sets the error number when it fails.
- The parameter *uId* is the memory area handle;
- The parameter *pvOldMem* is the previously allocated memory pointer;
- The parameter *stNewByteSize* is the new memory size that needs to be allocated and its unit is in byte.

When *stNewByteSize* is larger than the originally allocated memory, new memory will be allocated. Meanwhile the original memory is copied into the newly allocated memory, and the original allocated memory is recycled (this is only the most common case. If there is enough free memory to be contiguous with the currently allocated memory address, the current memory will be extended directly so that the return is the original memory pointer.). When *stNewByteSize* is equal to the originally allocated memory size, it returns directly to the original memory pointer. When *stNewByteSize* is less than the originally allocated memory size, it returns the original memory pointer. However, if the extra memory can be zoned, the extra memory is divided into a new memory zoning recovery management. In addition, the function also contains two hidden behaviors:

- When *stNewByteSize* is 0, just free the memory that *pvOldMem* points to, and the return value is *pvOldMem*.
- When *pvOldMem* is `LW_NULL`, only the memory of *stNewByteSize* is allocated.

Therefore, a more accurate description of the function performance should be: recycle old memory and allocate new memory. It should also be noted that, regardless of the return result after the function is called, the program should not revisit the memory that *pvOldMem* points to.

The two functions with the same performance are stated as follows:

```
#include <SylixOS.h>
PVOID Lw_Region_Reget(LW_HANDLE uId,
                     PVOID pvOldMem,
                     size_t stNewByteSize);
PVOID Lw_Region_Retake(LW_HANDLE uId,
                       PVOID pvOldMem,
                       size_t stNewByteSize);
```

12.2.7 Free the Memory

```
#include <SylixOS.h>
PVOID Lw_Region_Free(LW_HANDLE uId,
                    PVOID pvSegmentData);
```

Prototype analysis on function `Lw_Region_Free`:

- The function returns `LW_NULL` when it succeeds and returns the memory pointer that needs to be freed when it fails.
- The parameter *uId* is the memory area handle;
- The parameter *pvSegmentData* is the memory pointer that needs to be freed;

The two functions with the same performance are stated as follows:

```
#include <SylixOS.h>
PVOID Lw_Region_Put(LW_HANDLE uId,
                   PVOID pvSegmentData);
PVOID Lw_Region_Give(LW_HANDLE uId,
                    PVOID pvSegmentData);
```

12.2.8 Get Current State of Memory Area

```
#include <SylixOS.h>
ULONG Lw_Region_Status(LW_HANDLE uId,
                      size_t *pstByteSize,
                      ULONG *pulSegmentCounter,
                      size_t *pstUsedByteSize,
                      size_t *pstFreeByteSize,
                      size_t *pstMaxUsedByteSize);
```

Prototype analysis on function `Lw_Region_Status`:

- This function returns `ERROR_NONE` when it succeeds and returns an error code when it fails;
- The parameter *uId* is the memory area handle;
- The output parameter *pstByteSize* is the total memory size of memory area;
- The output parameter *pulSegmentCounter* is the total number of memory shards;
- The output parameter *pstUsedByteSize* is the memory size currently used;
- The output parameter *pstFreeByteSize* is the currently remaining memory size;
- The output parameter *pstMaxUsedByteSize* represents the maximum size of memory used so far.

The `Lw_Region_Status` function gets the area information that specifies the variable length memory. The `Lw_Region_StatusEx` function will get an extra Section list in addition to the performance of the `Lw_Region_Status` function.

```
#include <SylixOS.h>
ULONG   Lw_Region_StatusEx(LW_HANDLE           ulId,
                           size_t             *pstByteSize,
                           ULONG              *pulSegmentCounter,
                           size_t            *pstUsedByteSize,
                           size_t            *pstFreeByteSize,
                           size_t            *pstMaxUsedByteSize,
                           PLW_CLASS_SEGMENT psegmentList[],
                           INT               iMaxCounter);
```

Each memory allocation may result in new memory sharding. The memory sharding in one memory area includes the allocated memory, the recovered memory, and the remained memory after allocation. Many tiny memory shardings may be generated after many times of allocating/releasing memory. Because it is too tiny to meet the application usage requirements, the memory allocation fails and this is called memory fragmentation which will “consume” the available memory. The SylixOS kernel uses the memory management algorithm of the “first fit - immediate aggregation”, which can effectively reduce the generation of memory fragmentation.

Use the `Lw_Region_StatusEx` function to look more closely at the fragmentation of the current memory area. The output parameter *psegmentList* is used to save the memory sharding information. *iMaxCounter* indicates that the corresponding buffer can contain the number of holding sharding information, and `PLW_CLASS_SEGMENT` is described as follows:

```
typedef struct {
    LW_LIST_LINE  SEGMENT_lineManage;           /* Neighbor pointer */
    LW_LIST_RING  SEGMENT_ringFreeList;        /* Next free segment linked
list */
    size_t        SEGMENT_stByteSize;          /* Segment size */
    size_t        SEGMENT_stMagic;             /* Segment identification
*/
} LW_CLASS_SEGMENT;
typedef LW_CLASS_SEGMENT *PLW_CLASS_SEGMENT;
```

The definition of this structure is closely related to the memory management algorithm. This book does not focus on discussing specific implementation details, and the application usually does not need to care about this information. Therefore, the specific meaning of the structure members is not explained here.

12.2.9 Get Memory Area Name

```
#include <SylixOS.h>
ULONG Lw_Region_GetName(LW_HANDLE uLld,
                        PCHAR pcName);
```

Prototype analysis on function `Lw_Region_GetName`:

- This function returns `ERROR_NONE` when it succeeds and returns an error code when it fails;
- The parameter *uLld* is the memory area handle;
- The output parameter *pcName* saves the memory area name.

As mentioned above, the *pcName* buffer length should not be less than `LW_CFG_OBJECT_NAME_SIZE` to ensure that the operation buffer is safe.

The following is a concrete example to how the memory area is used and where it is worth noting.

Program List 12.2 Memory Area Test

```
#include <SylixOS.h>

#define REGION_SIZE      (1024)
#define BLOCK_SIZE      (256)
#define BLOCK_CNT       (REGION_SIZE / BLOCK_SIZE)

LW_STACK    _G_pstackRegionMemory[REGION_SIZE / sizeof(LW_STACK)];
LW_HANDLE   _G_hMyRegion;

int main(int argc, char *argv[])
{
    VOID     *pvBlockTable[BLOCK_CNT] = {LW_NULL};
    VOID     *pvBlockTmp;
    ULONG    ulError;
    INT      i = 0;

    _G_hMyRegion = Lw_Region_Create("my_region",
                                   _G_pstackRegionMemory,
                                   REGION_SIZE,
                                   LW_OPTION_DEFAULT,
                                   LW_NULL);

    if (_G_hMyRegion == LW_HANDLE_INVALID) {
        fprintf(stderr, "create region failed.\n");
        return (-1);
    }

    /*
```

```
        * How many times can the test be allocated
    */
    while (1) {
        pvBlockTmp = Lw_Region_Allocate(_G_hMyRegion, BLOCK_SIZE);
        if (pvBlockTmp != LW_NULL) {
            pvBlockTable[i] = pvBlockTmp;
            fprintf(stdout, "alloc block successfully, count = %d.\n", i);
        } else {
            fprintf(stderr, "alloc block failed,          count = %d.\n", i);
            break;
        }
        i++;
    }

    /*
    * Test to remove memory area without fully freeing memory
    */
    ulError = Lw_Region_Delete(&_G_hMyRegion);
    if (ulError != ERROR_NONE) {
        fprintf(stdout, "delete region error.\n");
    } else {
        return (0);
    }

    for (i = 0; i < BLOCK_CNT; i++) {
        pvBlockTmp = pvBlockTable[i];
        if (pvBlockTmp != LW_NULL) {
            pvBlockTmp = Lw_Region_Free(_G_hMyRegion, pvBlockTmp);
            if (pvBlockTmp != LW_NULL) {
                fprintf(stderr, "block%d free failed.\n", i);
            }

        } else {
            break;
        }
    }

    /*
    * Delete memory partition after releasing all memory
    */
    ulError = Lw_Region_Delete(&_G_hMyRegion);
    if (ulError != ERROR_NONE) {
        fprintf(stderr, "delete region error.\n");
        return (-1);
    } else {
```

```
        fprintf(stdout, "delete region successfully.\n");
    }

    return (0);
}
```

After run the program, the results are stated as follows:

```
# ./region_test
alloc block successfully, count = 0.
alloc block successfully, count = 1.
alloc block successfully, count = 2.
alloc block failed,          count = 3.
delete region error.
delete region successfully.
```

In the test program, we created a memory area of 1024 bytes of memory. After each allocation of 256 bytes of memory, the results show that only 3 memory blocks have been allocated successfully. In other words, the memory provided when we create the memory area cannot be fully used by the program. The memory area itself uses some space to store the memory sharding information, which is significantly different from the memory partition. In addition, memory locations allocated through memory areas are always aligned (8-byte alignment on 32-bit systems, the same as Linux), which is the default processing within SylixOS.

12.3 POSIX Standard Memory Management

The POSIX standard memory management-related functions are exactly the same as the variable length memory management described in Section 12.2 in terms of performance and internal behavior. The POSIX specifies that the memory addresses allocated by malloc, calloc, and realloc must be aligned. The purpose of specifying address alignment is to efficiently access any type of data structure on any hardware platform meanwhile it can also avoid on some hardware platforms the abnormal hardware errors caused by multi-byte access on unaligned addresses. This is also consistent with the default processing within SylixOS. In other words, SylixOS memory management itself conforms to the POSIX standard. Each time a new process is created, the memory is allocated internally to create a memory heap for it, rather than requiring the user to specify the memory space as the memory area does.

12.3.1 Memory Allocation

```
#include <malloc.h>
void *malloc(size_t stNBytes);
void *calloc(size_t stNum, size_t stSize);
void *realloc(void *pvPtr, size_t stNewSize);
```

Prototype analysis on function malloc:

- This function returns the allocated memory pointer when it succeeds, returns LW_NULL and sets the error number when it fails;
- The parameter **stNBytes** represents the number of bytes for memory allocation.

Prototype analysis on function calloc:

- This function returns the allocated memory pointer when it succeeds, returns LW_NULL and sets the error number when it fails;
- The parameter **stNNum** represents the number of data blocks;
- The parameter **stSize** represents the size of a data block with byte as unit.

Note: The parameters of calloc seem to indicate that it allocates stNNum size to stSize memory. But in fact, it allocates the memory of a contiguous address space of stNNum*stSize, which is no different from malloc. Unlike malloc, calloc will zero the allocated memory.

Prototype analysis on function realloc:

- This function returns the allocated memory pointer when it succeeds, returns LW_NULL and sets the error number when it fails;
- The parameter **stNewSize** represents the number of bytes of the newly allocated memory.

Note: The behavior of realloc is exactly the same as Lw_Region_Realloc in Section 12.2, which is not repeated here.

Call the malloc function to allocate memory for the application. Support 3 memory allocation methods in SylixOS: the dlmalloc method, the orig method (this method is implemented by the SylixOS kernel, and thus it is usually used in kernel memory allocation), and tlf methods.

Dmalloc is a memory allocator which was developed by Doug Lea in 1987 and has been widely used in many operating systems. Dmalloc adopts two ways to apply for memory. If the applied memory amount in single application is less than 256kb, dlmalloc calls the BRK function to expand the process heap space. But the memory amount of that dlmalloc applies to the kernel is greater than the memory amount applied by the application. After applying to memory, dlmalloc divides the memory into two pieces: one returns to application and the other one is reserved as free memory. When the next application applies for memory, dlmalloc does not need to apply memory to the kernel again, thus speeding up the memory allocation efficiency. When the application calls the free function to free the memory, if the memory block is less than 256kb, dlmalloc does not immediately free the memory block, but marks the memory block as idle. There are two reasons for this: firstly, the memory blocks may not be immediately freed back to the kernel (for example, the memory blocks are not located on the top of the heap); secondly, it supplies the application for memory use next time (this is the main reason). The free

memory is released back into the kernel when the amount of idle memory in the `dmalloc` function reaches a certain value. If the applied memory amount by application is more than 256kb, the `dmalloc` function calls the `mmap` function to apply one memory block to the kernel and returns it to the application. If the application frees the memory more than 256kb, the `dmalloc` function immediately calls the `munmap` function to free the memory. `Dmalloc` does not cache the memory blocks greater than 256kb.

`tlsf` is mainly used to support dynamic memory management of embedded real-time system. It combines the advantages of classified search algorithm and bitmap search algorithm, and is featured by fast speed and low memory consumption. The time complexity of `malloc` and `free` in `tlsf` does not vary with the number of free memory blocks, and it is always $O(1)$.

Note: `tlsf` has the memory management algorithm of $O(1)$ time complexity and it is suitable for real-time operating system. However, it can only maintain 4-byte alignment on 32-bit system, and it can only maintain 8-byte alignment on 64-bit system. It does not satisfy the alignment requirement that POSIX should have $2 * \text{sizeof}(\text{size_t})$ for `malloc`. Therefore, some software may have serious errors such as Qt/JavaScript engine, and it should be used carefully! Use it only after verifying that the application does not have $2 * \text{sizeof}(\text{size_t})$ alignment requirements.

SylixOS can choose which memory allocation method is used by configuring the macro `LW_CFG_VP_HEAP_ALGORITHM`. The macro can be found out in the header file `<SylixOS/config/kernel/memory_cfg.h>`.

12.3.2 Allocate Memory Specifying Alignment Value

```
#include <malloc.h>
void *memalign(size_t stAlign, size_t stNbytes);
int  posix_memalign(void **memptr, size_t alignment, size_t size);
```

Prototype analysis on function `memalign`:

- This function returns the allocated memory pointer when it succeeds, returns `LW_NULL` and sets the error number when it fails;
- The parameter *stAlign* is the aligned value and must be the power of 2;
- The parameter *stNbytes* is the memory size to be allocated.

Prototype analysis on function `posix_memalign`:

- This function returns `ERROR_NONE` when it succeeds and returns an error code when it fails;
- The output parameter *memptr* is the aligned memory pointer that saves allocation;
- The parameter *alignment* is the alignment value and must be the power of 2, and the value must be no less than the CPU word length;

- The parameter **size** is the memory size that needs to be allocated with byte as unit.

`posix_memalign` is the function defined in POSIX1003.1d. This function is different from `memalign` and it requires the alignment value to be no less than the CPU word length, which is the same as the requirement of `Lw_Region_AllocateAlign` function in SylixOS. Note that when this function allocation memory fails, the value of **memptr** is undefined. Therefore, the application should not determine whether the memory is allocated successfully by the value of **memptr** as NULL or not.

12.3.3 Free the Memory

```
#include <malloc.h>
void free(void *pvPtr);
```

Prototype analysis on function free:

- The parameter **pvPtr** is the memory pointer that needs to be freed;

If the input parameter **pvPtr** equals NULL, the free function does not do anything. The free function frees the memory allocated by all the memory allocation functions above.

12.3.4 Memory Allocation Function with Security Detection

```
#include <malloc.h>
void *xmalloc(size_t stNBytes);
void *xcalloc(size_t stNum, size_t stSize);
void *xrealloc(void *pvPtr, size_t stNewSize);
void *xmemalign(size_t stAlign, size_t stNbytes);
```

The above function has the same performance as the memory allocation function with the corresponding name (without x prefix), except that there is a difference in behavior. When the allocated memory fails, the inside of these functions outputs the error information to the standard error output device, `stderr`. Meanwhile it also calls the `exit` function to end the current process, and thus it makes no sense to judge the return value of these functions. Using these functions can bring convenience to your application at some point. For example, when an application has an error code that fails to detect whether the memory is allocated successfully, and the access pointer causes the program to crash, such errors are often difficult to detect. Using this kind of function can provide us with useful information. However, this kind of function cannot meet the requirements in many cases because it has fixed the behavior of memory allocation failure (ending the current process). Note that this set of functions may have an `xfree` function in some systems to free the corresponding memory, and SylixOS uses the `free` function to free.

12.4 Virtual Memory Management

SylixOS, as a multi-process operating system, is divided into kernel space and user space, like the other multi-process operating systems. Kernel threads, drivers, and kernel modules exist in the kernel space meanwhile applications (i.e. processes) and dynamic link libraries exist in the user space.

12.4.1 Memory Division

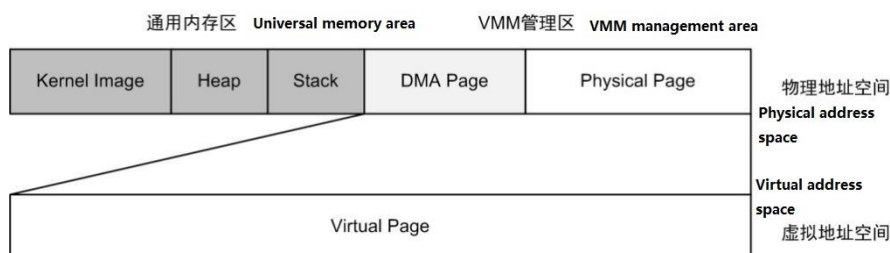


Figure 12.1 System Memory Division

Figure 12.1 describes the layout of SylixOS in physical memory and its relationship to virtual memory. The common memory area is the memory space used by the operating system itself, namely the kernel space. It mainly includes operating system image, memory heap and stack space used by the system. Its physical address and virtual address are exactly the same. It can be seen that they don't have a corresponding virtual page. VMM (Virtual Memory Management) manages all physical memories except the common memory area in the form of page. VMM is also responsible for managing the virtual memory space in the form of page and mapping the virtual memory pages to the physical memory pages when needed. The size of virtual page and physical page has the same value as PAGESIZE (usually 4KB).

In Figure 12.1, there is a special DMA page area dedicated to DMA data transmission (because DMA hardware can only access physical addresses). SylixOS especially provides an API for allocating DMA memory only for kernel modules and drivers. The applications should not use these APIs, and thus they will not be explained here. The rest is the physical pages used by applications and dynamic link libraries which all have the corresponding virtual pages. What we often call the virtual memory is the contiguous virtual page space at this address. The operating system ensures that the virtual page address does not overlap with the common memory and DMA memory addresses. Imagine, if you have any overlap, the process of data itself (global variables, stack space, and code etc.) may be mapped to system memory or DMA memory, which will cause unpredictable errors. The space that we cannot overlap is often called the reserved space of operating system. The mapping relation between the virtual page and the physical page in the Figure above shows only the corresponding relationship between the two ("applications use special methods to directly access DMA memory in virtual space" comes next). However, the physical address space of DMA Page cannot be overlapped

with the virtual address space, and thus the DMA Page is clearly distinguished from the Physical Page.

12.4.2 Process Page Management

As mentioned above, a process is accessing virtual addresses. This includes two aspects: firstly, when the process is created, the loader assigns virtual pages to the process itself, including the process data segment, code segment, and heap memory etc. Secondly, when the process is running, access the stack memory or use the memory allocated by the memory allocation function mentioned above. SylixOS currently allocates 32MB virtual memory pages of contiguous addresses for each newly created process, and it allocates physical memory for some necessary data of the process itself, such as code segments and data segments etc. In addition, the physical memory is allocated only when the process is running, depending on the need for memory access.

The virtual memory space can be larger than the physical memory space. The number of processes supported by the system is not only limited to the size of physical memory, but also limited to the size of virtual memory space. As mentioned above, because of the reserved space, the virtual space is always less than the maximum space that the hardware can access (for example, the virtual space is less than 4GB in the 32-bit CPU). This is a common feature of all multi-process operating systems.

VMM can ensure that the virtual pages allocated each time have contiguous addresses, but the corresponding physical page addresses are not necessarily contiguous. When the process frees the memory, only the corresponding physical page memory is freed and the virtual page is not recycled. When the process exits, the virtual pages and physical pages are all recycled.

12.4.3 Virtual Memory Mapping

The application can use the `mmap` function to map a device file to the application virtual space, this is a memory access changed from file I/O access. The `mmap` function prototype declaration is located in `<sys/mman.h>`. This file contains a set of applications that explicitly handle the functions associated with virtual memory mapping.

```
#include <sys/mman.h>

void *mmap(void *pvAddr, size_t stLen, int iProt,
           int iFlag, int iFd, off_t off);

void *mmap64(void *pvAddr, size_t stLen, int iProt,
             int iFlag, int iFd, off64_t off);
```

Prototype analysis on function `mmap`:

- This function returns the allocated virtual memory address when it succeeds, returns `MAP_FAILED` and sets the error number when it fails.

- The parameter ***pvAddr*** represents the process virtual address that needs to be mapped. If it is not NULL, the return value is the same, and the ***pvAddr*** must be the page aligned address. In most cases, we should use NULL, which means that the system automatically allocates the new virtual memory;
- The parameter ***stLen*** specifies the partial content size of the corresponding file to be mapped with byte as unit. Because VMM manages the memory in pages, when ***stLen*** is not an integer multiple of page size, the number of virtual page bytes actually allocated will be greater than ***stLen***;
- The parameter ***iProt*** is a mapped memory protection option whose value can be either one of the following options or their combination (multiple options are composed of “or”);

Table 12.2 Memory Protection Options

Protection Option Name	Explanation
PROT_READ	Memory pages can read access
PROT_WRITE	Memory pages can write access
PROT_EXEC	Memory pages can execute code
PROT_NONE	Memory pages cannot be accessed

Note: When the PROT_NONE option is used, any access (read, write and execute) mapping memory will result in an invalid page memory access error. In addition, the settings of protection options cannot exceed the open permissions of the file itself. That is, you cannot create a mapping with a file that is opened in read-only mode with the PROT_WRITE method.

- The parameter ***iFlag*** is a memory mapping identifier whose value can be either one of the following options or their combination (multiple options are composed of “or”);

Table 12.3 Memory Mapping Identification

Identification name	Explanation
MAP_SHARED	Shared mapping
MAP_PRIVATE	Private mapping
MAP_FIXED	Fixed virtual address mapping
MAP_ANONYMOUS	Anonymous mapping

Note: When MAP_SHARED is used, the mapped files are shared by multiple processes. This means that a process that modifies its mapping space to all other shared mappings is visible. If multiple processes use MAP_PRIVATE to map the same file, each process modification to its mapping space is not visible to other processes. Once a process writes to the mapping region, the system copies a private mapping space for that process. This is similar to what we know about writing copy technology, which is also supported SylixOS. Meanwhile the written data is not synchronized to the file itself. When

MAP_FIXED is used, the `pvAddr` parameter passed in by the user is used as the virtual address for mapping. The system assumes that the address is a valid virtual address and does not perform any security test and corresponding processing. Therefore, this is a dangerous mode and POSIX does not encourage program to use this option. In SylixOS, the MAP_FIXED option will always cause the mapping to fail. The file descriptor `iFd` and the file offset value `off` are ignored when MAP_ANONYMOUS is used. However, for the portability of the program, `iFd` should be set to -1 when this option is used.

- The parameter ***iFd*** is the file descriptor that needs to be mapped;
- The parameter ***off*** specifies a mapping from one of the starting locations of the file. This parameter must be an integer multiple of the page size.

The parameters ***stLen*** and ***off*** determine the scope of mapping file. When the scope exceeds the size of the file itself, it is still able to create the mapping successfully. However, the data beyond the scope will not be synchronized to the file, that is, the content written will not exceed the size of the file. If the size of the file itself is 0, it will cause the mapping to fail.

Note that not all device files can use `mmap` functions to map memory, and the support of device drivers is usually required. In SylixOS, the disk files and the `FrameBuffer` device files support the `mmap` function operation, and the `mmap` function cannot be used for the serial port device file. The `mmap` function also adds 1 to the mapped file reference counts. This means that even if the `close` function is called to close the file, its file descriptor is still valid; this is because the file is not actually closed and the `close` function only reduces one file descriptor reference count. Therefore, after the `mmap` is called, there is no strict requirement on when to close the mapped file.

`Mmap64` can support 64-bit file offsets. In fact, in SylixOS, the data types of `off_t` and `off64_t` are defined as the 64-bit signed type. Therefore the performances of these two functions are exactly the same. SylixOS provides the `mmap64` functions to improve program compatibility.

At some point, the application may need to expand or shrink the current virtual memory mapping and call the `mremap` function for processing.

```
#include <sys/mman.h>
void *mremap(void *pvAddr, size_t stOldSize, size_t stNewSize, int iFlag, ...);
```

Prototype analysis on function `mremap`:

- This function returns the first address of the new virtual memory when it succeeds, returns MAP_FAILED and sets the error number when it fails.
- The parameter ***pvAddr*** is the virtual address of the current memory mapping and must be the page aligned address;
- The parameter ***stOldSize*** is the size of the current memory mapping with byte as unit;
- The parameter ***stNewSize*** is the new mapping size with byte as unit;

- The parameter *iFlag* is the remapped option, as shown in Table 12.4.

Table 12.4 Remapped Options

Option Name	Explanation
MREAP_MAYMOVE	Allow virtual space of mobile mapping
MREAP_FIXED	Use the specified new virtual address mapping

The behavior of `mremap` function is very similar to the `Lw_Region_Realloc` function and the `realloc` function mentioned above. When *stNewSize* is larger than *stOldSize*, if there are enough pages that are contiguous with the original virtual address, then the original virtual memory will be extended directly to return the originally mapped page address. If there is no virtual page that satisfies this condition and the *iFlag* sets the `MREMAP_MAYMOVE` flag, the new virtual pages will be allocated meanwhile the original virtual pages are recycled and the new page address is returned. This result means that the virtual memory is moved when it is expanded. When *stNewSize* is smaller than *stOldSize*, it will recycle the redundant virtual pages and the corresponding physical pages.

If the `MREMAP_FIXED` option is used, the function will accept the fifth variable parameter, which is void **pvNewAddr*. This parameter is specified by the user specifying the virtual address that needs to be remapped, instead of the internal automatic allocation. This means that the `mmap` function will be mapped and used before the *pvNewAddr* is removed. Using this option is risky and it does not have portability (different systems have different support for this option), and thus it is not recommended to be used in the program. Using this option in SylixOS will return the error directly.

The `mremap` function is not the function as specified by the POSIX standard. But it has been supported since Linux 2.3.1, SylixOS still provides this function for better compatibility.

```
#include <sys/mman.h>
int munmap(void *pvAddr, size_t stLen);
```

Prototype analysis on function `munmap`:

- The function returns 0 when it succeeds and returns error number when it fails;
- The parameter *pvAddr* is the virtual address that uses `mmap` mapping;
- The parameter *stLen* is the memory size that needs to be unmapped with byte as unit, and the inside will be processed as an integer multiple of the page.

The `munmap` function performs opposite to `mmap` function: subtract 1 from the file reference counts (if the reference count of the file is 0, then the file is closed), remove virtual space mappings, and recycle virtual pages and corresponding physical pages (if they exist) etc. Note that `munmap` does not guarantee that the data modified for the

mapping space can be synchronized to the file itself. Therefore, the `msync` function should be called manually to ensure that the modified data is fully written back to the file.

```
#include <sys/mman.h>
int msync(void *pvAddr, size_t stLen, int iFlag);
```

Prototype analysis on function `msync`:

- The parameter `pvAddr` is the mapped virtual space address;
- The parameter `stLen` is the data size that needs to be synchronized with byte as unit;
- The parameter `iFlag` is the synchronization option, as shown in Table 12.5.

Table 12.5 Synchronization Options

Option Name	Explanation
MS_ASYNC	Write back file asynchronously
MS_SYNC	Write back file synchronously
MS_INVALIDATE	Virtual space for invalid mappings

When you use an asynchronous method, calling `msync` function will return immediately, and the kernel asynchronously writes the data back to the file at the appropriate time. When you use the synchronization method, you wait for the data to be written back before returning. The parameter `MS_INVALIDATE` invalidates the virtual memory, then reads the virtual memory and reads the data from the file. In practice, `MS_INVALIDATE` can be combined with `MS_ASYNC` or `MS_SYNC`, but `MS_ASYNC` and `MS_SYNC` cannot be used at the same time.

Use virtual memory mapping for the following purposes:

- Use memory instead of I/O to access device files to improve efficiency;
- Virtual space of multiple processes are mapped to the same device file for shared memory;
- Using anonymous mappings is equivalent to allocating virtual memory, or for communicating between parent and child processes.

Next, based on practical application, several examples illustrate how to use the `mmap` series of functions..

1. Access I/O Devices by Memory

Program List 12.3 Use `mmap` to Access Device

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```

```
#include <stdio.h>

#define FILE_NAME      "mmap.dat"
#define FILE_SIZE      32
#define DATA_OFF      10

char data_buff[] = "<mmap data>";

int main(int argc, char *argv[])
{
    char *file_buff;
    char tmp_buff[FILE_SIZE];
    int fd;
    int ret;

    ret = access(FILE_NAME, F_OK);
    if (ret < 0) {
        char tmp = 'X';
        int i;

        fd = open(FILE_NAME, O_CREAT | O_RDWR, S_IWUSR | S_IRUSR);
        if (fd < 0) {
            fprintf(stderr, "create file failed.\n");
            return (-1);
        }
        for (i = 0; i < FILE_SIZE; i++) {
            write(fd, &tmp, 1);
        }
        close(fd);
    }
    fd = open(FILE_NAME, O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "open file failed.\n");
        return (-1);
    }
    file_buff = mmap(NULL, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (file_buff == MAP_FAILED) {
        fprintf(stderr, "mmap failed.\n");
        return (-1);
    }
    /*
     * Write data using memory access
     */
    memcpy(file_buff + DATA_OFF, data_buff, sizeof(data_buff) - 1);
}
```

```

msync(file_buff, FILE_SIZE, MS_SYNC);
munmap(file_buff, FILE_SIZE);
/*
 * Reading data using I/O functions
 */
ret = read(fd, tmp_buff, FILE_SIZE);
if (ret <= 0) {
    fprintf(stderr, "read file data failed.\n");
} else {
    tmp_buff[ret] = '\0';
    fprintf(stdout, "read file date: %s\n ", tmp_buff);
}
close(fd);
return (0);
}

```

The results of program operation are stated as follows:

```

# ./share_test
read file date: XXXXXXXXXXXX<mmap data>XXXXXXXXXXXXXXXX

```

In the above program, firstly check if the disk file that needs to be mapped exists; if not, create a new file. As mentioned above, if the data length of a file is 0, then mmap cannot be used to set up the mapping, and thus the file is written into data of 32 bytes in length. For intuitive comparison, the data is the “X” that can be displayed. Because we call mmap to map the file in a read-write way, the file is opened in the read-write way. To get the data back to the file itself, we must use the MAP_SHARED mapping option. The program operates file data in the form of memory write and I/O functions. Based on the results, as we expected, the memory of operation mapping is the same as the operation file itself. You can access a part of a file more flexibly using the memory, without the need to use the file location function such as lseek.

Sometimes the application needs to directly access the memory data of device file itself. The most common is frame buffer device (Framebuffer). The device itself has a piece of physical memory that the controller can access through the DMA bus, which is commonly referred to as video memory, from the DMA Page area shown in Figure 12.1. If I/O function is used to operate the video memory, it is inevitable that there will be a data copy between the user buffer and the memory, which greatly affects the refresh response speed of graphical interface. Using the mmap function, you can map the video memory directly to the user space, allowing the application to directly operate the memory itself. Using the frame buffer device usually includes the following procedure: open the device (its device name is usually "/dev/fb0" and "/dev/fb1" etc.), get the relevant information (such as video memory size, and color coding of pixels etc.) of video memory, call mmap to map virtual space, and read and write mappings to virtual space to operate video memory. You can see that the process of using mmap to operate video memory is the same as that of the above operation of common files.

2. Use mmap for Shared Memory

Compared to message queues, pipes and other inter-process communication methods, shared memory has higher communication efficiency on some occasions. Though the purpose of shared memory can be achieved with any device file, there should be a dedicated device in the POSIX-defined operating system that can be used to implement shared memory matched with the mmap function cluster. There is no difference between the device and common disk devices in respect of external performance (both can be used to create/delete files etc.). But it doesn't have a physical storage medium, and you can think of it as a virtual device. Therefore, it makes no sense to call msync on the device file. The relevant API functions are stated as follows:

```
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

The function shm_open is the same as the normal open function, except that it opens or creates a file on the shared memory device. Its prototype analysis is stated as follows:

- This function returns the file descriptor when it succeeds, returns -1 and sets the error number when it fails;
- The parameter **name** is the file name for shared memory mapping;
- The parameter **oflag** is the operational identification, such as O_CREAT, O_RDWR etc.;
- The parameter **mode** is the mode to create the file.

The function shm_unlink is used to delete a file on the memory device. The prototype analysis is stated as follows:

- The function returns 0 when succeeds and returns -1 when fails, and sets the error number;
- The parameter **name** is the file name that needs to be deleted.

Note that there is no shm_close function corresponding to shm_open, this is because the shared memory device is also a standard I/O device, and thus you can close the corresponding file by using the close function. On different systems, the shared memory device may have different implementations with different names, and shm_open hides the difference to achieve portability.

The following two processes simulate the process of a login session between the client-side and the server-side, showing how mmap implements the shared memory.

Program List 12.4 Sample of Server-Side Program of Shared Memory

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
```



```
#include <string.h>
#include <stdio.h>

#define MAX_MSG_SIZE    32
#define REQ_SIG_NAME    "req_signal"
#define ACK_SIG_NAME    "ack_signal"
#define SHM_FILE_NAME   "msg_buffer"

int main(int argc, char *argv[])
{
    char    *msg_buff;
    sem_t   *req_signal;
    sem_t   *ack_signal;
    int     shm_fd;

    req_signal = sem_open(REQ_SIG_NAME, O_CREAT, 0666, 0);
    if (!req_signal) {
        fprintf(stderr, "create request signal failed.\n");
        return (-1);
    }
    ack_signal = sem_open(ACK_SIG_NAME, O_CREAT, 0666, 0);
    if (!ack_signal) {
        fprintf(stderr, "create acknowledge signal failed.\n");
        return (-1);
    }
    shm_fd = shm_open(SHM_FILE_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd < 0) {
        fprintf(stderr, "open file failed.\n");
        return (-1);
    }
    ftruncate(shm_fd, MAX_MSG_SIZE);
    msg_buff = mmap(NULL, MAX_MSG_SIZE, PROT_READ |
                    PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (msg_buff == MAP_FAILED) {
        fprintf(stderr, "mmap failed.\n");
        return (-1);
    }

    sem_wait(req_signal);
    fprintf(stdout, "get new request message: %s.\n", msg_buff);
    strcpy(msg_buff, "welcome");
    sem_post(ack_signal);

    munmap(msg_buff, MAX_MSG_SIZE);
}
```

```
    close(shm_fd);
    sem_close(req_signal);
    sem_close(ack_signal);
    shm_unlink(SHM_FILE_NAME);
    sem_unlink(REQ_SIG_NAME);
    sem_unlink(ACK_SIG_NAME);

    return (0);
}
```

As shown in Program List 12.4, the server-side program waits for a request from the client-side. When the server-side program receives the request, the response information is returned to the client-side. Here we use the mapped shared memory for both sides to communicate, and thus this is called the message buffer. Meanwhile in order to synchronize access to the message buffer, two POSIX named semaphores are created, respectively for the request message and the notification event for the reply message. It can also be seen from above that all the resources are created and destroyed by the server-side, which is the most common way of processing applications. It should be noted that after creating the file, we use the `ftruncate` function to adjust the file size. As mentioned above, if the file length is 0, the `mmap` function will fail.

In the previous examples, we changed the disk file size by calling the standard I/O function `write`. But for the files on the shared memory devices, we cannot guarantee all the systems to be able to use the `write` function (using the `write` function in SylixOS to operate the file on the memory device will return the error directly). For better portability, the application should use the `ftruncate` function to resize the file.

Program List 12.5 Sample of Client-Side Program of Shared Memory

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
#include <string.h>
#include <stdio.h>

#define MAX_MSG_SIZE    32
#define REQ_SIG_NAME    "req_signal"
#define ACK_SIG_NAME    "ack_signal"
#define SHM_FILE_NAME  "msg_buffer"

int main(int argc, char *argv[])
{
    char    *msg_buff;
    sem_t   *req_signal;
    sem_t   *ack_signal;
```

```
int    shm_fd;

req_signal = sem_open(REQ_SIG_NAME, 0, 0666, 0);
if (!req_signal) {
    fprintf(stderr, "open request signal failed.\n");
    return (-1);
}
ack_signal = sem_open(ACK_SIG_NAME, 0, 0666, 0);
if (!ack_signal) {
    fprintf(stderr, "open acknowledge signal failed.\n");
    return (-1);
}
shm_fd = shm_open(SHM_FILE_NAME, O_RDWR, 0666);
if (shm_fd < 0) {
    fprintf(stderr, "open shm file failed.\n");
    return (-1);
}
msg_buff = mmap(NULL, MAX_MSG_SIZE, PROT_READ |
                PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (msg_buff == MAP_FAILED) {
    fprintf(stderr, "mmap failed.\n");
    return (-1);
}

strcpy(msg_buff, "request login");
sem_post(req_signal);
sem_wait(ack_signal);
fprintf(stdout, "get acknowledge message: %s.\n", msg_buff);

munmap(msg_buff, MAX_MSG_SIZE);
close(shm_fd);
sem_close(req_signal);
sem_close(ack_signal);

return (0);
}
```

The client-side program is relatively simple. Assume that the relevant resources already exist, and you only need to open them and close the relevant resources just before the program exits. The `msg_buff` mapped in the client-side and the `msg_buff` pointer mapped in the server-side are actually pointing to the same physical memory block (this refers to the logical or several physical memory blocks with discontinuous address). Therefore, the process of sending a message here only needs to rewrite the message buffered data, and then send the synchronization signal. If you use message queues, there is bound to be a data copy process. If the amount of message data is large,

it will consume a lot of memory. A large number of data copies can also greatly reduce operational efficiency.

Finally, let's look at the operation results. Firstly, operate the server-side program using the background execution and there is no output at this point. Then, the operate client-side program and the terminal outputs the following information. It can be seen that the server-side and client-side correctly implement information interaction through the shared memory.

```
# ./shm_test
get new request msg: request login.
get acknowledge: welcome.
```

3. Anonymous Mapping

Anonymous mapping is the mapped virtual memory without any associated device files. In systems that support the fork system call (e.g. Unix and Linux etc.), because the child process inherits the virtual memory mapping of the parent process, the shared memory between the two is also possible. Currently, though SylixOS has the concept of parent-child process, it does not support the fork system call. The parent-child process is just a logical connection, and thus you cannot use this to realize the shared memory between parent and child processes.

Another purpose of anonymous mapping is to allocate virtual memory for applications, which is similar to malloc in performance, but there is a difference between the two, as shown in Figure 12.2:

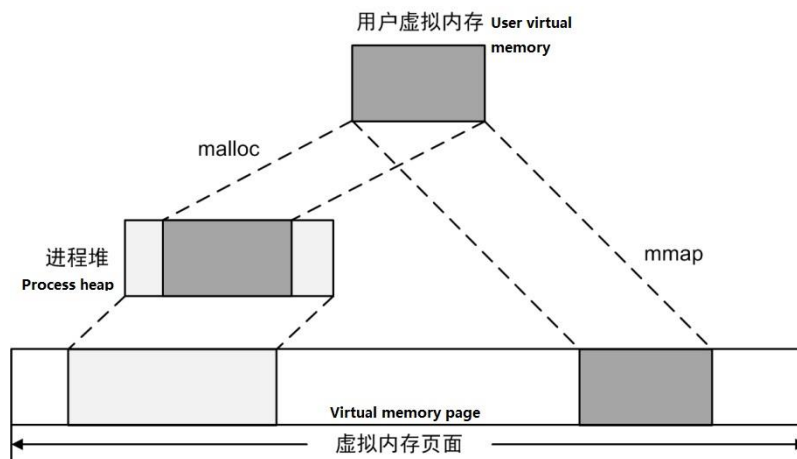


Figure 12.2 Use mmap to Allocate Memory Pages

The memory allocated by the application using the malloc function comes from the memory heap of the process itself, which is allocated from the virtual memory page by the operating system when the process is created. The memory allocated using the mmap function comes directly from the virtual memory page area. In addition, the malloc function allocates memory in bytes, and the mmap function uses the page mode to allocate

memory. Typically, the latter has a higher memory allocation efficiency. The usage of anonymous mapping is shown below:

```
mmap(NULL, BUF_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,  
-1, 0);
```

At this point, we pass in the file descriptor parameter as -1, and add the MAP_ANONYMOUS option. There is no difference between this and the usual usage. The function returns the mapped virtual space when it succeeds. The virtual space does not have the corresponding physical memory at the beginning. Only when the actual access is done, the operating system allocates the appropriate physical memory pages for the missing page interrupts.

12.4.4 Other Operations of Virtual Memory

1. Lock Memory

```
#include <sys/mman.h>  
int mlock(const void *pvAddr, size_t stLen);  
int munlock(const void *pvAddr, size_t stLen);  
int mlockall(int iFlag);  
int munlockall(void);
```

When the physical memory size cannot meet the needs of current application, some operating systems allow free physical memory pages owned by some applications to be freed for use by other programs, and the freed data is swapped to disk space (the free term here refers to the different page switching algorithms unused for a long time, and the time is differently defined and processed). The function mlock is used to lock a segment of virtual memory in the user space to avoid the corresponding physical memory being swapped to disk space by the operating system. munlock is used to remove memory locking meanwhile mlockall locks all the virtual space in the process.

Page swapping has its inherent advantages but it also brings uncertainty to the application, reduces disk usage efficiency and service life, which is generally unacceptable in embedded systems. Therefore, most embedded systems do not implement page swapping algorithms, and SylixOS either. The application using the above function will not make any sense.

2. Set Memory Protection Attribute

```
#include <sys/mman.h>  
int mprotect(void *pvAddr, size_t stLen, int iProt);
```

Prototype analysis on function mprotect:

- The function returns 0 when it succeeds and returns error number when it fails;

- The parameter ***pvAddr*** is the virtual page address;
- The parameter ***stLen*** is the virtual page size with byte as unit;
- The parameter ***iProt*** is the page protection option as shown in Table 12.2.

This function allows the application to specify attributes that require access to memory. The mprotect can implement some special features that the application does not normally require.

3. Memory Advice

Memory advice refers to an application that tells the operating system that it will use a specified range of memory space in a particular mode. It is advised that the operating system should optimize the resource management related to this memory to improve system performance according to the information.

```
#include <sys/mman.h>
int posix_madvise(void *addr, size_t len, int advice);
```

Prototype analysis on function posix_madvise:

- The function returns 0 when it succeeds and returns error number when it fails;
- The parameter ***addr*** is virtual memory address;
- The parameter ***len*** is the memory size with byte as unit;
- The parameter ***advice*** is the memory advice options as shown in Table 12.6.

Table 12.6 Memory Advice Options

Option Name	Explanation
POSIX_MADV_NORMAL	There is no advice and the operating system manages memory related resources by default.
POSIX_MADV_RANDOM	The application will randomly access the specified memory.
POSIX_MADV_SEQUENTIAL	The application will access the specified memory from a low address to a high address.
POSIX_MADV_WILLNEED	The application will access the specified memory in the near future.
POSIX_MADV_DONTNEED	The application will no longer access the specified memory.

Based on the above options, they have a lot to do with memory management algorithms, such as page recycling and page swapping etc. But using this function requires an application to specify the usage of memory to achieve the intended purpose. Currently, using this function in SylixOS will return 0 directly.

Chapter 13 Standard I/O Devices

13.1 /dev/null

A null device is a special device file, and usually called a "black hole". It is equivalent to a write-only file, and all the contents written into it will be lost forever, but the number of written bytes can be successfully returned. In SylixOS, this device does not support being read. However, it is very useful for command-lines and scripts. Here is an example of using the device in the command-line:

```
# cat file
This is test sylixos string functions example.
# cat file >/dev/null
```

The upper **cat** command outputs the contents of the "file" to the standard output and the lower **cat** command re-send the output contents to the "/dev/null" device so that the contents of the file are lost forever, and therefore there is not any content is displayed in the terminal. We can also re-send the standard error to the "/dev/null" device with the following method.

```
# ll abc 2>/dev/null
```

13.2 /dev/zero

A zero device is also a special device file. It can be regarded as a storage device with an infinite capacity and the data in it is always 0. Reading the device can obtain infinite data with the content of 0 (NULL, ASCII NUL and 0x00). Writing to this device will not have any effect on its contents.

We can use this device to create a null file with a specified length for initialization, for example: a temporary exchange file. In described in Chapter 12 Memory Management, we can use mmap's anonymous mapping to allocate virtual memory. Similarly, we can also use mmap to create a virtual mapping with the zero device, and the memory with an initial content of 0 can be allocated.

13.3 Terminal

Terminal device, is also known as tty device. The word "tty" is derived from Teletypes, or teletypewriters, originally referring to a teletypewriter, which was the medium for reading and sending information through the serial cable using the printer's keyboard, and

later was replaced by a keyboard and a display, so it is more appropriate to be named as a terminal now. A terminal is a character-type device, and it is usually used for human-computer interaction. For example, if we start a shell interface through a serial port after starting the system, then the terminal is namely a serial terminal.

In SylixOS, the names of common serial terminal devices are /dev/ttyS0, /dev/ttyS1, etc., and the names of USB serial-port devices are /dev/ttyUSB0, /dev/ttyUSB1, and so on. Normally, /dev/ttyS0 is used for the default shell service of the system and other serial devices can be used for general communication.

The following program shows the general use method of serial devices.

Program List 13.1 Serial Port Test Program

```
#include <SylixOS.h>
#include <fcntl.h>
#include <stdio.h>

#define SERIL_BUF_SIZE      512
#define SERIL_DEV_NAME     "/dev/ttyS1"

int main(int argc, char *argv[])
{
    int      iFd;
    char     pcBuff[SERIL_BUF_SIZE];
    char     pc      int      iRet;
    ssize_t  sstReadLen;
    ssize_t  sstWriteLen;

    iFd = open(SERIL_DEV_NAME, O_RDWR);
    if (iFd < 0) {
        fprintf(stderr, "open %s failed.\n", SERIL_DEV_NAME);
        return (-1);
    }
    /*
     * Baud rate : 9600
     * Hardware options 为: 8bits data_bit, 1bit stop_bit, No parity
     */
    iRet = ioctl(iFd, SIO_BAUD_SET, 9600);
    if (iRet != 0) {
        goto __error;
    }
    iRet = ioctl(iFd, SIO_HW_OPTS_SET, CS8);
    if (iRet != 0) {
        goto __error;
    }
}
```



```
iRet = ioctl(iFd, FIORBUFSET, SERIL_BUF_SIZE);
if (iRet != 0) {
    goto __error;
}

while (1) {
    sstReadLen = read(iFd, pcBuff, SERIL_BUF_SIZE);
    if (sstReadLen < 0) {
        fprintf(stderr, "read error.\n");
        goto __error;
    }
    if (sstReadLen == 0) {
        continue;
    }

    sstWriteLen = write(iFd, pcBuff, sstReadLen);
    if (sstWriteLen < 0) {
        fprintf(stderr, "write error.\n");
        goto __error;
    }
    if (sstWriteLen < sstReadLen) {
        printf("write data may be lost.\n");
    }
}

close(iFd);
return (0);

__error:
close(iFd);
return (-1);
}
```

The above is a simple serial echo test program. `/dev/ttyS0` is already used by Shell, so `/dev/ttyS1` is used for the test here. Before reading and writing the serial port, we need to set the communication parameters, such as baud rate, data bit, stop bit and enable check or not. These parameters must be exactly the same as those on the other end of the communication. The relevant commands for setting the serial communication parameters are located in the `<SylixOS/system/util/sioLib.h>` file. Since the default buffer zone of the serial device is limited in size, it may not be able to meet the requirements of a single transmission. If the sender sends the data too fast, the data that is too late to be read will be overwritten, so the program uses the `FIORBUFSET` (`ioctl` command) command to set the size of the receiving buffer zone, which is also a problem that must be considered when using serial communication. `FIORBUFSET` is located in the `<SylixOS/system/include/s_option.h>` file, in which almost all device control options are

defined. The operations about the device buffer also include FLOWBUFSET, FIORFLUSH, FLOWFLUSH, etc., used to set the size of the sending buffer zone and empty the reading and writing buffer zones.

In Linux, the applications typically use the termios assembly to operate tty devices. SylixOS is compatible with most of the termios operations to improve the program's compatibility. To use termios to operate the serial port, we need to include the following header files in the source file:

```
#include <termios.h>
```

13.4 Virtual Terminal

A virtual terminal, is called pty (pseudo-tty), namely a pseudo terminal. It is a terminal device that is virtualized by the system, usually used for the remote login service of the system. A virtual terminal contains two I/O devices, which are called the host device end and the device end respectively. The host device end is viewed from the perspective of the local system. It is a tty device that is the same as the serial device and behaves exactly like all the other tty devices. The device end is viewed from the perspective of a remote device. It can be regarded as an intermediate device that connects the host device end to the remote end communication port. The system simulates it as a device with the same serial hardware behavior in terms of internal implementation. The Telnet in SylixOS uses the virtual terminal device. If we use the **devs** command after using Telnet to log in to the system, we will see the following information (only the pty devices are listed):

```
# devs
device show (minor device) >>
drv open name
 14   1  /dev/pty/9.dev
 15   1  /dev/pty/9.hst
```

At this point, there are two pty devices that have been turned on. The number 9 before the device name is the unique identifier of a pty device, and the paired device and host have the same identifier.

13.5 Graphic Device

A graphic device, is also known as FrameBuffer device, through which, the video memory itself can be operated directly. The name of a graphic device in SylixOS is /dev/fb0. If the hardware supports multiple layers, the devices such as /dev/fb1 and /dev/fb2 will exist accordingly. Before using a graphic device, we need to first obtain its information related to the display mode, such as the resolution, the byte size occupied by each pixel, and the RGB encoding structure, the capacity of the video memory, etc., so that we can write the image data to be displayed into the video memory correctly. In SylixOS, the structure definition used to describe the graphic device information is located at <SylixOS/system/device/graph/gmemDev.h> as shown in Program List 13.2

Program List 13.2 Graphic Device Information Description Structure

```

typedef struct {
    ULONG          GMVI_ulXRes;          /* Visible area          */
    ULONG          GMVI_ulYRes;

    ULONG          GMVI_ulXResVirtual;   /* Virtual area          */
    ULONG          GMVI_ulYResVirtual;

    ULONG          GMVI_ulXOffset;       /* Display area offset   */
    ULONG          GMVI_ulYOffset;

    ULONG          GMVI_ulBitsPerPixel;  /* Number of data bits per pixel
    */
    ULONG          GMVI_ulBytesPerPixel; /* Number of bytes per pixel
    */

    /* Some graphics processors DMA for
alignment      */

    /* Used to pad invalid bytes */
    ULONG          GMVI_ulGrayscale;     /* Gray scale            */

    ULONG          GMVI_ulRedMask;       /* Red mask              */
    ULONG          GMVI_ulGreenMask;     /* Green mask            */
    ULONG          GMVI_ulBlueMask;      /* Blue mask             */
    ULONG          GMVI_ulTransMask;     /* Transparency mask     */

    LW_GM_BITFIELD GMVI_gmbfRed;        /* true color bitfield  */
    LW_GM_BITFIELD GMVI_gmbfGreen;
    LW_GM_BITFIELD GMVI_gmbfBlue;
    LW_GM_BITFIELD GMVI_gmbfTrans;

    BOOL          GMVI_bHardwareAccelerate; /* Whether to use hardware
acceleration      */
    ULONG          GMVI_ulMode;          /* Display mode          */
    ULONG          GMVI_ulStatus;        /* Display status        */
} LW_GM_VARINFO;
typedef LW_GM_VARINFO *PLW_GM_VARINFO;

typedef struct {
    PCHAR          GMSI_pcName;          /* Display name          */
    ULONG          GMSI_ulId;            /* ID                    */
    size_t         GMSI_stMemSize;       /* framebuffer memory size
    */
    size_t         GMSI_stMemSizePerLine; /* The memory size of each row
    */
    caddr_t        GMSI_pcMem;          /* Display memory (requires driver
mapping)          */
}

```

```

    } LW_GM_SCRINFO;
typedef LW_GM_SCRINFO *PLW_GM_SCRINFO;

```

LW_GM_VARINFO contains information that is closely related to the display data, of which, the most important is the pixel's RGB mask and the number of data bits it occupies. The numbers of data bits include the following:

- 8-bit: A maximum of 256 colors can be displayed. If the hardware only supports black and white display, then one pixel can support 256 gray-scale values. If the hardware supports color display, the 256 coded values usually correspond to the 256 colors most commonly used in life. This is the palette mode, which uses a limited number of colors to approximately express the actual display requirements.
- 16-bit: A maximum of 65,536 colors can be displayed, and it is also called pseudo-true color. The hardware supporting 16-bit color can display the vast majority of colors in life. With 16-bit data display, there are two encoding methods, i.e. RGB555 and RGB565, which can be obtained by the above GMVI_ulRedMask, GMVI_ulGreenMask and GMVI_ulBlueMask;
- 24-bit: Up to 16 million colors can be displayed, and it is almost impossible to distinguish its difference from the actual color with the naked eye, so it is also called true color. With the 24-bit display, the red, green, and blue colors of a pixel are respectively expressed by 8 bits;
- 32-bit: The additional 8 bits relative to 24-bit are used to express the 256 transparency scale of the pixel (0 means opaque, 255 means fully transparent, and the pixel is not displayed in this case). Through the use of GMVI_ulTransMask, we can know the position of the pixel transparency value.

In most embedded systems, the display controller usually supports 8-bit or 16-bit data display, and only some high-end processors support 24-bit or 32-bit true color display. Normally, one pixel occupies 1 byte of memory in the case of 8-bit data display, 16-bit occupies 2 bytes of memory, 24-bit occupies 3 bytes of memory, and 32-bit occupies 4 bytes of memory accordingly. Due to alignment restrictions of some types of hardware on DMA memory, however, it is required to pad bytes between the memories occupied by pixels to meet the alignment needs. Therefore, in actual use, we should set for pixels using GMVI_ulBitsPerPixel and RGB mask values, and process the memory offset between pixels using GMVI_ulBytesPerPixel.

LW_GM_SCRINFO contains some necessary information for video memory. GMSI_stMemSize denotes the total byte size of the video memory. GMSI_stMemSizePerLine denotes the size of the byte occupied by each line. Based on this information, we can know how to process the memory offset between lines and know the total number of columns.

Normally, we can correctly achieve the display the image based on the above information. In addition, there is some other information, such as video area, virtual area,

display area offset, hardware acceleration and so on. The following program simply shows how to use a graphic device.

Program List 13.3 Graphic Device Operation Example

```

#include <SylixOS.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>

static VOID __drawPixel (VOID          *pvFrameBuffer,
                        LW_GM_SCRINFO  *pscrinfo,
                        LW_GM_VARINFO  *pvarinfo,
                        INT             iX,
                        INT             iY,
                        UINT32         uiColor)
{
    VOID *pvPixelAddr;

    if ((iX < 0) || (iX >= pvarinfo->GMVI_ulXResVirtual) ||
        (iY < 0) || (iY >= pvarinfo->GMVI_ulYResVirtual)) {
        return;
    }

    pvPixelAddr = (UINT8 *)pvFrameBuffer + iY *
pscrinfo->GMSI_stMemSizePerLine;

    switch (pvarinfo->GMVI_ulBitsPerPixel) {
    case 16:
        if (pvarinfo->GMVI_ulGreenMask == (0x3f << 5)) { /* RGB565 mode
*/
            uiColor = ((uiColor & 0xff0000) >> 16 >> 3 << 11) |
                ((uiColor & 0x00ff00) >> 8 >> 2 << 5) |
                ((uiColor & 0x0000ff) >> 0 >> 3 << 0);
        } else { /* RGB555 mode */
            uiColor = ((uiColor & 0xff0000) >> 16 >> 3 << 10) |
                ((uiColor & 0x00ff00) >> 8 >> 3 << 5) |
                ((uiColor & 0x0000ff) >> 0 >> 3 << 0);
        }
        *((UINT16 *)pvPixelAddr + iX) = (UINT16)uiColor;
        break;

    case 24:

    case 32:

```

```
        *((UINT32 *)pvPixelAddr + iX) = uiColor;
        break;
    }
}

static VOID __drawHorizLine (VOID          *pvFramebuffer,
                            LW_GM_SCRINFO *pscrinfo,
                            LW_GM_VARINFO *pvarinfo,
                            INT           iXstart,
                            INT           iYstart,
                            INT           iXend,
                            UINT32       uiColor)
{
    for (; iXstart <= iXend; iXstart++) {
        __drawPixel(pvFramebuffer, pscrinfo, pvarinfo,
iXstart, iYstart, uiColor);
    }
}

int main (int argc, char *argv[])
{
    INT           iFbFd;
    LW_GM_SCRINFO scrInfo;
    LW_GM_VARINFO varInfo;
    INT           iError;
    VOID          *pvFramebuffer;

    iFbFd = open("/dev/fb0", O_RDWR);
    if (iFbFd < 0) {
        fprintf(stderr, "open /dev/fb0 failed.\n");
        return (-1);
    }
    iError = ioctl(iFbFd, LW_GM_GET_SCRINFO, &scrInfo);
    if (iError < 0) {
        fprintf(stderr, "get /dev/fb0 screen info failed.\n");
        goto __error;
    }
    iError = ioctl(iFbFd, LW_GM_GET_VARINFO, &varInfo);
    if (iError < 0) {
        fprintf(stderr, "get /dev/fb0 var info failed.\n");
        goto __error;
    }

    pvFramebuffer = mmap(LW_NULL, scrInfo.GMSI_stMemSize,
```

```
        PROT_READ | PROT_WRITE, MAP_SHARED, iFbFd, 0);
if (pvFramebuffer == MAP_FAILED) {
    fprintf(stderr, "mmap /dev/fb0 failed.\n");
    goto __error;
}

memset(pvFramebuffer, 0xff, scrInfo.GMSI_stMemSize); /* Clear screen */

__drawHorizLine(pvFramebuffer, &scrInfo, &varInfo, 0, 10,
                varInfo.GMVI_ulXResVirtual - 1, 0xff0000);
__drawHorizLine(pvFramebuffer, &scrInfo, &varInfo, 0, 20,
                varInfo.GMVI_ulXResVirtual - 1, 0x00ff00);
__drawHorizLine(pvFramebuffer, &scrInfo, &varInfo, 0, 30,
                varInfo.GMVI_ulXResVirtual - 1, 0x0000ff);

munmap(pvFramebuffer, scrInfo.GMSI_stMemSize);
close(iFbFd);
return (0);

__error:
close(iFbFd);
return (-1);
}
```

In the main function, we first obtain the information related to the display, and then use `mmap` to map the video memory to the user's virtual space, thus we operate the virtual space is equivalent to direct operation of the video memory. For details about `mmap`, see Section 12.4 Management of Virtual Memory.

In the `__drawPixel` function, the parameter `uiColor` is defined as a type of 32-bit data, starting from the lower address to the higher address, byte 0 denotes blue, byte 1 denotes green and byte 2 denotes red, which is the same as the true color (24-bit or 32-bit) in terms of color format. For a 16-bit data pattern, we can know the current RGB mode just through the green mask value and can perform the corresponding conversion according to the corresponding mode. Note that since all the true colors can not be fully expressed in the 16-bit mode, a corresponding linear conversion process is performed, such as 0 to 255. For the 5-bit data, it corresponds to 0 to 31, and for a 6-bit data, it corresponds to 0 to 63. The focus of this example is to show how to use a graphics device, so only three horizontal lines are drawn. The reader can understand the relevant algorithm for drawing any line, circle or ellipse through other ways.

Due to space limitations, and in order to make the program simple and intuitive, many details are omitted. For example, the actual byte size occupied by the pixel is not taken into account, and it is just assumed to be the same as the number of its data bits; the offset between the real display area and the virtual display area is not considered as well, and it is just assumed as 0; the 8-bit data display is not involved in the program; and, the

pixel transparency information in the 32-bit mode is also ignored. The reader must understand that this information must be properly processed in practical applications.

Generally, the application indirectly operates the video memory device through GUI assembly. The GUI itself processes all of the above information. Only in some special situations (for example, a higher display efficiency is required, or there are some simple graphic applications), it is necessary to directly operate the video memory itself.

13.6 Input Device

The input device usually includes a mouse, a touch screen, and a keyboard, as well as input devices with special functions such as joysticks and tablet. SylixOS defines the supported input devices in the SylixOS/system/device/input/ directory. Currently, only the mouse and keyboard are defined. They are located in <mouse.h> and <keyboard.h> respectively.

13.6.1 Mouse Device

The mouse driver notifies the system of the occurrence of a mouse event by reporting the data of a `mouse_event_notify` structure. The structure is defined as follows:

Program List 13.4 Structure of Mouse Event

```
typedef struct mouse_event_notify {
    int32_t      ctype;          /* coordinate type          */

    int32_t      kstat;         /* mouse button stat       */
    int32_t      wscroll[MOUSE_MAX_WHEEL]; /* wheel scroll            */

    int32_t      xmovement;
    int32_t      ymovement;

    /*
     * if use absolutely coordinate (such as touch screen)
     * if you use touch screen:
     * (kstat & MOUSE_LEFT) != 0 (press)
     * (kstat & MOUSE_LEFT) == 0 (release)
     */

#define xanalog      xmovement          /* analog samples values */
#define yanalog      ymovement
} mouse_event_notify;
```

Ctype is used to distinguish the coordinate type of a device, which can be `MOUSE_CTYPE_REL` or `MOUSE_CTYPE_ABS`, namely relative coordinates (general mouse device) or absolute coordinates (touch screen device).

Kstat is used to identify the state of the mouse key, including the left key, middle key, and right key. In addition, it defines additional key state that can be used to satisfy specific mouse (such as gaming mouse) applications. Each key state is denoted by a data bit, 0 denotes being released and 1 denotes being pressed. The key state is defined as follows:

Table 13.1 Definition of Mouse Key Status Bit

Key State	Description
MOUSE_LEFT	Left mouse key
MOUSE_RIGHT	Right mouse key
MOUSE_MIDDLE	Middle mouse key
MOUSE_BUTTON4 ~ MOUSE_BUTTON7	Additional 4 predefined keys

When the device is a touch screen, the left mouse key status bit is used to indicate the information that it is released and pressed, as noted in the Program List 13.4.

Xmovement and ymovement denote the relative displacement values of the mouse. When it is an absolute coordinate, the system suggests the program to use xanalog and yanalog (although they are the same member variable as xmovement and ymovement) so that the program is more intuitive and readable.

In SylixOS, the mouse device names are /dev/input/mse0, /dev/input/mse1, etc., and the touch screen device names are /dev/input/touch0, /dev/input/touch1, and so on. General operation methods are demonstrated below by reading mouse device events. The operation methods of the touch screen device are similar to these.

Program List 13.5 Mouse Application Example

```
#include <stdio.h>
#include <fcntl.h>
#include <mouse.h>

#define MOUSE_DEV_NAME    "/dev/input/mse0"
#define MOUSE_READ_CNT    50

int main(int argc, char *argv[])
{
    int                mse_fd;
```

```
mouse_event_notify mse_event;
ssize_t read_len;
int read_cnt = 0;

mse_fd = open(MOUSE_DEV_NAME, O_RDONLY);
if (mse_fd < 0) {
    fprintf(stderr, "open %s failed.\n", MOUSE_DEV_NAME);
    return (-1);
}

while (read_cnt++ < MOUSE_READ_CNT) {
    read_len = read(mse_fd, (void *)&mse_event, sizeof(mouse_event_notify));
    if (read_len < 0) {
        fprintf(stderr, "read mouse event error, abort.\n");
        break;
    }
    if (read_len < sizeof(mouse_event_notify)) {
        fprintf(stderr, "read mouse event invalid, continue.\n");
        continue;
    }

    fprintf(stdout, "mouse report [%d] >>\n", read_cnt);
    fprintf(stdout, "key : ");
    if (mse_event.kstat & MOUSE_LEFT) {
        fprintf(stdout, "left ");
    }
    if (mse_event.kstat & MOUSE_RIGHT) {
        fprintf(stdout, "right ");
    }
    if (mse_event.kstat & MOUSE_MIDDLE) {
        fprintf(stdout, "middle ");
    }
    if (mse_event.kstat == 0) {
        fprintf(stdout, "none");
    }
    fprintf(stdout, "\n");
    fprintf(stdout, "move : x: %d, y: %d\n", mse_event.xmovement,
        mse_event.ymovement);
    fprintf(stdout, "wheel: %d[%s]\n", mse_event.wscroll[0],
        mse_event.wscroll[0] == 0 ? "none" :
        mse_event.wscroll[0] > 0 ? "up" : "down");
    fprintf(stdout, "\n");
}
```

```
close(mse_fd);
return (0);
}
```

In the above programs, up to 50 mouse events will be read. After run the program, the following information may be printed:

```
mouse report [2] >>
key : left right middle
move : x: -1, y: 3
wheel: 1[up]

mouse report [28] >>
key : left right middle
move : x: 4, y: 1
wheel: -1[down]
```

The above structure requires such operations as pressing the left, middle (pulley), and right three keys of the mouse simultaneously, and scrolling the pulley up or down while moving the mouse. This shows that a mouse event can pass multiple messages at the same time. If it is a touch screen device, there is no key information, but the released or pressed state.

When there are multiple mouse devices in the system (multiple USB devices or touch screens exist at the same time), the application does not need to handle the events of these multiple devices separately. This is because SylixOS provides a standard kernel module called `xinput.ko`. After the module is registered, two devices will be created, namely `/dev/input/xmse` and `/dev/input/xkbd`, which will collect all the mouse and keyboard events in the system. The application will only need to read the two devices. For `xmse` devices, there are general mouse messages and touch screen messages, so the application need to handle them separately. The general operation of the `xmse` device is as follows:

Program List 13.6 Handling Mouse Event Pseudocode Using `xmse` Devices

```
mse_fd = open(/dev/input/xmse, O_RDONLY);
read(mse_fd, &mse_event, ...);
if (mse_event.ctype == MOUSE_CTYPE_REL) {
    /*
     * Handle normal mouse events
     */
} else {
    /*
     * Handling touch screen events
     */
}
```

13.6.2 Keyboard Device

The keyboard device driver notifies the system of the occurrence of a keyboard event by reporting the data of a `keyboard_event_notify` structure. The structure is defined as follows:

Program List 13.7 Structure of Keyboard Events

```
typedef struct keyboard_event_notify {
    int32_t      nmsg;                /* message num, usually one msg*/
    int32_t      type;                /* press or release          */
    int32_t      ledstate;            /* LED stat                   */
    int32_t      fkstat;              /* func-key stat              */
    int32_t      keymsg[KE_MAX_KEY_ST]; /* key code                    */
} keyboard_event_notify;
```

The value of `type` is either `KE_PRESS` or `KE_RELEASE`, which means being pressed or released respectively.

`ledstate` is used to indicate the state of the key with the LED indicator. If the corresponding bit is 0, it indicates the LED key is in the On state, otherwise it is in the Off state. In the On state, the keyboard driver usually turns on the corresponding LED indicator, the status bits of these keys are defined as follows:

Table 13.2 LED Key Status Bits

Status Bit	Description
<code>KE_LED_NUMLOCK</code>	Used to indicate whether the numeric keypad is On.
<code>KE_LED_CAPSLOCK</code>	Used to indicate whether the capitalization of the alphabet is enabled.
<code>KE_LED_SCROLLLOCK</code>	Used to indicate whether the scroll lock state is enabled.

`fkstate` is used to indicate the state of the function key. If the corresponding bit is 0, it indicates the function key is pressed, and if not, it is not pressed. The status bits of all function keys are defined as follows:

Table 13.3 Function Key Status Bit

Status Bit	Description
<code>KE_FK_CTRL</code>	Used to indicate whether the left Ctrl key is pressed.
<code>KE_FK_ALT</code>	Used to indicate whether the left Alt key is pressed.
<code>KE_FK_SHIFT</code>	Used to indicate whether the left Shift key is pressed.
<code>KE_FK_CTRLR</code>	Used to indicate whether the right Ctrl key is pressed.
<code>KE_FK_ALTR</code>	Used to indicate whether the right Alt key is pressed.
<code>KE_FK_SHIFTR</code>	Used to indicate whether the right Shift key is pressed.

nmsg and keymsg are used to indicate the codes of all common keys except the LED keys and function keys. The macro KE_MAX_KEY_ST is currently defined as 8, which means that the system allows the keyboard driver to report up to 8 common key messages at a time. Note that the capitalization of alphabetic keys is not processed by the driver. For example, if we press the Shift key and then press the letter key A, the code in keymsg will be the character 'a', and the KE_FK_SHFT status bit in fkstate will be 1. In the case of enabling capslock, the code in keymsg is still 'a', but the KE_LED_CAPSLOCK status bit in ledstate is 1. state Therefore, the application needs to perform proper case conversion of characters based on this information. Some other keys that are affected by the Shift state, such as the number keys 1 to 9 on the main keyboard, when the shift is pressed, the code in the keymsg is correspondingly " !,@ and #". The following program shows the general method to process the keyboard messages through the /dev/input/xkbd device.

List of Program 13.8 Example of Processing Keyboard Messages

```
#include <stdio.h>
#include <fcntl.h>
#include <ctype.h>
#include <keyboard.h>

#define KBD_DEV_NAME      "/dev/input/xkbd"
#define KBD_READ_CNT     50

int main(int argc, char *argv[])
{
    int                kbd_fd;
    keyboard_event_notify  kbd_event;
    ssize_t           read_len;
    int               read_cnt = 0;
    int32_t           keymsg;
    int               i;

    kbd_fd = open(KBD_DEV_NAME, O_RDONLY);
    if (kbd_fd < 0) {
        fprintf(stderr, "open %s failed.\n", KBD_DEV_NAME);
        return (-1);
    }

    while (read_cnt++ < KBD_READ_CNT) {
        read_len = read(kbd_fd, (void *)&kbd_event,
            sizeof(keyboard_event_notify));
        if (read_len < 0) {
            fprintf(stderr, "read keyboard event error, abort.\n");
            break;
        }
    }
}
```

```
    }
    if (read_len < sizeof(keyboard_event_notify)) {
        fprintf(stderr, "read keyboard event invalid, continue.\n");
        continue;
    }

    fprintf(stdout, "keyboard report [%d] >>\n", read_cnt);
    for (i = 0; i < kbd_event.nmsg; i++) {
        keymsg = kbd_event.keymsg[i];
        fprintf(stdout, "key code: <%3d [%c] (%s)> ",
            keymsg,
            isprint(keymsg) ? keymsg : ' ',
            kbd_event.type == KE_PRESS ? "press" : "release");
    }

    fprintf(stdout, "\nled sta : ");
    if (kbd_event.ledstate == 0) {
        printf("none");
    }
    if (kbd_event.ledstate & KE_LED_NUMLOCK) {
        fprintf(stdout, "numlock ");
    }
    if (kbd_event.ledstate & KE_LED_CAPSLOCK) {
        fprintf(stdout, "capslock ");
    }
    if (kbd_event.ledstate & KE_LED_SCROLLLOCK) {
        fprintf(stdout, "scrollock ");
    }

    fprintf(stdout, "\nfunc key: ");
    if (kbd_event.fkstat == 0) {
        fprintf(stdout, "none");
    }
    if (kbd_event.fkstat & KE_FK_CTRL) {
        fprintf(stdout, "left-ctrl ");
    }
    if (kbd_event.fkstat & KE_FK_ALT) {
        printf("left-alt ");
    }
    if (kbd_event.fkstat & KE_FK_SHIFT) {
        fprintf(stdout, "left-shift ");
    }
    if (kbd_event.fkstat & KE_FK_CTRLR) {
        fprintf(stdout, "right-ctrl ");
    }
}
```

```

    }
    if (kbd_event.fkstat & KE_FK_ALTR) {
        fprintf(stdout, "right-alt ");
    }
    if (kbd_event.fkstat & KE_FK_SHIFTR) {
        fprintf(stdout, "right-shift ");
    }
    fprintf(stdout, "\n\n");
}

close(kbd_fd);
return (0);
}

```

The above program just prints out the original keyboard message obtained without any conversion processing. The reader can use this program to test the difference between the messages in the case of different key combinations. In normal circumstances, the original message input into the device is not processed by the application directly, but analyzed and processed by the GUI layer, where it is converted into higher-level message description for use by the program.

13.7 Memory Device

The memory device drives the program to access the memory as if it were accessing a virtual I/O device. The creation of the memory device does not currently provide support for the application layer (requires macro `__SYLIXOS_KERNEL` to increase the support to the creation of the memory device). After a memory device is created in the kernel layer, the application can use standard I/O functions to read and write the memory device. The following is a list of functions for creating the memory devices the kernel layer:

```

#define __SYLIXOS_KERNEL
#include <SylixOS.h>
INT  API_MemDrvInstall(void);
INT  API_MemDevCreate(char *name, char *base, size_t length);
INT  API_MemDevCreateDir(char *name, MEM_DRV_DIRENTRY *files, int numFiles);
INT  API_MemDevDelete(char *name);

```

Before creating a memory device, we need to first call the `API_MemDrvInstall` function to install the memory device driver. An memory device can correspond to a file/device or multiple files/devices and be created through different functions.

The `API_MemDevCreate` function can create a single-file memory device, as shown in Program List 13.9, and the use method is as shown in Program List 13.10.

Program List 13.9 Creation of the Memory Device(Module)

```

#define __SYLIXOS_KERNEL

```

```
#include <SylixOS.h>
#include <module.h>

char membase[64] = "SylixOS mem device test.";

int module_init (void)
{
    API_MemDrvInstall();
    API_MemDevCreate("/dev/mem", membase, 64);

    return (0);
}

void module_exit (void)
{
    API_MemDevDelete("dev/mem");
}
```

Load the kernel module using the following command:

```
# modulreg memdev.ko
```

Note: See Chapter 19 Dynamic Load for details on the Kernel Module.

Program List 13.10 Reading of the Memory Device

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    int    fd;
    char   buf[64];

    if (argc < 2) {
        fprintf(stderr, "%s [memdev].\n", argv[1]);
        return (-1);
    }
    fd = open("/dev/mem", O_RDWR);
    if (fd < 0) {
        return (-1);
    }

    read(fd, buf, 64);
    fprintf(stdout, "buf: %s\n", buf);

    return (0);
}
```



```
}

```

Run the program under the SylixOS Shell and the results are as follows:

```
# ./memdev_test /dev/mem
buf: SylixOS mem device test.
```

The `API_MemDevCreateDir` function can create multiple-file memory devices as shown in Program List 13.11.

Program List 13.11 Creation of Multi-file Memory Device(module)

```
#define __SYLIXOS_KERNEL
#include <SylixOS.h>
#include <module.h>

static char membase0[64] = "SylixOS mem device test.";
static char membase1[64] = "membase1 device test.";

static MEM_DRV_DIRENTRY memdirdata0[] = {"mem.dat", membase0, NULL, 64};
static MEM_DRV_DIRENTRY memdirdata1[] = {"mem.dat", membase1, NULL, 64};

static MEM_DRV_DIRENTRY memdir[] = {
    {"mem0", NULL, memdirdata0, 1},
    {"mem1", NULL, memdirdata1, 1},
};

int module_init (void)
{
    API_MemDrvInstall();
    API_MemDevCreateDir("/mem/", memdir, 2);

    return (0);
}

void module_exit (void)
{
    API_MemDevDelete("/mem/");
}
```

13.8 Random Device

entropy is a physical quantity that describes the disorder of a system. The greater the entropy of a system is, the worse the orderability of the system (the greater the uncertainty of the system) will be. In informatics, entropy is used to indicate the uncertainty of a symbol or system. The greater the entropy is, the less information the system contains.

The computer itself is a predictable system, so it is impossible to generate real random using the computer algorithm. However, the machine environment is full of various kinds of noise, for example, the time when the hardware device is interrupted, and the time interval between the user's clicks of the mouse is completely random and cannot be predicted in advance. The random number generator implemented by the SylixOS kernel generates high-quality random sequences with such random noises in the system.

The random in SylixOS can be generated from two special files, one is `/dev/urandom` and the other is `/dev/random`. The principle that they generate random is to use the entropy pool of the current system to calculate a fixed number of random bits, and then return these bits as a byte stream.

13.9 Audio Device

OSS (Open Sound System) is a unified audio interface standard on the Unix platform. Previously, each Unix manufacturer provided a set of proprietary APIs for processing audio. This means that audio processing applications written for a Unix platform must be rewritten when ported to another Unix platform. More than that, the features available on one platform may be not available on another platform.

However, the situation is quite different after the emergence of the OSS standard. As long as the audio processing application is written in accordance with the API under the OSS standard, it only needs to be recompiled when being ported to another platform. Therefore, the OSS standard provides source code-level portability.

SylixOS supports simple OSS standards.

13.9.1 Basics

Digital audio device (sometimes called codec, PCM, DSP or ADC/DAC device): Play or record of digitized sound. Its main indicators include: sampling rate (8KHz for telephones and 96KHz for DVDs), number of channels (mono and stereo), and sampling resolution (8bit and 16bit).

- **Mixer:** used to control the volume of multiple inputs and outputs, as well as the switch between the inputs (microphone, line-in, CD).
- **Synthesizer:** used to synthesize sounds using the pre-defined waveforms, and sometimes used to produce sound effects in the games.
- **MIDI Interface:** It is a serial interface for connecting synthesizer, keyboard, props, and lighting controllers on the stage.

In the SylixOS system, the device is abstracted into a file and the access to the device is realized through the access to the file (open the file first, and then read/write it, at the same time, read/set the parameters using `ioctl`, and finally close the file). In the OSS standard, there are mainly the following device files:

- `/dev/mixer`: used to access the built-in mixer in the sound card, adjust the volume, and select the sound source;
- `/dev/sndstat`: used to test the sound card, performing the `cat /dev/sndstat` will result in the display of the information of the sound card driver information;
- `/dev/dsp`, `/dev/dspW`, `/dev/audio`: Reading this device is equivalent to recording, and writing it is equivalent to playing. The difference between `/dev/dsp` and `/dev/audio` is that the sampling code is different. `/dev/audio` uses μ -law code, `/dev/dsp` uses 8bit (unsigned) linear code, and `/dev/dspW` has linear code when using 16bit (signed). `dev/audio` is used mainly for compatibility with SunOS and is generally not recommended;
- `/dev/sequencer`: used to access the synthesizer with a built-in sound card, or connected to a MIDI interface.

The specific audio device files in SylixOS actually depend on the implementation of the underlying driver, and typically there are only the `/dev/dsp` and `/dev/mixer` devices.

13.9.2 Audio Programming

Next we will discuss turning-on, playback, recording and parameter adjustment of the audio device.

1. Definition of Header File

```
#include <ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/soundcard.h>

#define BUF_SIZE 4096
int audio_fd;
unsigned char audio_buffer[BUF_SIZE];
```

2. Opening of Audio Device

```
if ((audio_fd = open("/dev/dsp", open_mode, 0)) == -1) {
perror("/dev/dsp");
exit(1);
}
```

There are three options for `open_mode`: `O_RDONLY`, `O_WRONLY`, and `O_RDWR`, which represent read-only, write-only, and read-write, respectively. The OSS standard recommends that read-only or write-only should be used as much as possible, and the

read-write mode should only be used in the full-duplex situation (i.e. recording and playing simultaneously).

3. Recording

```
int len;
if ((len = read(audio_fd, audio_buffer, count)) == -1) {
perror("audio read");
exit(1);
}
```

Count is the number of bytes of recording data (an exponent of 2 is recommended), but it should not exceed the size of audio_buffer. The time can be measured accurately by reading the number of bytes. For example, the rate of 8KHz, 16bit stereo is $8000 \times 2 \times 2 = 32000$ bytes/second, which is the only way to know when to stop recording.

4. Playback

Playback is actually very similar to recording, except that read is changed to write, and in the corresponding audio_buffer is audio data, and count is the length of the data.

Note that users always have to read/write a complete sample. For example, in a 16-bit stereo mode, there are 4 bytes in each sample, so the application must read/write the bytes based on the number of multiples of 4 each time.

In addition, since OSS is a cross-platform audio interface, users should consider portability during programming. One of the important aspects is the byte order of read/write.

5. Setting of Sampling Format

```
int format;
format = AFMT_S16_LE;
if (ioctl(audio_fd, SNDCTL_DSP_SETFMT, &format) == -1) {
perror("SNDCTL_DSP_SETFMT");
exit(1);
}
if (format != AFMT_S16_LE) {
/*
 * This device does not support the selected sampling format
 */
}
```

Before setting the sampling format, you can first test which sampling formats the device supports. The method is as follows:

```
int mask;
```

```
if (ioctl(audio_fd, SNDCTL_DSP_GETFMTS, &mask) == -1) {
perror("SNDCTL_DSP_GETFMTS");
exit(1);
}
if (mask & AFMT_MPEG) {
/*
 * This device supports MPEG sampling format...
 */
}
```

6. Setting of the Number of Channels

```
int channels = 2; /* 1=mono, 2=stereo */
if (ioctl(audio_fd, SNDCTL_DSP_CHANNELS, &channels) == -1) {
perror("SNDCTL_DSP_CHANNELS");
exit(1);
}
if (channels != 2){
/*
 * This device does not support stereo mode...
 */
}
```

7. Setting of Sampling Rate

```
int speed = 11025;
if (ioctl(audio_fd, SNDCTL_DSP_SPEED, &speed)==-1) {
perror("SNDCTL_DSP_SPEED");
exit(1);
}
if ( /* The rate of return (ie the rate supported by the hardware) is very different
from the rate required... */ ) {
/*
 * This device does not support the required rate...
 */
}
```

The audio device generates the required sampling clock by frequency division, so it is not possible to generate all the frequencies. The driver will calculate the frequency that is closest to the requirement, and the user program will check the returned rate value. If the error is small, it can be ignored.

13.9.3 Mixer Programming

The control of Mixer includes adjusting the volume, selecting the recording source (microphone, line-in), querying the function and state of the mixer, and the control is achieved mainly through the ioctl interface of the /dev/mixer for the Mixer. Correspondingly, the functions provided by the ioctl interface are also divided into three categories: adjusting the volume, querying the ability of the mixer, and selecting the recording channel of the mixer. The following describes the use method:

The mixer_fd below is the file descriptor returned by the open operation of the mixer device.

1. Adjusting the volume

The application reads/sets the volume via the ioctl's MIXER_READ and MIXER_WRITE function number. In the OSS standard, the volume ranges from 0 to 100. Use methods are as follows:

```
int vol;
if (ioctl(mixer_fd, MIXER_READ(SOUND_MIXER_MIC), &vol) == -1) {
/*
 * Accessed an undefined mixer channel...
 */
}
```

SOUND_MIXER_MIC is a channel parameter that indicates the volume of the read microphone channel and the result is placed in vol. If the channel is stereo, then the least significant byte of vol is the left channel's volume value, followed by the right channel's volume value, and the other two bytes are unused. If the channel is mono, the left and right channels in vol have the same value.

2. Querying the Ability of Mixer

```
int mask;
if (ioctl(mixer_fd, SOUND_MIXER_READ_xxxx, &mask) == -1) {
/*
 * Mixer does not have this ability...
 */
}
```

The xxxx in SOUND_MIXER_READ_xxxx represents the content to be queried:

- Check the available mixer channels using SOUND_MIXER_READ_DEVMASK;
- Check available the recording device using SOUND_MIXER_READ_REC_MASK;
- Check mono/stereo using SOUND_MIXER_READ_STEREODEVS;
- Check the general capabilities of the mixer, using SOUND_MIXER_READ_CAPS and so on.

The result of the query for all channels is placed in the mask. Therefore, use `mask&(1 << channel_no)` to distinguish the state of a particular channel.

3. Selecting the Recording Channel of Mixer

Check the available recording channels through `SOUND_MIXER_READ_REC_MASK` first, and then select the recording channels using `SOUND_MIXER_WRITE_RECSRC`. You can query the recording channel already selected in the current sound card through `SOUND_MIXER_READ_RECSRC` at any time.

The OSS standard proposes to separate the user control functions of the mixer and make it form a general program. But the premise is that you need to check the capabilities of the sound card first through the API query function before using the mixer.

The following procedure shows the operation method to play music using the OSS standard.

```
#include "unistd.h"
#include "stdlib.h"
#include "sys/soundcard.h"

#define __OSS_TEST_BUFFER_LEN      10 * 1024
#define __OSS_TEST_WAV_FILE        "/apps/wo.wav"
#define __OSS_TEST_SAMPLE_RATE     11025
#define __OSS_TEST_CHANNELS        2
#define __OSS_TEST_SAMPLE_FORMAT   AFMT_S16_LE

int main (int argc, char *argv[])
{
    CHAR          *pcBuffer;
    CHAR          *pcPtr;
    INT           iSampleFmt;
    INT           iChannels;
    INT           iSampleRate;
    INT           iDspFd;
    INT           iFileFd;
    ssize_t      stLen;
    ssize_t      stRet;

    pcBuffer = malloc(__OSS_TEST_BUFFER_LEN);
    if (!pcBuffer) {
        printf("failed to alloc buffer!\n");
        return (-1);
    }

    iDspFd = open("/dev/dsp", O_WRONLY, 0666);
    if (iDspFd < 0) {
```

```
        printf("failed to open /dev/dsp device!\n");
        return (-1);
    }

    iSampleFmt = __OSS_TEST_SAMPLE_FORMAT;
    stRet = ioctl(iDspFd, SNDCTL_DSP_SETFMT, &iSampleFmt);
    if (stRet < 0) {
        printf("failed to set sample format!\n");
        close(iDspFd);
        return (-1);
    }

    iChannels = __OSS_TEST_CHANNELS;
    stRet = ioctl(iDspFd, SNDCTL_DSP_CHANNELS, &iChannels);
    if (stRet < 0) {
        printf("failed to set channels!\n");
        close(iDspFd);
        return (-1);
    }

    iSampleRate = __OSS_TEST_SAMPLE_RATE;
    stRet = ioctl(iDspFd, SNDCTL_DSP_SPEED, &iSampleRate);
    if (stRet < 0) {
        printf("failed to set sample rate!\n");
        close(iDspFd);
        return (-1);
    }

    iFileFd = open(__OSS_TEST_WAV_FILE, O_RDONLY, 0666);
    if (iFileFd < 0) {
        printf("failed to open test audio file %s!\n",
__OSS_TEST_WAV_FILE);
        close(iDspFd);
        return (-1);
    }

    read(iFileFd, pcBuffer, 0x2E * 2);

    while ((stLen = read(iFileFd, pcBuffer,
__OSS_TEST_BUFFER_LEN)) > 0) {

        pcPtr = pcBuffer;

        while (stLen > 0) {
```



```

        stRet = write(iDspFd, pcPtr, stLen);
    if (stRet < 0) {
        break;
    }
    pcPtr += stRet;
    stLen -= stRet;
}
}

sleep(3);
close(iFileFd);
close(iDspFd);
free(pcBuffer);

return (0);
}

```

13.10 Audio Device

In a video capture interface, there may be multiple video input sources, and there may also be multiple video output channels via video format conversion. Each channel may support different output formats, such as RGB format for direct display, or YUV or JPEG formats for compressed storage or transmission. When a user uses a specific video device, they usually do not care too much about the information of the video input source (such information is usually processed by the driver), and they are more concerned with the video output information, such as the format, size, and occupied memory of the output image. All definitions of video devices in SylixOS are located in the `<system/device/video/video.h>` header file, and all of its data structures and control commands are designed based on the video interface features described above.

13.10.1 Device Description

SylixOS uses the following structure to describe a specific video device:

```

typedef struct video_dev_desc {
    CHAR    driver[32];
    CHAR    card[32];
    CHAR    bus[32];

    UINT32  version;                /* Video driver version
    */
#define VIDEO_DRV_VERSION        1

    UINT32  capabilities;          /* ability */
}

```

```

#define VIDEO_CAP_CAPTURE      1          /* Video capture
capability*/
#define VIDEO_CAP_READWRITE    2          /* read/write System call
support */

    UINT32  sources;                    /* The number of video sources
*/
    UINT32  channels;                    /* Total number of acquisition
channels */

    .....
} video_dev_desc;

```

The structure is described as follows:

- driver is the drive name used by the video device;
- card is the name of the corresponding video interface card (video processing device);
- bus is the description information for the bus of the video interface card;
- version represents the version of the video framework that the video device's driver should follow^①;
- capabilities describes the functions of a video device. The currently-defined functions include video capture and support for the call of the read/write system.
- sources is the total number of video input sources for a video device;
- channels is the number of video capture channels, which is also the number of video output channels;
- reserve is a reserved byte and it is compatible with subsequent extension definitions.

13.10.2 Description of Device Channels

According to video_dev_desc, we can obtain the overall information of a video device, and the application is most concerned with the information of each video output channel. SylixOS describes a video output channel through the following structure:

```

typedef struct video_channel_desc {
    UINT32  channel;                    /* Specified video capture
channel */

    CHAR    description[32];            /* Instructions */
    UINT32  xsize_max;                  /* max size */
    UINT32  ysize_max;

```

```

    UINT32 queue_max; /* Maximum supported storage
sequence number */

    UINT32 formats; /* Number of supported video
capture formats */

    UINT32 capabilities; /* ability */
#define VIDEO_CHAN_ONESHOT 1 /* Only one frame is acquired
*/

    .....
} video_channel_desc;

```

The structure is described as follows:

- channel is the channel number;
- description is a format description string;
- Xsize_max and ysize_max: respectively represent the maximum width and maximum height (in pixel) of the output image supported by the channel;
- Queue_max: the maximum number of supported memory sequences. The sequence here refers to a sequence of image frames;
- formats: the number of supported video formats;
- capabilities: the function of channel. Currently, it is only defined that the channel is allowable to sample only one frame of data at a time.

13.10.3 Description of Image Format of the Device Channel

```

typedef struct video_format_desc {
    UINT32 channel; /* Specified video capture
channel */
    UINT32 index; /* Specified sequence number
*/

    CHAR description[32]; /* Instructions */
    UINT32 format; /* Frame format
video_pixel_format*/
    UINT32 order; /* MSB or LSB video_order_t*/

    UINT32 reserve[8];
} video_format_desc;

```

As mentioned above, the number of video formats supported by the channel is given in the channel descriptor through the format member and video_format_desc describes a specific video format.

- channel is the channel number corresponding to the format. The user sets this value to obtain the supported format of the specified channel.
- index is relative to formats and its value should be 0~formats, indicating that the number of type of the obtained format description information.
- description is the description string of the format;
- order indicates that a pixel data is stored at the big end or the small end, and its value is of video_order_t type.

```
typedef enum {
    VIDEO_LSB_CRCB = 0,                /*Low   front   (LSB)
    */
    VIDEO_MSB_CRCB = 1                /* High front (MSB)
    */
} video_order_t;
```

format is a specific video format flag whose value is the video_pixel_format enumeration type and is defined as follows:

```
typedef enum {
    VIDEO_PIXEL_FORMAT_RESERVE        = 0,
    /*
    * RGB
    */
    VIDEO_PIXEL_FORMAT_RGBA_8888     = 1,
    VIDEO_PIXEL_FORMAT_RGBX_8888     = 2,
    VIDEO_PIXEL_FORMAT_RGB_888       = 3,
    VIDEO_PIXEL_FORMAT_RGB_565       = 4,
    VIDEO_PIXEL_FORMAT_BGRA_8888     = 5,
    VIDEO_PIXEL_FORMAT_RGBA_5551     = 6,
    VIDEO_PIXEL_FORMAT_RGBA_4444     = 7,
    /*
    * 0x8 ~ 0xF range reserve
    */
    VIDEO_PIXEL_FORMAT_YCbCr_422_SP = 0x10,        /* NV16          */
    VIDEO_PIXEL_FORMAT_YCrCb_420_SP = 0x11,        /* NV21          */
    VIDEO_PIXEL_FORMAT_YCbCr_422_P  = 0x12,        /* IYUV          */
    VIDEO_PIXEL_FORMAT_YCbCr_420_P  = 0x13,        /* YUV9          */
    VIDEO_PIXEL_FORMAT_YCbCr_422_I  = 0x14,        /* YUY2          */
    /*
    * 0x15 reserve
    */
    VIDEO_PIXEL_FORMAT_CbYCrY_422_I = 0x16,
    /*
    * 0x17 0x18 ~ 0x1F range reserve
    */
}
```

```

    */
    VIDEO_PIXEL_FORMAT_YCbCr_420_SP_TILED = 0x20,    /* NV12 tiled    */
    VIDEO_PIXEL_FORMAT_YCbCr_420_SP      = 0x21,    /* NV12         */
    VIDEO_PIXEL_FORMAT_YCrCb_420_SP_TILED = 0x22,    /* NV21 tiled    */
    VIDEO_PIXEL_FORMAT_YCrCb_422_SP      = 0x23,    /* NV61         */
    VIDEO_PIXEL_FORMAT_YCrCb_422_P       = 0x24     /* YV12         */
} video_pixel_format;

```

Currently, SylixOS only gives the definitions for YUV and RGB image formats, and does not define other compression formats such as JPEG, BMP, etc., namely, it mainly processes video capture-related devices. From the above it can be seen that there are many video formats, no more information about YUV and RGB will be given in this section, and readers can understand it by other means.

13.10.4 Setting of Device Channel

After obtaining the information about device descriptors, channel descriptors, and format descriptor for each channel, the user can set related parameters of the channel according to actual needs. The structure of the channel control is defined as follows:

```

typedef struct video_channel_ctl {
    UINT32  channel;           /* Video channel number    */

    UINT32  xsize;           /* Acquisition output size */
    UINT32  ysize;

    UINT32  x_off;          /* Relative acquisition start
offset */
    UINT32  y_off;

    UINT32  x_cut;          /* Relative acquisition end offset
*/
    UINT32  y_cut;

    UINT32  queue;          /* Acquisition sequence number
*/

    UINT32  source;         /* Specified video input source
*/
    UINT32  format;         /* video_pixel_format */
    UINT32  order;         /* MSB or LSB video_order_t */

    .....
} video_channel_ctl;

```

The structure is described as follows:

- Xsize and ysize are used to specify the size of the final output image;
- X_off, y_off, x_cut, and y_cut refers to trimming processes of image frame. This means that we can extract only part of the original image frame and scale the partial area through xsize and ysize. Of course, these features require the support of a specific video device driver. If the video device does not support trimming and scaling, the application can only obtain the original video image data;
- queue indicates how many frames of data the video device needs to acquire. This value cannot exceed the maximum number of frames supported by the video device (queue_max defined by the channel descriptor described above);
- source specifies the input source of this channel, not exceeding the maximum number of video sources supported by the device;
- format and order should be one of the video format descriptors supported by the channel.

13.10.5 Setting of Device Buffer

Video acquisition is a continuous process, and the acquisition process continues while the upper layer is processing and acquiring a frame of data. This requires a buffer that can hold several frames of video data. The user always handles the buffering of valid frame data. The video device always puts the acquired data into the idle frame buffer, which avoids the conflict of the two. In fact, the vast majority of video capture devices support the setting of multi-frame buffering on the hardware. That is, each video output channel has one or more frame buffer queues. The queue is also called the ping-pong buffer. The capture card cyclically places the video data in the ping-pong buffer. For example, if a channel has 4 queue buffers, the video capture card will repeat this process from frame 1 to frame 4 at a time. Among the device channel descriptors mentioned above, queue_max represents the maximum number of valid frames of data that a video channel can buffer. Unlike other systems, the frame buffer defined by SylixOS is a piece of memory with continuous physical addresses that can hold several frames of data. This memory can be allocated for application by the application or automatically allocated by the driver.

In addition to the parameter of the number of frames in the buffer, there are other parameters that the user should attention. We talked about using the channel control structure video_channel_ctl to set the channel parameters, including the image size, cropping area, and format. In fact, after setting the channel parameters, the driver can get the buffer size of each frame of data and the total size of the buffer according to its own hardware. The user uses the buffer calculation request structure to obtain the detailed buffer parameters given by the driver:

```

typedef struct video_buf_cal {
    UINT32  channel;                /* Video channel number */

    size_t  align;                  /* Minimum memory
alignment requirement */
    size_t  size;                   /* The total size of the
channel buffer memory */
    size_t  size_per_fq;           /* Memory size per frame in
the queue */
    size_t  size_per_line;         /* One line of memory in
each image */

    .....
} video_buf_cal;

```

align is the memory alignment value of the entire buffer. In most cases, the video device internally uses DMA to transfer video data and has the address alignment requirements for the memory used.

size is the total size of the memory. Size_per_fq is the size of a frame of data, size_per_line is the size of each line of memory in a frame of image. In general, the size of a frame of data is equal to the product of the number of bytes per pixel and the length and width of the frame. However, different types of hardware may have data alignment requirements for each row of the frame data, so there may be cases where the end of row is padded with invalid data to satisfy the alignment condition. The user can know how to deal with the frame data according to the actual size of the image, combining with size_per_fq and size_per_line.

After obtaining the specific parameters of the buffer, set the specific buffer through the buffer control structure:

```

typedef struct video_buf_ctl {
    UINT32  channel;                /* Video channel number */

    PVOID   mem;                   /* Frame buffer (physical
memory address) */
    size_t  size;                   /* Buffer size */
    UINT32  mtype;                 /* Frame buffer type
video_mem_t */

    .....;
} video_buf_ctl;

```

As mentioned earlier, the memory of the frame buffer can be allocated by the user and provided to the driver. mem points to the memory allocated by the user. Note that the parameter requirements specified by video_buf_cal, such as the total memory size and the alignment value, must be satisfied. size indicates the size of the actual memory, which

is not less than the size specified by `videoo_buf_cal`. `mtype` indicates the type of the frame buffer and its value is of enumeration type, i.e. `video_mem_t`, defined as follows:

```
typedef enum {
    VIDEO_MEMORY_AUTO = 0,          /* Automatically allocate
frame buffer          */
    VIDEO_MEMORY_USER = 1         /* User allocation frame
buffer              */
} video_mem_t;
```

If the user allocates a buffer himself/herself, he/she needs to set `mtype` as `VIDEO_MEMORY_USER`, otherwise as `VIDEO_MEMORY_AUTO`, and set `mem` as `LW_NULL`. That the user allocates physical memory by himself/herself, which can result in better performance and efficiency to the application sometimes. If the driver allocates memory on its own, the application can only access the memory through the `mmap` memory map, which consumes the page space of the virtual memory. If there is a hardware in the system that needs to reprocess the image data, it also uses DMA to access the image memory, then we can apply the memory allocated by the user to the hardware and video interfaces at the same time so that the two directly perform memory interactions to achieve zero copy.

13.10.6 Video Capture Control

After completing the above steps, we can start the video device to start video capture. The structure of controlling the video capture is as follows:

```
typedef struct video_cap_ctl {
    UINT32 channel;                /* Video channel number */
#define VIDEO_CAP_ALLCHANNEL    0xffffffff

    UINT32 on;                    /* on / off          */
    UINT32 flags;

#define VIDEO_CAP_ONESHOT      1    /* Only one frame is
acquired          */

    .....;
} video_cap_ctl;
```

The structure is described as follows:

- `channel` specifies the channel to start the capture. If all the channels are set, then multiple channels can be started once via `VIDEO_CAP_ALLCHANNEL`. Of course, if only part of the channels are set, we need to start each channel separately;

- on indicates start or stop of the capture, 0 indicates stop and non-zero indicates start;
- Flags is a capture flag. Currently, only VIDEO_CAP_ONESHOT is defined, which corresponds to the capabilities of the channel descriptor.

After starting the video capture, in order to correctly process the captured data, we also need to obtain the current capture state using the following structure:

```
typedef struct video_cap_stat {
    UINT32 channel; /* 视频通道号 */
    UINT32 on; /* on / off */
    UINT32 qindex_vaild; /* The frame number of the
most recent valid frame */
    UINT32 qindex_cur; /* The queue number being
collected */
#define VIDEO_CAP_QINVAL 0xffffffff
    .....
} video_cap_stat;
```

The structure is described as follows:

- on indicates the current capture state, 0 is stopped, non-zero is started;
- qindex_valid indicates the queue number of the nearest frame, and the application should use this frame data.
- qindex_cur indicates the queue number being collected and the application should not use this frame data.

If the queue number is VIDEO_VAP_QINVAL, it indicates an invalid queue number, indicating that there is no valid frame data yet, and the application should continue to query the capture state.

As mentioned earlier, in SylixOS, all the frame buffers captured by the video are one continuous physical memory space, and the application can access the specified frame data according to the size of each frame of data and the frame index.

13.10.7 Summary of Operation Commands for the Video Device

Like other systems, the video device in SylixOS is a standard I/O device. All of the previously mentioned operations on the device are performed through ioctl system commands. Table 13.4 lists all the operating commands.

Table 13.4 Device operation command

Command Word	Parameter (for this type of pointer)	Description
VIDIOC_DEVDESC	video_dev_desc	Get the device descriptor

VIDIOC_CHANDESC	video_channel_desc	Get the specified channel descriptor of the device
VIDIOC_FORMATDESC	video_format_desc	Get the format descriptors supported by the channel
VIDIOC_GCHANCTL	video_channel_ctl	Get the parameters of the current channel
VIDIOC_SCHANCTL	video_channel_ctl	Set the parameters of the current channel
VIDIOC_MAPCAL	video_buf_cal	Get the parameters of the frame buffer
VIDIOC_MAPPREPAIR	video_buf_ctl	Set the frame buffer (pre-allocation of memory)
VIDIOC_CAPSTAT	video_cap_stat	Get the capture state
VIDIOC_GCAPCTL	video_cap_ctl	Get the current capture parameter
VIDIOC_SCAPCTL	video_cap_ctl	Set the current capture parameter

13.10.8 Video Device Application Examples

Next we will show an example of how to get specific information for a video device, as shown in Program List 13.12:

Program List 13.12 Getting Video Device Information

```
#include <SylixOS.h>
#include <video.h>

int main (int argc, char *argv[])
{
    int          fd;
    int          i, j;
    video_dev_desc dev;
    video_channel_desc channel;
    video_format_desc format;

    fd = open("/dev/video0", O_RDWR);
    ioctl(fd, VIDIOC_DEVDESC, &dev);
    for (i = 0; i < dev.channels; i++) {
        channel.channel = i;
        ioctl(fd, VIDIOC_CHANDESC, &channel);
        for (j = 0; j < channel.formats; j++) {
            format.channel = i;
            format.index = j;
            ioctl(fd, VIDIOC_FORMATDESC, &format);
        }
    }

    return 0;
}
```

In the above program, the device descriptor is first got, thus the number of video output channels supported by the device is obtained. Then, the channel descriptor of each channel is got, thus the number of video formats supported by each channel is obtained, and all the video formats supported by each channel of the video device is also obtained.

Let's suppose a specific application scenario below: we need to display the captured video in real time through the LCD. We need to understand the display parameters of the LCD device (that is, frame buffer FrameBuffer) and the format parameters of the video data, the data should be put it into FrameBuffer for display after appropriate software processing. Due to space limitations, the following program is implemented in pseudo-code and a lot of simplification is made.

Program List 13.13 Video Capture Example

```
#include <SylxOS.h>
#include <video.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

int main (int argc, char *argv[])
{
    int          fd;
    int          fb_fd;
    video_channel_ctl  channel;
    video_buf_cal    cal;
    video_buf_ctl    buf;
    video_cap_ctl    cap;
    video_cap_stat   sta;
    void          *pcapmem;
    void          *pfbmem;
    void          *pframe;
    fd_set        fdset;

    /*
     * Open the video device and get the necessary information    */
    fd = open("/dev/video0", O_RDWR);
    ...
    /*
     * Set the desired output image parameters
     * At the same time get the frame memory parameters
     */
    channel.channel = 0;
    channel.xsize   = 640;
    channel.ysize   = 480;
    channel.x_off   = 0;
    channel.y_off   = 0;
    channel.queue   = 1;

    channel.source  = 0;
    channel.format  = VIDEO_PIXEL_FORMAT_RGBX_8888;
    channel.order   = VIDEO_LSB_CRCB;
    ioctl(fd, VIDIOC_SCHANCTL, &channel);
    ioctl(fd, VIDIOC_MAPCAL, &cal);

    /*
     * Prepare memory data
     */
```

```

    buf.channel = 0;
    buf.mem     = NULL;
    buf.size    = cal.size;
    buf.mtype   = VIDEO_MEMORY_AUTO;
    ioctl(fd, VIDIOC_MAPPREPAIR, &buf);

    /*
     * Map frame memory
     */
    pcapmem = mmap(NULL, buf.size, PROT_READ, MAP_SHARED, fd, 0);

    cap.channel = 0;
    cap.on      = 1;
    cap.flags   = 0;
    ioctl(fd, VIDIOC_SCAPCTL, &cap);

    /*
     * Open FrameBuffer device
     * Map FrameBuffer Memory
     */
    fb_fd = open(...);
    pfbmem = mmap(..., fb_fd, 0);

    for (;;) {
        FD_ZERO(&fdset);
        FD_SET(fd, &fdset);

        /*
         * Wait for the device to read.
         * Every time the valid frame data is completed, the driver will wake up
the thread blocked here.
         */
        select(fd + 1, &fdset, NULL, NULL, NULL);
        if (FD_ISSET(fd, &fdset)) {
            ioctl(fd, VIDIOC_CAPSTAT, &sta);
            pframe = (char *)pcapmem + cal.size_per_fq * sta.qindex_vaild;
            ...
            memcpy(pfbmem, pframe, cal.size_per_fq);
        }
    }

    munmap(pcapmem, buf.size);
    close(fd);

```

```
munmap(pfbmem, ...);  
close(fb_fd);  
  
return (0);  
}
```

In the above example, we assume that the format parameter of the display device are known as RGB32 format, and the width and height are 640 and 480 pixels, respectively, and the same video output format is set. After starting the video capture, we wait for each frame of data through the call of the “select” system. The current valid frame index is obtained through the VIDIOC_CAPSTAT command. Note that a lot of simplification processing is made for the cases of obtaining the current frame data to be processed, the padding problem of invalid data is not considered, but it is assumed that all the data are valid pixel data. We get the current frame buffer address by the size of each frame and the current valid frame index, and then copy it directly to the display buffer. It is assumed here that the format of the captured video data is exactly the same as that of the display buffer format, otherwise, appropriate conversion processing should be performed before copying.

In fact, the operation of video device is relatively more cumbersome than other devices because it involves more parameter control. It is not simple to implement a video application that can adapt to a wide variety of hardware and software platforms. For example, if the video format does not support the RGB format, we also need to convert it to RGB format to achieve correct display. This example only serves as a guide and readers can gain insights through other means.

13.11 Real-Time Clock Device

A real-time clock device, i.e. RTC device, is an external device that is independent of the CPU clock and is usually powered by a separate power supply. Therefore, it is possible to continue processing time counts after the system is powered off. So we can believe that the RTC time indicates the real physical time. In SylixOS, the RTC device is a standard I/O device. Although the application can directly operate the device using standard I/O functions, this method is not recommended because an RTC device may be integrated inside the processor chip and may also be an external clock counting device, so its device name is not unique, absolutely depending on the driver. Therefore, in order to have a better portability of the program, it is recommended to use the standard API provided by SylixOS to operate RTC time (see Section 11.1.2 RTC time).

13.12 GPIO Devices

GPIO is namely a general-purpose input/output port, hereinafter referred to as an I/O port. An I/O port can provide input, output, or interrupt function. The GPIO device in SylixOS manages all available GPIO ports on the entire hardware system, allowing the

application to use the GPIO's three functions through the standard interface. The relevant definitions for the GPIO devices are located in the <sys/gpiofd.h> file. The relevant APIs are described below.

```
#include <sys/gpiofd.h>
int gpiofd(unsigned int gpio, int flags, int gpio_flags);
int gpiofd_read(int fd, uint8_t *value);
int gpiofd_write(int fd, uint8_t value);
```

Function gpiofd prototype analysis:

- The file descriptor corresponding to the GPIO port will be returned when the function succeeds, and a negative number will be returned when it fails;
- The parameter **gpio** is the unique number of the GPIO port. This number is associated to the specific system hardware. The application program should be selected correctly according to the definition of the GPIO port number in the BSP package.
- The parameter **flags** has the similar meaning to the second parameter of the **open** function, which can be O_RDONLY, O_RDWR, etc.
- The parameter **gpio_flags** is an identifier associated to the GPIO characteristic, which can be a combination of multiple bit identifiers. Refer to the following table:

Figure 13.1 GPIO function signature

Bit Identifier	Description
GPIO_FLAG_DIR_OUT	Set GPIO as the output function
GPIO_FLAG_DIR_IN	Set GPIO as the input function
GPIO_FLAG_IN	The same as GPIO_FLAG_DIR_IN
GPIO_FLAG_OUT_INIT_LOW	Set GPIO as the output function and initialize the output low level at the same time
GPIO_FLAG_OUT_INIT_HIGH	Set GPIO as the output function and initialize the output high level at the same time
GPIO_FLAG_OPEN_DRAIN	Set GPIO output as the drain output mode
GPIO_FLAG_OPEN_SOURCE	Set GPIO output as the source output mode
GPIO_FLAG_PULL_DEFAULT	Use the default pull-up/pull-down mode

GPIO_FLAG_PULL_UP	Use pull-up resistor mode
GPIO_FLAG_PULL_DOWN	Use pull-down resistor mode
GPIO_FLAG_PULL_DISABLE	Disable pull-up/pull-down mode
GPIO_FLAG_TRIG_FALL	Set GPIO as the interrupt function and the falling edge trigger interrupted
GPIO_FLAG_TRIG_RISE	Set GPIO as the interrupting function and the rising edge trigger interrupted
GPIO_FLAG_TRIG_LEVEL	Set GPIO as the interrupting function and the level trigger interrupted

Note: When using the GPIO_FLAG_TRIG_LEVEL flag, we can only use one of GPIO_FLAG_TRIG_FALL and GPIO_FLAG_TRIG_RISE to combine with it, indicating low level trigger and high level trigger respectively. When GPIO_FLAG_TRIG_LEVEL is not used, we can use GPIO_FLAG_TRIG_FALL in combination with GPIO_FLAG_TRIG_RISE to indicate double-edge triggering.

The function `gpiofd_read` reads the level state of a GPIO port with only two values of 0 and 1. The prototype analysis is as follows:

- This function returns 0 when it succeed and returns the error code when it fails.
- The parameter ***fd*** is the file descriptor corresponding to the GPIO port;
- The output parameter ***value*** holds the read level value, 0 means low level, 1 means high level.

The function `gpiofd_write` sets the level state of a GPIO port. There are only two values of 0 and 1. The prototype analysis is as follows:

- This function returns 0 when it succeed and returns the error code when it fails.
- The parameter ***fd*** is the file descriptor corresponding to the GPIO port;
- The parameter ***value*** is the level value to be set, 0 means low level, and 1 means high level.

The above function can only process the input and output of GPIO, but it cannot use its interrupt function. The way to use the interrupt function of the GPIO in the driver is to call the system kernel API to register the corresponding interrupt service program, but the application cannot call these functions. We know that I/O multiplexing (select) allows a

task to wait for one or more file descriptors in a block way to satisfy a specified state (read, write, or exception). SylixOS uses this to provide the application with the method to use the GPIO interrupt function. When a GPIO with an interrupt function generates an interrupt, the kernel wakes up all threads that call “select” to wait for the readable state of the GPIO file descriptor, to notify the thread of the generation of an interrupt. When “select” returns correctly, the thread will perform the corresponding processing, which is similar to completing an interrupt service. The following program shows the general use of GPIO devices.

Program List 13.14 Examples of GPIO Device Operations

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/select.h>
#include <sys/gpiofd.h>

#define LED_GPIO_NUM    52
#define KEY_GPIO_NUM    32

int main (int argc, char *argv[])
{
    fd_set  fdset;
    int     led_fd;
    int     key_fd;
    int     ret;
    uint8_t value = 1;

    led_fd = gpiofd(LED_GPIO_NUM, O_WRONLY, GPIO_FLAG_OUT_INIT_LOW);
    if (led_fd < 0) {
        fprintf(stderr, "open led gpiofd failed.\n");
        return (-1);
    }
    key_fd = gpiofd(KEY_GPIO_NUM, O_RDONLY, GPIO_FLAG_TRIG_FALL);
    if (key_fd < 0) {
        fprintf(stderr, "open key gpiofd failed.\n");
        return (-1);
    }

    while (1) {
        FD_ZERO(&fdset);
        FD_SET(key_fd, &fdset);

/*
* 等待按键中断产生
*/
```

```
        ret = select(key_fd + 1, &fdset, NULL, NULL, NULL);
    if (ret != 1) {
        fprintf(stderr, "select error.\n");
        break;
    }

    gpiofd_write(led_fd, value);
    value = !value;
}

close(led_fd);
close(key_fd);

return (0);
}
```

In the above program, we assume that the port number of the control LED is 52, the port number of the interrupt detection key is 32, and it is assumed that a falling edge interrupt will be generated when the key is pressed. We set `fdset` as the read wait file descriptor set for the `select` function, that is, waiting for the GPIO file to be readable, actually waiting for a key interrupt to occur. The effect of the above program is that the LED light's state changes every time the button is pressed (from being lighted to being extinguished or from being extinguished to being lighted).

13.13 CAN Bus Device

CAN is namely the Controller Area Network. and it is a serial communication protocol, and widely used in automotive electronics, automatic control, and security monitoring. The CAN bus protocol only defines the physical layer and data link layer in the OSI model. In practical applications, communication is usually performed in a high-layer protocol based on the CAN basic protocol. In the CAN,basic protocol, the frame is used as the basic transmission unit. Similar to the MAC frame in Ethernet, the CAN controller is responsible for performing level conversion, message verification, error handling, and bus arbitration for the CAN frame. However, there is no source address and destination address in the CAN frame. This indicates that in a CAN bus system, the CAN controllers of all nodes cannot perform hardware address filtering like Ethernet to achieve directional communication, and only the application layer determines whether to receive the frame. The CAN high-layer protocol deals with similar tasks. It carefully defines various fields of the CAN frame, giving it special meaning, and handles some of the work that the underlying protocol does not handle, such as device type definition, device status monitoring and management on the bus. We refer to CAN high-layer protocols collectively as CAN application layer protocols.

Currently, CAN application layer protocols include DeviceNet, CANopen, CAL, etc. They have their own protocol standards for different applications. Due to space limitations,

this document does not introduce the specific knowledge about the CAN underlying protocol and application protocol.

The CAN bus device in SylixOS only supports the underlying protocol. The device is a character device, but its read and write operations must use a CAN frame as the basic unit. The definition of the CAN frame in SylixOS is located in <SylixOS/system/device/can.h> and the structure is as follows:

```
#define CAN_MAX_DATA      8                /* CAN The maximum length of the
frame data                */
typedef struct {
    UINT      CAN_uid;                    /* Identification code */
    UINT      CAN_uiChannel;              /* Channel number */
    BOOL      CAN_bExtId;                 /* Whether it is an extended
frame                        */
    BOOL      CAN_bRtr;                   /* Whether it is a remote frame
*/
    UCHAR     CAN_ucLen;                  /* Data length */
    UCHAR     CAN_ucData[CAN_MAX_DATA];  /* Frame data */
} CAN_FRAME;
typedef CAN_FRAME *PCAN_FRAME;
```

The member `CAN_uid` is a CAN node identifier. In a CAN bus system, the identifier of each node is unique. If the member `CAN_bExtId` is `FALSE`, it indicates a standard frame, the lower 11 bits of `CAN_uid` are valid, otherwise it is expressed as an extended frame, then the lower 29 bits of `CAN_uid` are valid. Most application protocols will redefine `CAN_uid`, for example, using a part of bits to indicate the data type of the device, and some bits to indicate the address of the device. Therefore, the purpose of the extension frame is to meet more application data requirements and support more devices within the unified network on an existing basis. `CAN_uiChannel` is not specified by the CAN protocol. This data is used in SylixOS to indicate the hardware channel number of the CAN device in the system. It is usually not processed in actual applications. `CAN_bRtr` indicates whether it is a remote frame. The role of a remote frame is to allow the nodes that desire for the frame to request for the frames having the same remote frame identifier from the node in the CAN system. The maximum frame data length of a CAN frame is 8 bytes. The member `CAN_ucLen` denotes the actual length of the data in the current frame, and `CAN_ucData` denotes the actual data.

Now we consider such an application scenario: In a CAN bus system, there are many nodes responsible for data acquisition. The data types they acquired are different, and the corresponding data formats are also different. Of course, there may be multiple nodes for the acquisitions of the same data type. In the system, there is also a node responsible for collecting and processing these data. The basic requirement on it is to correctly identify different data formats and perform corresponding parsing processing. In order to effectively distinguish the different data types, we can artificially define the data in the CAN frame. For example, we can use part of the data bits of `CAN_uid` to indicate the data

type, and the rest indicates the node ID, but this reduce the number of CAN nodes that can be supported by the entire CAN system; another method is to use part of the CAN_ucData data bits (such as the first byte) to indicate the data type, but this will reduce the amount of data that can be transmitted in a single time. In order to describe these behaviors, we need to define a common operating standard. This is equivalent to that we define a CAN application layer protocol ourselves. Here we will simply call the custom protocol APP and define it in appLib.h.

Program List 13.15 CAN Custom Application Protocol

```
#ifndef __APP_LIB_H
#define __APP_LIB_H

#define APP_TYPE_MASTER      0
#define APP_TYPE_INT32      1
#define APP_TYPE_STRING     2
#define APP_ADDR_MASTER     0

#define APP_TYPE(id)        ((id >> 7) & 0x0f)
#define APP_ADDR(id)        (id & 0x3f)
#define APP_NET_ID(t, a)    (((UINT)t & 0x0f) << 7) | ((UINT)a & 0x3f)

static inline INT32 __appByteToInt32 (const UCHAR *pucByte)
{
    INT32 iData;

    iData = ((INT32)pucByte[0])
        | ((INT32)pucByte[1] << 8)
        | ((INT32)pucByte[2] << 16)
        | ((INT32)pucByte[3] << 24);

    return (iData);
}

static inline VOID __appInt32ToByte (UCHAR *pucByte, INT32 iData)
{
    pucByte[0] = iData & 0xff;
    pucByte[1] = (iData >> 8) & 0xff;
    pucByte[2] = (iData >> 16) & 0xff;
    pucByte[3] = (iData >> 24) & 0xff;
}
```

```
extern UINT __appSlaveAddrGet(VOID);

#endif /* __APP_LIB_H */
```

As shown in Program List 13.15, we have redefined the ID of the CAN underlying protocol. First of all, only standard frames exist in the system. This means that the effective data bits of the ID are 11 bits. We use the upper 4 bits to indicate the data type and the lower 7 bits to indicate the CAN device address. For simplicity, we define two data types, one indicates the data is the 32-bit signed integer and the other indicates that the data is a string. In addition, we refer to the node responsible for collecting data in the system as the master and refer to the node responsible for information acquisition as the slave. The macro definition **APP_ADDR_MASTER** reserves a device address for the master mode. This shows that there can be only one master node in the entire system.

The inline functions `__appByteToInt32` and `__appInt32ToByte` process the data with the type of integers, the former is used for the integers that are parsed to be the actual application after the master node has received the byte-form data, and the latter is used for the data that has been processed to be byte form by the slave node prior to the transmission of the integer data, and the data will be send to the network.

`__appSlaveAddrGet` is used to get a unique slave node address. Just like the IP address and MAC address in Ethernet, the unique identifier of the node in a network must be managed by a third party organization (IP address is managed by IANA, and MAC address is managed by IEEE). In our CAN bus system, although each node can be artificially guaranteed with the uniqueness of the address, the source or storage of the addresses may not be the same. It can come from non-volatile memory such as EEPROM, NANDFLASH, or SD card, and even we can provide a service node similar to DHCP in the entire system, allowing other nodes to dynamically obtain the address information. Because of so many possibilities, we let the method of address acquisition achieved in the specific applications, the above function is only declared as an external function instead of being implementation.

Program List 13.16 Example of Master Nodes for CAN Custom Protocols

```
#include <SylixOS.h>
#include "appLib.h"

#define CAN_DEV_NAME "/dev/can0"

int main(int argc, char *argv[])
{
    INT iCanFd;
```

```
CAN_FRAME    canframe;

UINT         uiType;
UINT         uiAddr;
ssize_t      sstReadLen;
UINT         uiNetId;

iCanFd = open(CAN_DEV_NAME, O_RDONLY);
if (iCanFd < 0) {
    fprintf(stderr, "open %s failed.\n", CAN_DEV_NAME);
    return (-1);
}

while (1) {
    sstReadLen = read(iCanFd, &canframe, sizeof(CAN_FRAME));
    if (sstReadLen < 0) {
        fprintf(stderr, "read error.\n");
        break;
    }
    if (sstReadLen < sizeof(CAN_FRAME)) {
        continue;
    }

    uiNetId = canframe.CAN_uiId;
    uiAddr = APP_ADDR(uiNetId);
    uiType = APP_TYPE(uiNetId);
    switch (uiType) {
    case APP_TYPE_INT32:
    {
        INT32  iData;
        iData = __appByteToInt32(canframe.CAN_ucData);
        printf("node addr = %d, type = int32, value = %d.\n",
            uiAddr, iData);
    }
        break;

    case APP_TYPE_STRING:
    {
        CHAR *pcData = (CHAR *)canframe.CAN_ucData;
        pcData[canframe.CAN_ucLen] = '\0';
        printf("node addr = %d, type = string, value = %s.\n",
            uiAddr, pcData);
    }
        break;
    }
}
```

```

        default:
            break;
    }
}

close(iCanFd);
return (0);
}

```

The above is the master node program, and its function is very simple, namely continuously obtaining the data from the slave node in the network, and printing it out after corresponding processing according to the data type. Be sure to read the data with a CAN frame as the basic size.

Program List 13.17 Sample Slave Nodes for CAN Custom Protocols

```

#include <SylixOS.h>
#include "appLib.h"

#define CAN_DEV_NAME      "/dev/can0"

int main(int argc, char *argv[])
{
    INT          iCanFd;
    CAN_FRAME    canframe;
    UINT         uiAddr;
    ssize_t      sstWriteLen;
    INT32        iData = 0;

    iCanFd = open(CAN_DEV_NAME, O_WRONLY);
    if (iCanFd < 0) {
        fprintf(stderr, "open %s failed.\n", CAN_DEV_NAME);
        return (-1);
    }

    uiAddr = __appSlaveAddrGet();
    canframe.CAN_uiId      = APP_NET_ID(APP_TYPE_INT32, uiAddr);
    canframe.CAN_ucLen     = sizeof(INT32);
    canframe.CAN_bExtId    = LW_FALSE;
    canframe.CAN_bRtr      = LW_FALSE;
    canframe.CAN_uiChannel = 0;
    while (1) {
        __appInt32ToByte(canframe.CAN_ucData, iData);
        sstWriteLen = write(iCanFd, &canframe, sizeof(CAN_FRAME));
        if (sstWriteLen < 0) {
            fprintf(stderr, "write error.\n");

```

```
        break;
    }

    iData++;
    sleep(5);
}

close(iCanFd);
return (0);
}
```

The above slave program reports the data of integer type to the system every 5 seconds. All that needs to be done is to set the data type field in the ID to APP_TYPE_INT32. In our example program, we did not use the extension frame and remote frame identification in the CAN frame, nor did we deal with the details of more communication, such as the effective recovery, initiation and response to communication, processing of big and small endian data between different nodes when the abnormal communication occurs on a certain node. As mentioned earlier, the existing CAN application layer protocols can handle the problems of ensuring the stability and validity of communications, and provide convenient operating interfaces for applications.

13.14 Virtual Device Files

The Linux kernel has been gradually added with three virtual device files since version 2.6.22, namely, eventfd, timerfd and signalfd. These three files allow applications to use events (semaphores), timers, and signal resources through standard I/O operations instead of the traditional method to call API. The biggest advantage due to this is that applications can use “select (or poll and epoll)” while monitoring multiple such files simultaneously (or such files and other files), so that the asynchronous parallel processing of multiple events can be converted into synchronous serial processing, which is very useful in many applications. SylixOS is fully compatible with these three virtual device files and added with a hstimerfd used for high-precision timers. The following describes how to use them.

13.14.1 eventfd

eventfd is mainly used for event notification between threads. Linux supports the call of the fork system, so it can also be used for event notification between parent and child processes. The relevant API is described as follows:

```
int eventfd(unsigned int initval, int flags);
int eventfd_read(int fd, eventfd_t *value);
int eventfd_write(int fd, eventfd_t value);
```

The prototype analysis of the function eventfd:

- The function returns a file descriptor if it succeeds, and returns a negative number and set the error number when it fails;
- The parameter *initval* indicates the initial state of the event. For example, 0 means no event occurs currently.
- The parameter *flags* is the operation option for this event file and can be EFD_CLOEXEC, EFD_NONBLOCK, and EFD_SEMAPHORE. The specific meaning of EFD_SEMAPHORE will be explained later.

The function `eventfd_read` is used to read the event file, namely, waiting for the sending of event. When there is no event and the `EFD_NONBLOCK` parameter is not used, the function will be blocked. Its prototype analysis is as follows:

- The function returns 0 when it succeeds, and returns the error code when it fails;
- The parameter *fd* is an event file descriptor opened by `eventfd`;
- The output parameter *value* holds the number of the read events, and its behavior is related to the `EFD_SEMAPHORE` flag. `eventfd_t` is defined as an unsigned 64-bit integer in most systems. In normal circumstances, the number of events it can indicate can be interpreted as infinite.

When the parameter *flag* using the `eventfd` function contains `EFD_SEMAPHORE`, it means that the event is operated by the counting semaphore. That is, if multiple events have been generated, only one event will be obtained for each read. The internal event counter of the file only performs the calculation by decrement of one, and the *value* in the file is 1. When `EFD_SEMAPHORE` is not used, all events are read at a time. The contents of *value* are the number of read events, and the internal event counter will be zeroed. The two ways can meet the needs of the application in different scenarios.

The function `eventfd_write` is used to write event files, i.e. sending events. When the internal event counter reaches the maximum value, no events can be further sent, and the prototype analysis is as follows:

- The function returns 0 when it succeeds, and returns the error code when it fails;
- The parameter *fd* is an event file descriptor opened by `eventfd`;
- The input parameter *value* is the number of the sent events, and its behavior is related to the `EFD_SEMAPHORE` flag.

The following example shows the use of `eventfd`.

Program List 13.18 Achieving synchronization between threads using the event file

```
#include <stdio.h>
#include <sys/eventfd.h>
#include <pthread.h>

void *event_write_routine (void *arg)
```

```
{
    eventfd_t    value    = 1;
    int          event_fd = (int)arg;
    int          ret;

    while (1) {
        ret = eventfd_write(event_fd, value);
        if (ret) {
            fprintf(stderr, "write eventfd error.\n");
            break;
        }
        value++;
    }
    return (NULL);
}

int event_write_server_start (int event_fd, void *(*routine)(void *))
{
    pthread_t    tid;
    int          ret;

    ret = pthread_create(&tid, NULL, routine, (void *)event_fd);
    if (ret != 0) {
        fprintf(stderr, "pthread create failed.\n");
        return (-1);
    }

    return (0);
}

int main (int argc, char *argv[])
{
    int          event_fd;
    int          ret;
    eventfd_t    value;

    event_fd = eventfd(0, 0);
    if (event_fd < 0) {
        fprintf(stderr, "open eventfd failed.\n");
        return (-1);
    }

    ret = event_write_server_start(event_fd, event_write_routine);
    if (ret) {
```

```
        fprintf(stderr, "start eventfd write server failed.\n");
    close(event_fd);
    return (-1);
}

while (1) {
    ret = eventfd_read(event_fd, &value);
    if (ret) {
        fprintf(stderr, "read eventfd error.\n");
        break;
    }
    fprintf(stdout, "read event value count: %llu.\n", value);
}

close(event_fd);
return (0);
}
```

In the above program, the child thread continuously writes an incremented event to the event file. The main thread continuously reads the event and prints the value of this event. Its operating results are as follows:

```
# ./eventfd_test
read event value count: 1.
read event value count: 2.
read event value count: 3.
read event value count: 4.
read event value count: 5.
read event value count: 6.
read event value count: 7.
read event value count: 8.
```

The above result is very regular. It can be seen that the event value of each read reflects only the value of the event written once, rather than the sum of multiple write values. In fact, the `eventfd_write` function will always wait for the last write event to be read to complete the write of this event. Therefore, the read and write of the event file is a synchronized process. The value of an event can be understood as the number of resources that can be processed after an event occurs, `EFD_SEMAPHORE` allows the event receiver to decide whether to handle multiple resources at a time or a single resource.

13.14.2 timerfd

In Chapter 11 Time Management, we have described the use of SylixOS timer-related API functions. The timer function can also be achieved through the use of Timerfd. The relevant APIs are as follows.

```
#include <timerfd.h>
int timerfd_create(clockid_t clockid, int flags);
int timerfd_settime(int fd, int flags, const struct itimerspec *ntmr,
struct itimerspec *otmr);
int timerfd_gettime(int fd, struct itimerspec *currvalue);
```

The prototype analysis of the function timerfd_create:

- The function returns a timer descriptor if it succeeds, and returns a negative number and set the error number when it fails;
- The parameter **clockid** indicates the type of clock source referenced by the timer, which may be CLOCK_REALTIME or CLOCK_MONOTONIC, representing real time and linear increment time, respectively;
- The parameter **flags** is the option bit identifier of the timer file, which can be TFD_CLOEXEC (equal to O_CLOEXEC) and TFD_NONBLOCK (equal to O_NONBLOCK);

The function timerfd_settime is used to set the timer start time and reload interval. The prototype analysis is as follows:

- The function returns 0 if it succeeds, and returns a negative number and set the error number when it fails;
- The parameter **fd** is a timer file descriptor;
- The parameter **flags** is an identifier related to timing, which may be 0 or TIMER_ABSTIME, and affects the meaning of the parameter **ntmr**;
- The parameter **ntmr** describes the time parameters of the timer: start time and reload time;
- The output parameter **otmr** holds the old time parameter, which can be NULL.

The data type itimerspec is defined as follows:

```
struct itimerspec {
    struct timespec it_interval;           /* Timer reload value */
    struct timespec it_value;             /* The remaining time until the
next expiration */
};
```

it_interval is the time interval for the timer period trigger, and it_value indicates the time when the timer is first triggered. The meaning of this value is defined by **flags**. If the TIMER_ABSTIME flag is set, it indicates that it_value is an absolute time. When the system time reaches this value, the timer is triggered. If **flags** is 0, it indicates it_value is a

relative time, after which, the timer will be triggered. The types of the two are both `timespec`, which means that the timer can achieve nanosecond time accuracy.

The function `timerfd_gettime` obtains the current time parameter of the timer file. The prototype analysis is as follows:

- The function returns 0 if it succeeds, and returns a negative number and set the error number when it fails;
- The parameter ***fd*** is a timer file descriptor;
- The output parameter ***currvalue*** holds the current time parameter.

Once the `timerfd_settime` is called successfully, the timer will be started and the timer will be triggered when the set time condition is met. The application can use “read” or “select” to wait for a timer trigger (same as GPIO device).

Program List 13.19 Example of timerfd Application

```
#include <sys/timerfd.h>
#include <stdio.h>
#include <stdint.h>
#include <time.h>

static int show_elapsed_time (void)
{
    static struct timespec start;
    struct timespec curr;
    static int first_call = 1;
    int secs;
    int nsecs;

    if (first_call) {
        first_call = 0;
        if (clock_gettime(CLOCK_MONOTONIC, &start) == -1) {
            return (-1);
        }
    }
    if (clock_gettime(CLOCK_MONOTONIC, &curr) == -1) {
        return (-1);
    }
    secs = curr.tv_sec - start.tv_sec;
    nsecs = curr.tv_nsec - start.tv_nsec;
    if (nsecs < 0) {
        secs--;
        nsecs += 1000000000;
    }
    fprintf(stdout, "time elapsed: %d.%03d seconds.\n",
```

```
secs, (nsecs + 500000) / 1000000);
return (0);
}

int main(int argc, char *argv[])
{
    struct itimerspec    time;
    int                  timer_fd;
    int                  ret;
    uint64_t             expired;
    ssize_t              read_len;

    timer_fd = timerfd_create(CLOCK_MONOTONIC, 0);
    if (timer_fd < 0) {
        fprintf(stderr, "create timerfd failed.\n");
        return (-1);
    }

    time.it_value.tv_sec      = 3;
    time.it_value.tv_nsec    = 0;
    time.it_interval.tv_sec  = 1;
    time.it_interval.tv_nsec = 0;
    ret = timerfd_settime(timer_fd, 0, &time, NULL);
    if (ret) {
        fprintf(stderr, "start timerfd error.\n");
        close(timer_fd);
        return (-1);
    }

    ret = show_elapsed_time();
    if (ret) {
        close(timer_fd);
        return (-1);
    }

    while (1) {
        read_len = read(timer_fd, &expired, sizeof(uint64_t));
        if (read_len < sizeof(uint64_t)) {
            fprintf(stderr, "read timerfd error.\n");
            break;
        }
        ret = show_elapsed_time();
        if (ret) {
            break;
        }
    }
}
```

```
    }
    fprintf(stdout, "timer is triggered, expire count = %llu.\n", expired);
    sleep(2);
}

close(timer_fd);
return (0);
}
```

In the above program, when the timer is created, the parameter **flags** is 0, which means the relative time is used. When the `timerfd_settime` is called, the setting of the time parameter indicates that the timer is triggered 3 seconds after it was started, and then will be triggered once every 1 second. . The private function `show_elapsed_time` is used to display the elapsed time since the timer is started, which is used as a criterion for verifying the accuracy of the timer. After the timer is triggered for the first time, the program periodically waits for the timer's triggering state every 2 seconds. Note that the read way is used to wait for the trigger of the timer. The data it reads is a 64-bit unsigned type. The data indicates how many times the timer is triggered up to this triggering. We also refer to it as a timer's number of expirations. After the program is run, the result is as follows:

```
# ./timerfd_test
time elapsed: 0.000 seconds.
time elapsed: 2.996 seconds.
timer is triggered, expire count = 1.
time elapsed: 4.996 seconds.
timer is triggered, expire count = 2.
time elapsed: 6.996 seconds.
timer is triggered, expire count = 2.
time elapsed: 8.996 seconds.
timer is triggered, expire count = 2.
time elapsed: 10.996 seconds.
timer is triggered, expire count = 2.
time elapsed: 12.996 seconds.
timer is triggered, expire count = 2.
```

From the results, it can be seen that 2.996 seconds (there is a certain deviation in the precision) has passed in the timer when the timer is triggered for the first time, and the timer expiration count is displayed as 1, which is in accordance with our setting. Thereafter, the timer is triggered once every 1 second, we have to get the timer expiration count of 2 every 2 seconds, which is also consistent with expectations.

It should be noted that when setting the time parameter of the timer, if the time value of `it_value` is 0, it does not mean that the timer is triggered immediately, but means to stop the timer; similarly, when the time value of `it_interval` is 0, it does not indicate the timer is triggered infinitely without waiting time, but indicates to stop the timer.

13.14.3 hstimerfd

SylixOS supports the timer with the time accuracy higher than the system clock, and provides related API functions (see Chapter 11 Time Management). The high-precision timer can only guarantee its time accuracy is not lower than the ordinary timer, and the accuracy depends entirely on the system hardware and the support of BSP package. SylixOS also provides hstimerfd files similar to tirmfd, allowing the application to use the high-precision timer via standard I/O. The relevant API is defined as follows:

```
#include <sys/hstimerfd.h>
int hstimerfd_hz(void);
int hstimerfd_create(int flags);
int hstimerfd_settime(int fd,
const struct itimerspec*ntmr,
struct itimerspec      *otmr);
int hstimerfd_settime2(int fd, hstimer_cnt_t *ncnt, hstimer_cnt_t *ocnt);
int hstimerfd_gettime(int fd, struct itimerspec *currvalue);
int hstimerfd_gettime2(int fd, hstimer_cnt_t *currvalue);
```

Hstimerfd_settime and hstimerfd_gettime are consistent with the the common timer API mentioned in the previous section in terms of parameters and behaviors. The function hstimerfd_hz returns the count frequency of the high-precision timer, which is also the timing accuracy that can be achieved.

The function hstimerfd_create prototype analysis is as follows:

- The function returns a file descriptor if it succeeds, and returns a negative number and set the error number when it fails;
- The parameter *flags* is the file identifier, which can be HSTFD_CLOEXEC (equal to O_CLOEXEC) and HSTFD_NONBLOCK (equal to O_NONBLOCK);

The function hstimerfd_settime2 prototype analysis is as follows:

- The function returns 0 when it succeeds, and returns the error code when it fails;
- The parameter *fd* is a timer file descriptor;
- The parameter *ncnt* is similar to hstimer_cnt_t, indicating a new timer time count parameter;
- The output parameter *ocnt* holds the old timer time count parameter.

Hstimerfd provides a new time parameter method that uses the count value of a high-precision timer as its time parameter. The data structure hstimer_cnt_t is defined as follows:

```
typedef struct hstimer_cnt {
    unsigned long    value;
    unsigned long    interval;
} hstimer_cnt_t;
```


The member variable value denotes the count value (the count value upon first-time expiration) that the timer is triggered for the first time since the timer was started. `interval` is the count value for the timer period trigger.

The function `hstimerfd_gettime2` prototype analysis is as follows:

- The function returns 0 when it succeeds, and returns the error code when it fails;
- The parameter ***fd*** is a timer file descriptor;
- The output parameter ***currvalue*** holds the timer's current time parameter.

A high-precision timer is similar to a common timer except that the time accuracy is different, so no further examples are given here. The reader can perform a slight modification according to the programs listed in Program List 13.19, that means, a high-precision timer can be used.

13.14.4 signalfd

The traditional signal processing method is to use the `signal` or `sigaction` functions to register the signal processing function concerned. When the signal occurs, these functions will be called in an asynchronous manner. Therefore, the data concurrency needs to be considered in use, and See Chapter 10 Signal System for its related API and its use methods. `signalfd` allows the application to wait for signals to be generated in the file form and process them synchronously. `signalfd` is a standard I/O device file whose definition is located in the `<sys/signalfd.h>` header file, as shown below:

```
#include <sys/signalfd.h>
int signalfd(int fd, const sigset_t *mask, int flags);
```

The function `signalfd` prototype analysis is as follows:

- The function returns a signal descriptor if it succeeds, and returns a negative number and set the error number when it fails;
- The parameter ***fd*** indicates an existing signal file descriptor. If ***fd*** is -1, it indicates a new signal file should be created; and if ***fd*** is an existing signal file descriptor, it means the signal to be processes should be reset;
- The parameter ***mask*** contains a set of signals that need attention;
- The parameter ***flags*** may be `SFD_CLOEXEC` (equal to `O_CLOEXEC`) and `SFD_NONBLOCK` (equal to `O_NONBLOCK`).

When the function `signalfd` is successfully called, the file descriptor returned by it is associated with the signal specified by the parameter ***mask***. Thereafter, the function `read` is used to wait for the signal to occur, the read data is a `signalfd_siginfo` structure, which is defined as follows:

```
struct signalfd_siginfo {
```

```

uint32_t    ssi_signo;        /* Signal number
*/
int32_t     ssi_errno;       /* Error number (unused)    */
int32_t     ssi_code;        /* Signal code              */
uint32_t    ssi_pid;         /* PID of sender            */
uint32_t    ssi_uid;         /* Real UID of sender       */
int32_t     ssi_fd;          /* File descriptor (SIGIO)  */
uint32_t    ssi_tid;         /* Kernel timer ID (POSIX timers) */
uint32_t    ssi_band;        /* Band event (SIGIO)       */
uint32_t    ssi_overrun;     /* POSIX timer overrun count */
uint32_t    ssi_trapno;      /* Trap number that caused signal */
int32_t     ssi_status;      /* Exit status or signal (SIGCHLD) */
int32_t     ssi_int;         /* Integer sent by sigqueue(3) */
uint64_t    ssi_ptr;         /* Pointer sent by sigqueue(3) */
uint64_t    ssi_utime;       /* User CPU time consumed (SIGCHLD) */
uint64_t    ssi_stime;       /* System CPU time consumed (SIGCHLD) */
uint64_t    ssi_addr;        /* Address that generated signal */
                                     /* (for hardware-generated signals) */
uint8_t     pad[48];
};

```

The member variable `ssi_signo` of `signalfd_siginfo` is the number of the current signal, and we can deal with different signals according to it. `signalfd_siginfo` has the same meaning as many members of the same name in `siginfo_t`. See `siginfo_t` and refer to Chapter 10 Signal System for details. Next, a simple example for how to use `signalfd` is given.

Program List 13.20 Signalfd Application Example

```

#include <sys/signalfd.h>
#include <stdio.h>
#include <stdint.h>
#include <signal.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    sigset_t          sig_mask;
    int               sig_fd;
    struct signalfd_siginfo sig_info;
    ssize_t           read_size;

    sigemptyset(&sig_mask);
    sigaddset(&sig_mask, SIGINT);
    sigaddset(&sig_mask, SIGQUIT);

```

```

    if (sigprocmask(SIG_BLOCK, &sig_mask, NULL) == -1) {
        fprintf(stderr, "sigprocmask error.\n");
        return (-1);
    }

    sig_fd = signalfd(-1, &sig_mask, 0);
    if (sig_fd < 0) {
        fprintf(stderr, "create signalfd error.\n");
        return (-1);
    }

    while (1) {
        read_size = read(sig_fd, &sig_info, sizeof(struct signalfd_siginfo));
        if (read_size != sizeof(struct signalfd_siginfo)) {
            fprintf(stderr, "read signalfd error.\n");
            break;
        }

        if (sig_info.ssi_signo == SIGINT) {
            printf("got SIGINT signal.\n");
        } else if (sig_info.ssi_signo == SIGQUIT) {
            printf("got SIGQUIT signal.\n");
            break;
        } else {
            fprintf(stderr, "got unexpected signal.\n");
        }
    }

    close(sig_fd);
    return (0);
}

```

In the above program, we added the two signals of interest SIGINT and SIGQUIT to the signal set, and then block them through sigprocmask. This is not the same as using signal or sigaction to process signals. They require that the signal of interest cannot be blocked. When signalfd is used, if the signals are not blocked, they will go to the default signal processing function when the signal occurs. When the signal occurs, although these signals are blocked, the system will make the files associated with these signals readable, so we use the read function to wait for the signal to occur. In this way, SIGINT and SIGQUIT are serialized in one thread. After the program is running, first check its result with the **ps** command as follows:

```

# ./sigfd_test &
# ps

```

NAME	FATHER	PID	GRP	MEMORY	UID	GID	USER
------	--------	-----	-----	--------	-----	-----	------

```
-----  
-----  
kernel          <orphan>      0    0    36864    0    0    root  
sigfd_test      <orphan>      16   16   45056    0    0    root
```

The name of the test program is `sigfd_test`, and its PID is 16. Now we use the `kill` command to send the two signals of interest to the process. The corresponding number of `SIGINT` is 2, and that of `SIGQUIT` is 3. The result is as follows:

```
# kill -n 2 16  
got SIGINT signal.  
# kill -n 2 16  
got SIGINT signal.  
# kill -n 2 16  
got SIGINT signal.  
# kill -n 3 16  
got SIGQUIT signal.
```

Under Linux, we can use the `Ctrl+C` key combination to send the `SIGINT` signal to the current process. Use `Ctrl+\` to send the `SIGQUIT` signal to the current process. In SylixOS, using `Ctrl+C` always always exits the current process without outputting the above information.



Chapter 14 Hot-plug System

14.1 Introduction of the Hot-plug System

The hot-plug system is used to manage and monitor the plug-in/removal state of all the hot-plug devices in the system, so that the system internally create/delete such devices automatically, without the need of the user's manual operation. At the same time, the hot-plug system also collects information related to hot-plug and supplies the same to the application.

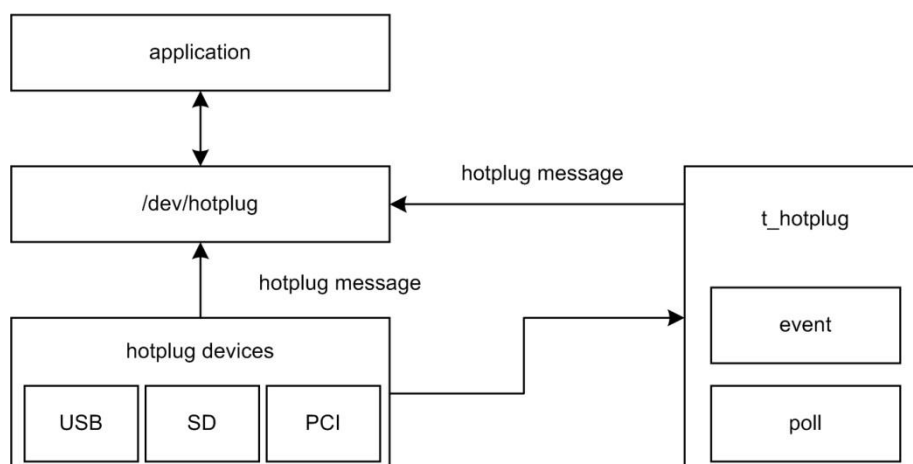


Figure 14.1 Overall Structure of a Hot-plug System

As shown in Figure 14.1, there is a kernel thread named "t_hotplug" in the SylixOS system. The hot-plug state of the device is reported to the thread through the event. The creation/deletion of the device is handled by this thread. Some devices cannot generate hot-plug events (such as the devices without plug-in interception detection pins). You can register hot-plug detection functions with the system, and the "t_hotplug" thread can periodically calls such registered functions to perform hot-plug detection, namely the polling detection (corresponding to the poll module in the above figure). The system also has a virtual device named "/dev/hotplug". It collects related hot-plug messages. Generally, hot-plug messages come from the "t_hotplug" thread, and may also from the device driver. The application can read the /dev/hotplug device and get the hot-plug messages it cares about.

SylixOS defines the current hot-plug device messages, such as USB, SD, and PCI. In addition, there are messages similar to the hot-plug behavior such as the connection and disconnection of the network and the change of the connection state of the power supply.

14.2 Hot-plug Message

14.2.1 Format of Hot-Plug Messages



Figure 14.2 Format of Hot-plug Message

As shown in the above figure, the first 4 bytes of the message indicates the type of the message, which is of the big-endian format. The currently-defined message type is located in the <SylixOS/system/hotplugLib/hotplugLib.h> header file, such as USB keyboard, USB mouse, SD memory card, SDIO wireless network card, and so on. On an actual hardware platform, a device driver can also define its own hot-plug message type.

The 5th byte is the device state, with 0 indicating the device is unplugged and 1 indicating the device is inserted.

Beginning from the 6th byte, it indicates the name of the device, the content of which is a string ending with '\0'. The application should get a full name with this as the terminator. The name is the full path name of a device, such as "/dev/ttyUSB0", "/media/sdcard0", and so on. The maximum length of a full path name in SylixOS is 512, plus the end character '\0', so the maximum length of the dev name field is 513.

Immediately following the device name (the end of the '\0' character) are four parameters that can be used for flexible expansion, and each of them has 4 bytes in length. These four parameters can be adapted to the special processing of different device messages. SylixOS does not specify the specific usage and storage format (large endian or small endian) of each parameter and is completely defined by the device driver.

From the above we can see that the maximum length of a hot-plug message is: $4 + 1 + 513 + 4 + 4 + 4 + 4 = 534$ bytes.

14.2.2 Processing the Hot-Plug Messages

The following is a specific example describing how to get and process hot-plug messages through the /dev/hotplug device.

Program List 14.1 Example of Processing Hot-plug Messages

```
#include <stdio.h>
#include <string.h>

#define MSG_LEN_MAX    (534)
```

```
int main(int argc, char *argv[])
{
    UINT8    pucMsgBuff[MSG_LEN_MAX];
    INT      iFd;
    INT32    iMsgType;
    BOOL     bInsert;
    CHAR     *pcDevName;
    UINT8    *pucArg;
    UINT8    *pucTemp;
    ssize_t  sstReadLen;

    iFd = open("/dev/hotplug", O_RDONLY);
    if (iFd < 0) {
        fprintf(stderr, "open /dev/hotplug failed.\n");
        return (-1);
    }

    while (1) {
        sstReadLen = read(iFd, pucMsgBuff, MSG_LEN_MAX);
        if (sstReadLen < 0) {
            fprintf(stderr, "read hotplug message error.\n");
            close(iFd);
            return (-1);
        }
        if (sstReadLen < 5) {
            continue;
        }
        /*
         * 解析热插拔消息
         */
        pucTemp = pucMsgBuff;
        iMsgType = (pucTemp[0] << 24)
            | (pucTemp[1] << 16)
            | (pucTemp[2] << 8)
            | (pucTemp[3]);

        pucTemp += 4;
        bInsert = *pucTemp ? TRUE : FALSE;

        pucTemp += 1;
        pcDevName = (CHAR *)pucTemp;

        pucArg = pucTemp + strlen(pcDevName) + 1;
    }
}
```



```

printf("get new hotplug message >>\n"
      " message type: %d\n"
      "device status: %s\n"
      " device name: %s\n"
      "      arg0: 0x%01x%01x%01x%01x\n"
      "      arg1: 0x%01x%01x%01x%01x\n"
      "      arg2: 0x%01x%01x%01x%01x\n"
      "      arg3: 0x%01x%01x%01x%01x\n",
      iMsgType,
      bInsert ? "insert" : "remove",
      pcDevName,
      pucArg[0],  pucArg[1],  pucArg[2],  pucArg[3],
      pucArg[4],  pucArg[5],  pucArg[6],  pucArg[7],
      pucArg[8],  pucArg[9],  pucArg[10], pucArg[11],
      pucArg[12], pucArg[13], pucArg[14], pucArg[15]);
}

close(iFd);
return (0);
}

```

In the above program, the obtained hot-plug message is subject to simple error processing, that means, the message length should be at least 5 bytes long, because a hot-plug message necessarily includes the message type and the device's plug-in/pull-out state. Note that when processing message types in the program, a parse is required according to the big endian data storage format, that means, the low address byte represents the high byte data. The starting address of the addition parameter of the message is namely the address of the device name plus the address of its length and the end character.

After the program is running, if we insert or unplug the SD memory card, the following message will be printed out:

Insert the SD memory card:

```

get new hotplug message >>
message type: 346
device status: insert
  device name: /media/sdcard0
    arg0: 0x0000
    arg1: 0x0000
    arg2: 0x0000
    arg3: 0x0000

```

Pull out the SD memory card:

```

get new hotplug message >>

```

```
message type: 346
device status: remove
device name: /media/sdcard0
arg0: 0x0000
arg1: 0x0000
arg2: 0x0000
arg3: 0x0000
```

The displayed message type value is 346 in decimal. Compared to the message type defined in the `<SylixOS/system/hotplugLib/hotplugLib.h>` file, it is equal to `LW_HOTPLUG_MSG_SD_STORAGE` (`0x0100+90`), which is namely the hot-plug message for the SD memory card device. The device name in the message is `/media/sdcard0`, which is the standard name for the SD memory card in SylixOS. In addition, other storage devices are also mounted by default under the `/media` directory. For example, the name of the U disk is `/media/udisk0`. The value of the 4 additional parameters is 0, indicating that the hot-plug message corresponding to SD memory card does not use this additional parameter (in fact, most of the hot-plug messages do not use the additional parameters).

The above program is an example for all hot-plug messages in a general detection system. It doesn't process the corresponding parameter messages for a specific message type, but prints out the value only in hexadecimal system. Since the `/dev/hotplug` device can be turned on multiple times by any program, in actual use, therefore, a program usually only needs to read the hot-plug message types that it concerns about, such as the `xinput` module in SylixOS, which only monitors the state of the input devices, such as USB mouse, USB keyboard, etc.



Chapter 15 Network I/O

15.1 socket interface

In the eyes of many underlying network application developers, all programming is socket, and almost all network programming relies on socket. When we open the browser every day to browse the website, the socket is required for communication between the browser process and the Web server. Socket is the interface between the process corresponding to the application program in the network communication and the network protocol, as shown in Figure 15.1.

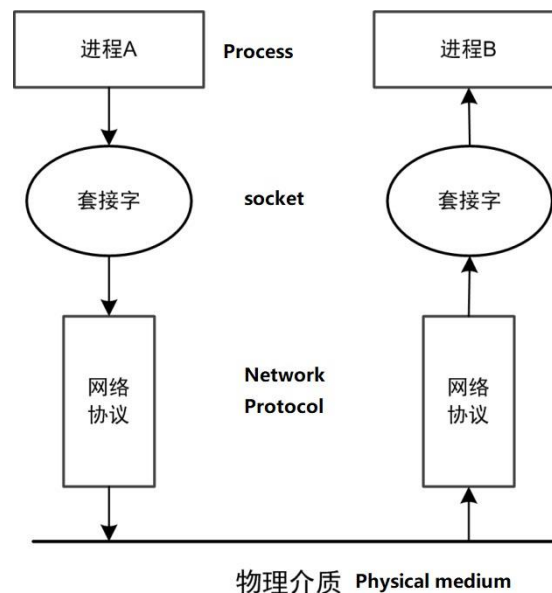


Figure 15.1 Position of socket in the network

Socket has the following functions in network transmission:

- The socket is located above the protocol, and shields the differences between different network protocols:
- Socket is the entrance of network programming. It provides a large number of system calls, and constitutes the main body of the network program;
- Socket is a device in the SylixOS file system, which can operate socket via the standard I/O function. This makes our control of the network as easy as control of files.

There are three common socket types: stream (SOCK_STREAM), datagram (SOCK_DGRAM) and raw (SOCK_RAW). Stream is a connection-oriented socket for

connection-oriented TCP service application. The datagram socket is a kind of connectionless socket corresponding to connectionless UDP service application. For TCP or UDP program development, the focus is on the Data field. We cannot directly modify the TCP or UDP header fields, and of course, the IP header. In other words, we have very limited space for their head operations, and can only use the source, destination IP, source and destination ports opened to us. For the raw type, one can obtain the raw IP packet, then customize the specific protocol type carried by IP, such as TCP, UDP or ICMP, and manually fill each type of packet carried in the IP protocol.

IP address and port number uniquely identify an application in the network communication, where the IP address plus the port number is called as socket, also referred to as the socket. The application layer programming interface designed for the TCP/IP is called as socket API. The socket API's location in the network hierarchy is shown in Figure 15.2.

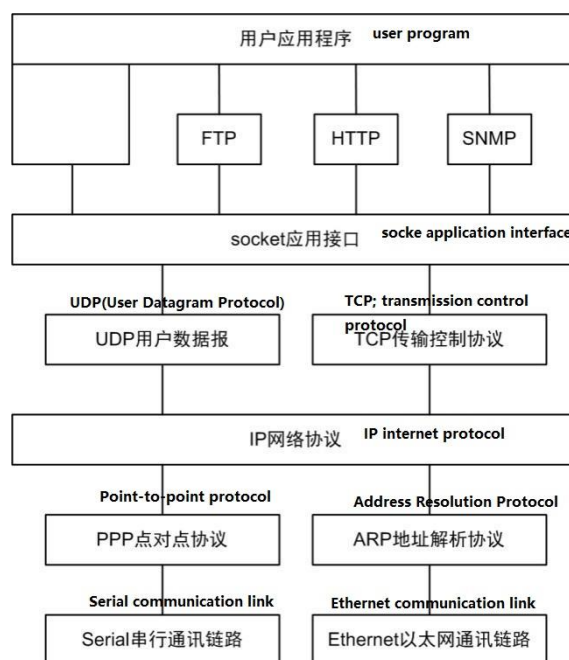


Figure 15.2 socket application interface

15.1.1 Network endian

We already know that the multi-byte data in memory has big and little endians with respect to the memory address, and the multi-byte data in the disk file also has big and little endians with respect to the offset address in the file. The network data stream also has the big endian and little endian. It is a topic requiring attention how to define the address of the network data stream. The send host usually sends the data in the send buffer zone according to the memory address from low to high. The receive host saves the byte received from the network in the receive buffer zone, i.e., save according to the memory address from low to high. Therefore, the address of the network data stream shall be specified as follows: the data sent first is the low address, and the data sent later is the high address.

The 16-bit data big-endian and little-endian represent the memory layout, as shown in Figure 15.3.

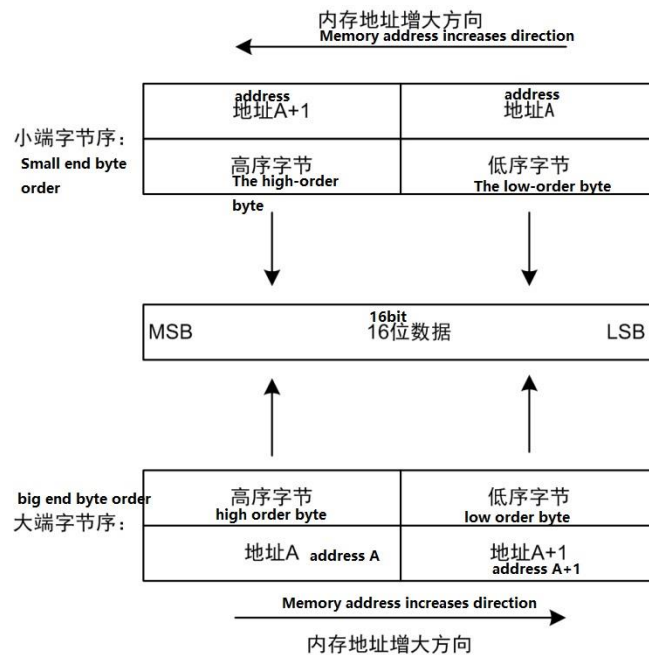


Figure 15.3 16-bit data big-endian and little-endian

The TCP/IP protocol stipulates that the network data stream shall adopt the big endian, that is to say, the low address stores the high byte. In order to make the network program portable, and the same C code run normally after compilation on big endian and little endian computers. The following library functions can be called for conversion of network endian and host endian.

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t x);
uint16_t htons(uint16_t x);
uint32_t ntohl(uint32_t x);
uint16_t ntohs(uint16_t x);
```

In the above function names, h represents host, n represents network, l represents long, and s represents short. For example, the htonl function converts the long integer from host endian to network endian, and returns it. For example, send after IP address conversion. If the host is the small endian, these functions will perform the corresponding big-little endian conversion of the parameters and return them. If the host is the big endian, these functions will not perform conversion, and return the parameters as they are. Although the <arpa/inet.h> header file is included when using these functions, system implementation often declares these functions in other header files. All of these header files are included in <arpa/inet.h>. For the system, these functions can also be implemented as macros.

15.1.2 Socket address

The socket API is applicable for a variety of underlying network protocols, such as IPv4, IPv6, and UNIX Domain sockets to be discussed later. However, the address formats of various network protocols are not the same. In order to enable addresses in different formats to be passed to the socket function, the address will be forcibly converted into a common address structure `sockaddr`.

The following is implementation in SylixOS system:

```
struct sockaddr {
    u8_t          sa_len;
    sa_family_t  sa_family;
#ifdef LWIP_IPV6
    char          sa_data[26]; /* sylixos add 4 bytes to a same size with in6 */
#else /* LWIP_IPV6 */
    char          sa_data[14];
#endif /* LWIP_IPV6 */
};
```

IPv4 (AF_INET) address `in_addr` structure:

```
struct in_addr {
    in_addr_t    s_addr;
};
```

IPv4 (AF_INET) address `sockaddr_in` structure:

```
struct sockaddr_in {
    u8_t          sin_len;
    sa_family_t  sin_family;
    in_port_t     sin_port;
    struct in_addr sin_addr;
#define SIN_ZERO_LEN 8
    char          sin_zero[SIN_ZERO_LEN];
};
```

- `Sin_len`: length of the data structure, increased to add OSI protocol support. With the length member, processing of variable-length socket address structure is simplified;
- `Sin_family`: address family (AF_INET);
- `Sin_port`: network protocol port number;
- `Sin_addr`: IPv4 address of network endian;
- `Sin_zero`: 8-byte data padding.

IPv6 (AF_INET6) address `in6_addr` structure:

```
struct in6_addr {
```

```

    union {
        u8_t    u8_addr[16];
        u32_t   u32_addr[4];
    } un;
#define s6_addr un.u8_addr
};

```

IPv6 (AF_INET6) address sockaddr_in6 structure:

```

struct sockaddr_in6 {
    u8_t          sin6_len;           /* length of this structure */
    sa_family_t   sin6_family;       /* AF_INET6 */
    in_port_t     sin6_port;         /* Transport layer port */
    u32_t         sin6_flowinfo;     /* IPv6 flow information */
    struct in6_addr sin6_addr;       /* IPv6 address */
    /*
     * sylixos add this
     */
    u32_t         sin6_scope_id;     /* set of interfaces for a scope */
};

```

- Sin6_len: length of data structure;
- Sin6_family: address family (AF_INET6);
- Sin6_port: transmission layer port number;
- Sin6_flowinfo: stream information, low order 20 bits are stream tags, high order 12 bits reserved;
- Sin6_addr: IPv6 address;
- Sin6_scope_id: scope ID.

UNIX protocol domain address sockaddr_un structure:

```

struct sockaddr_un {
    uint8_t       sun_len;           /* sockaddr len including null */
    uint8_t       sun_family;       /* AF_UNIX */
    char          sun_path[104];    /* path name (gag) */
};

```

- Sun_len: the length of the data structure;
- Sun_family: address family (AF_UNIX);
- Sun_path: path name;

Comparison of different socket address structures is shown in Figure 15.4.

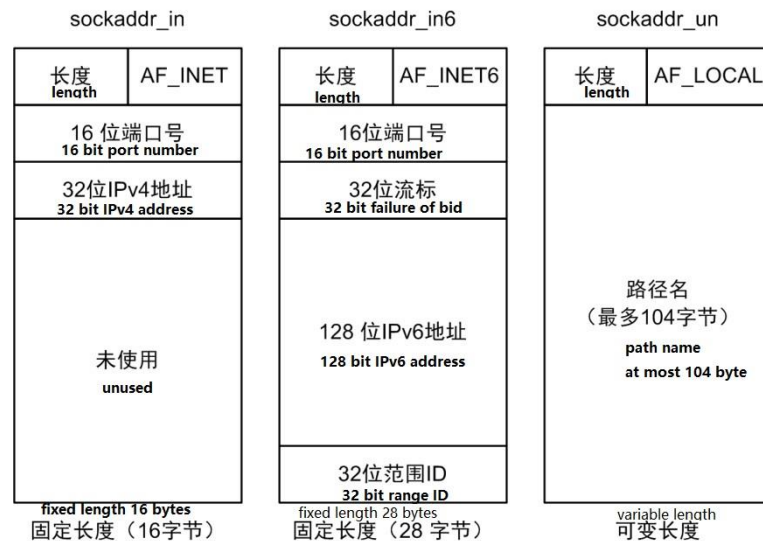


Figure 15.4 Comparison of different socket address structures

The member `sin_addr` in `sockaddr_in` represents the 32-bit IPv4 address. However, we usually use a dotted decimal character string to represent IPv4 addresses. The following functions can perform address conversion between the character string representation and the `in_addr` representation.

The dotted decimal character string (like "192.168.1.15") is converted to the 32-bit network endian binary value function:

```
#include <arpa/inet.h>
uint32_t inet_addr(const char *name);
```

Prototype analysis of Function `inet_addr`:

- For success of the function, return the 32-bit binary network endian address. For failure, return `INADDR_NONE`;
- Parameter ***name*** is a dotted decimal address, such as "192.168.1.15".

There is a problem in this function, which cannot represent all valid IP addresses (from 0.0.0.0 to 255.255.255.255). When the function fails, the return value is `INADDR_NONE` (usually a 32-bit value of 1), which means that the dotted decimal string 255.255.255.255 (which is the limited broadcast address for IPv4) cannot be converted by this function, because its binary value is used to indicate function failure. There is also a potential problem in this function. Some informal documents define the return value on error as -1 instead of `INADDR_NONE`. Therefore, there may be problem during comparison of the return value (unsigned value) and negative constant value of the function, which depends on the C compiler.

The dotted decimal character string (like "192.168.1.15") is converted to the 32-bit network endian binary value function:

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *name, struct in_addr *addr);
```

Prototype analysis of Function `inet_aton`:

- Return 1 if this function character string is valid and 0 if invalid.
- Parameter **name** is the dotted decimal address, such as "192.168.1.15";
- Parameter **addr** is the buffer address for storing network endian binary value.

This function has a feature not written into the official document. If the pointer is empty, the function still performs validity check of the input string, but does not store any result.

The 32-bit network endian binary value is converted to the dotted decimal character string (such as "192.168.1.15") function:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr addr);
```

Prototype analysis of Function `inet_ntoa`:

- Return the character string pointer when this function is correct, and return NULL when wrong;
- Parameter **addr** is the 32-bit network endian address.

Since the character string pointed to by this function's return value resides in the static memory, it means that the function is not reentrant, and this function takes the structure as the parameter. It is not a pointer to the structure. Normally, this function is designed as a macro.

The reentrant 32-bit network endian binary value is converted to the dotted decimal character string (such as "192.168.1.15") function:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa_r(struct in_addr addr, char *buf, int buflen);
```

Prototype analysis of Function `inet_ntoa_r`:

- Return the character string pointer when this function is correct, and return NULL when wrong;
- Parameter **addr** is the 32-bit network endian address;
- Parameter **buf** is the buffer zone for dotted decimal character string;
- Parameter **buflen** is the length of the buffer zone.

Inet_pton and inet_ntop are two newer address conversion functions which can process both IPv4 and IPv6 addresses. The letters p and n represent presentation and numeric respectively. The address presentation format is usually an ASCII string, and the numeric format is the binary value existing in the socket address structure.

Function inet_pton converts the character string address presentation to the binary value:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_pton(int af, const char *cp, void *buf);
```

Prototype analysis of Function inet_pton:

- Return 1 if the function is correct, -1 if wrong, and 0 if input is not a valid presentation format;
- Parameter **af** must be AF_INET or AF_INET6. Other address families are not supported and an error is returned;
- Parameter **cp** is the address string, such as "192.168.1.15" for IPv4;
- Parameter **buf** is used to save the binary results.

The inet_pton function does not specify the size of **buf**. Therefore, the application is required to guarantee that there is enough space to store 32-bit address at AF_INET and 128-bit address at AF_INET6.

The inet_ntop function converts the binary value to the character string expression:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
const char *inet_ntop(int af, const void *cp, char *buf, size_t len);
```

Prototype analysis of Function inet_ntop:

- Return the address string pointer when this function is correct, and return NULL when wrong;
- Parameter **af** must be AF_INET or AF_INET6. Other address families are not supported and an error is returned;
- Parameter **cp** saves the binary value;
- Parameter **buf** is used to save the address character string converted;
- Parameter **len** specifies the size of **buf** to avoid buffer area overflow.

The address conversion function is summarized in Figure 15.5.

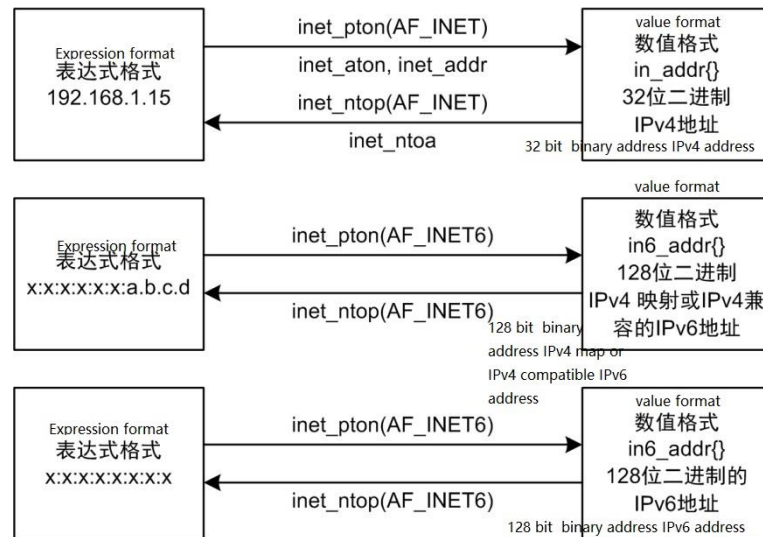


Figure 15.5 Address conversion

15.1.3 Socket function

In order to perform network I/O, the first thing which must be done is to call the socket function and specify the desired communication protocol type (TCP using IPv4, UDP using IPv6, UNIX domain protocol, etc.).

A socket can be created by calling the socket function:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Prototype analysis of Function socket:

- For success of the function, the file descriptor returned is also called the socket descriptor, which is simply called socketfd. For failure, return -1, and set the error number.
- Parameter **domain** is the protocol domain, also known as the protocol family. The protocol family determines the address type of the socket, and must use the corresponding address during communication. If AF_INET decides to use combination of IPv4 address (32-bit) and port number (16-bit), AF_UNIX decides to use an absolute path name as the address. Protocol family parameter description, as shown in Table 15.1;

Table 15.1 socket protocol domain

Protocol domain	Instructions
AF_UNSPEC	Not specified
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain
AF_UNIX	UNIX domain

AF_PACKET	PACKET domain
-----------	---------------

- Parameter **type** is the socket type. As shown in Table 15.2:

Table 15.2 Socket type

Socket type	Instructions
SOCK_DGRAM	Datagram socket, fixed length, connectionless, unreliable message passing
SOCK_RAW	Original socket, datagram interface of IP protocol
SOCK_SEQPACKET	Ordered packet socket, fixed-length, ordered, reliable, connection-oriented message passing
SOCK_STREAM	Byte stream socket, ordered, reliable, bidirectional and connection-oriented byte stream

- Parameter **protocol** is the protocol type:

Table 15.3 Protocol type

Protocol type	Instructions
IPPROTO_IP	IPv4 network protocol
IPPROTO_ICMP	Internet control message protocol
IPPROTO_TCP	Transmission control protocol
IPPROTO_UDP	User datagram protocol
IPPROTO_IPV6	IPv6 network protocol
IPPROTO_RAW	Original IP data packet protocol

Parameter **domain** determines the communication characteristics, including the address format and so on. AF_UNIX is an advanced IPC mechanism. The specific use will be described in detail in Section 15.5. AF_PACKET is a newer socket type, and supports operation on the data link layer. The specific use will be described in detail in Section 15.6. The AF_UNSPEC domain is not supported in SylixOS.

Parameter **protocol** is usually 0, indicating that the default protocol is chosen for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, **protocol** can be used to select a specific protocol. In the AF_INET

communication domain, the default protocol for the socket type SOCK_STREAM is the transmission control protocol (TCP), and the default protocol for the socket type SOCK_DGRAM is UDP.

For the datagram interface, no logical connection is required for communication between two peer-to-peer applications. It is only required send a message to the socket used by the peer-to-peer application. Therefore, the datagram (SOCK_DGRAM) provides a connectionless service. However, the bytestream (SOCK_STREAM) requires establishment of a logical connection between the local socket and the socket of peer-to-peer application of the communication before data exchange.

For the byte stream (SOCK_STREAM), the application cannot distinguish the boundaries of the message. This means that when reading data from the byte stream socket, it may not return the number of all bytes written by the sending application. All data sent can be got eventually, but may only be got through several function calls.

Not all combinations of socket protocol domain and type are valid. As shown in Table 15.4, some valid combinations and corresponding real protocols are given, where the item identified with "Y" signifies valid. Only a convenient abbreviation is not found, and the item identified with the "N" signifies unsupported.

Table 15.4 Protocol domain and type combination

Type	Protocol domain			
	AF_INET	AF_INET6	AF_UNIX	AF_PACKET
SOCK_STREAM	TCP	TCP	Y	N
SOCK_DGRAM	UDP	UDP	Y	Y
SOCK_RAW	IPv4	IPv6	N	Y

15.1.4 Socket option

The socket options is operated by calling the setsockopt function and the getsockopt function. The SylixOS network supports a variety of socket options, as shown in Table 15.5.

```
#include <sys/socket.h>
int setsockopt(int s, int level, int optname,
const void *optval, socklen_t optlen);
```

Prototype analysis of Function setsockopt:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);
- Parameter **level** is the option level, as shown in Table 15.5;
- Parameter **optname** is the option name, as shown in Table 15.5;
- Parameter **optval** is the option value;
- Parameter **optlen** is the option length.

Different options for different option levels are set by calling the setsockopt function. Parameter **optval** is the type of pointer to the variable. If the options are different, the types are also different, as shown in Table 15.5.

```
#include <sys/socket.h>
int getsockopt(int s, int level, int optname,
               void *optval, socklen_t *optlen);
```

Prototype analysis of Function getsockopt:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);
- Parameter **level** is the option level, as shown in Table 15.5;
- Parameter **optname** is the option name, as shown in Table 15.5;
- Output parameter **optval** returns the option value;
- Parameter **optlen** is the option length.

The socket option value can be obtained by calling the getsockopt function, and Parameter **optlen** will return the actual length of the option value.

Table 15.5 List of socket options

Option level	Option name	Note	Data type
SOL_SOCKET	SO_BROADCAST	Run to send the broadcast datagram	Int
	SO_ERROR	Get the pending error and eliminate it	Int
	SO_KEEPALIVE	Periodically test whether connection survives	Int

	SO_LINGER	Delay to close if there is data to send	struct linger
	SO_DONTLINGER	Turn off SO_LINGER option	int
	SO_RCVBUF	Accept the buffer zone size	int
	SO_RCVTIMEO	Accept timeout	struct timeval
	SO_SNDTIMEO	Send timeout	struct timeval
	SO_REUSEADDR	Allow reuse of local address	int
	SO_REUSEPORT	Allow reuse of local port	int
	SO_TYPE	Get socket type	int
	SO_CONTIMEO	Connection timeout	struct timeval
SOL_PACKET	PACKET_ADD_MEMBERSHIP	Join multi-cast group	struct packet_mreq
	PACKET_DROP_MEMBER	Leave multi-cast group	struct packet_mreq
	PACKET_RECV_OUTPUT	Whether to receive output data packet	int
	PACKET_RX_RING	Allocate memory space for mmap	struct tpacket_req
	PACKET_VERSION	Set AF_PACKET version	int
	PACKET_RESERVE	Reserved space for mmap to reserve extra header space	unsigned int
IPPROTO_IP	IP_TOS	Service type and priority	Int

	IP_TTL	Survival time	Int
	IP_MULTICAST_IF	Specify outgoing interface	struct in_addr
	IP_MULTICAST_TTL	Specify outgoing TTL	unsigned char
	IP_MULTICAST_LOOP	Specify whether to feed back	unsigned char
	IP_ADD_MEMBERSHIP	Join multi-cast group	struct in_mreq
	IP_DROP_MEMBERSHIP	Leave multi-cast group	struct in_mreq
IPPROTO_TCP	TCP_KEEPAIVE	Test the number of idle seconds before connection	Int
	TCP_KEEPIE	Permissible time before detection of a connection	Int
	TCP_KEEPIEVL	Time interval of two detections	Int
	TCP_KEEPCNT	Maximum times of detection	int
IPPROTO_IPV6	IPV6_V6ONLY	Only allow IPV6 (SylxOS does not support datagram communication)	int
IPPROTO_UDP LITE	UDPLITE_SEND_CSCOV	执行发送校验和	Int
	UDPLITE_RECV_CSCOV	执行接收校验和	int
IPPROTO_RA W	IPV6_CHECKSUM	IPV6 校验和	int

The option level of SOL_PACKET in Table 15.5 is used for AF_PACKET type socket, which are described in detail in Section 15.6.

The following program shows control of the behavior of the corresponding socket by setting the socket option. This program first sets the size of the IPv4 receive buffer zone by

calling the `setsockopt` function, and then calls the `getsockopt` function to get the receive the size of receive buffer zone to confirm that the size of receive buffer zone is correctly changed.

Program List 15.1 Set socket option

```
#include <stdio.h>
#include <sys/socket.h>

int main (int argc, char *argv[])
{
    int      sockfd;
    int      sopt = 2048;
    int      gopt;
    int      ret;
    socklen_t len = sizeof(int);

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        fprintf(stderr, "create socket error.\n");
        return (-1);
    }

    ret = setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &sopt, len);
    if (ret < 0) {
        fprintf(stderr, "setsockopt error.\n");
        return (-1);
    }

    ret = getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &gopt, &len);
    if (ret < 0) {
        fprintf(stderr, "getsocket error.\n");
        return (-1);
    }

    fprintf(stdout, "IPv4 recv buffer size: %d\n", gopt);
    return (0);
}
```

Run the program in SylixOS Shell, and the results are as follows:

```
# ./sockopt_test
IPv4 recv buffer size: 2048
```

15.2 Brief introduction to TCP/IP

TCP/IP is the abbreviation of Transmission Control Protocol / Internet Protocol, also known as the network communication protocol. It is the most basic protocol of the Internet, and the basis of the international Internet. It consists of IP protocol of the network layer and TCP protocol of the transmission layer. TCP/IP defines the criteria for access of electronic device to the Internet and data transmission between them. The 4-layer hierarchical structure is adopted for the protocol, and each layer calls the protocol provided by its next layer to complete its own demands. Popularly speaking, TCP is responsible for finding the problem in transmission. Once there is any problem, retransmission is required until all data is transmitted to the destination in a safe and correct manner. IP specifies an address for each Internet-connected device.

Data transmission process can be visually understood as two envelopes. TCP and IP are like envelopes. The information to be transmitted is divided into several segments. Each segment is stuffed into a TCP envelope, and the information of the section number is recorded on the envelope cover. Then insert TCP envelope into the IP large envelope, and send it to the network. At the receiving end, a TCP software package collects the envelopes, extracts the data, restores them in the sequence before sending, and verifies them. If an error is found, TCP will request retransmission. For ordinary users, they do not need to understand the entire structure of the network protocol, but only need to know the IP address format for network communications all over the world.

15.2.1 Layering of TCP/IP

Figure 15.6 shows the corresponding relation between the 4-layer structure of the TCP/IP protocol and the 7-layer structure of OSI.

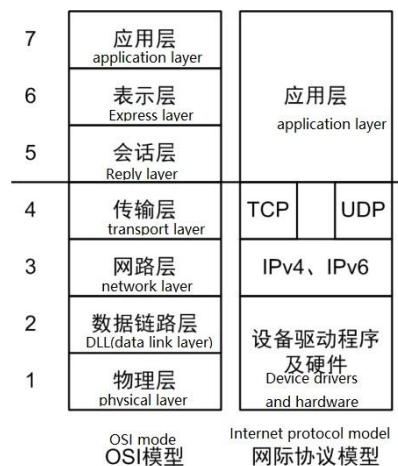


Figure 15.6 Network protocol model

TCP and UDP are the two most famous transport layer protocols, and both use IP as the network layer protocol. Although TCP uses unreliable IP services, it provides a kind of reliable transport layer service.

- Link layer: sometimes also called as the data link layer or network interface layer. It usually includes the device driver in the operating system and the corresponding network interface card in the computer. They deal with details of the physical interface associated with the cable (or any other transmission medium) together;
- Network layer: sometimes also called as the Internet layer, it deals with the activities of packets in the network, such as routing of packets. In the TCP/IP protocol family, the network layer protocols include IP protocol (Internet protocol), ICMP protocol (Internet Control Message Protocol), and IGMP protocol (Internet Group Management Protocol);
- Transport layer: it mainly provides end-to-end communications for applications on two hosts. In the TCP/IP protocol family, there are two different transmission protocols, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP provides data communication with high reliability. Its work includes: divide the data handed over by the application into appropriate small blocks to the following network layer, confirm the received packets, set the timeout clock sending the finally confirmed packet and so on. UDP provides a kind of very simple service for the application layer. It only sends packets called as datagram from one host to another, but does not guarantee that the datagram can reach the other end. Any necessary reliability must be provided by the application layer;
- The application layer is responsible for processing specific application details. Here are some common application protocols:
 - ◆ FTP file transmission protocol;
 - ◆ SMTP simple mail transmission protocol;
 - ◆ SNMP simple network management protocol.

There are many kinds of protocols in the TCP/IP protocol family. The common protocols are shown in Figure 15.7.

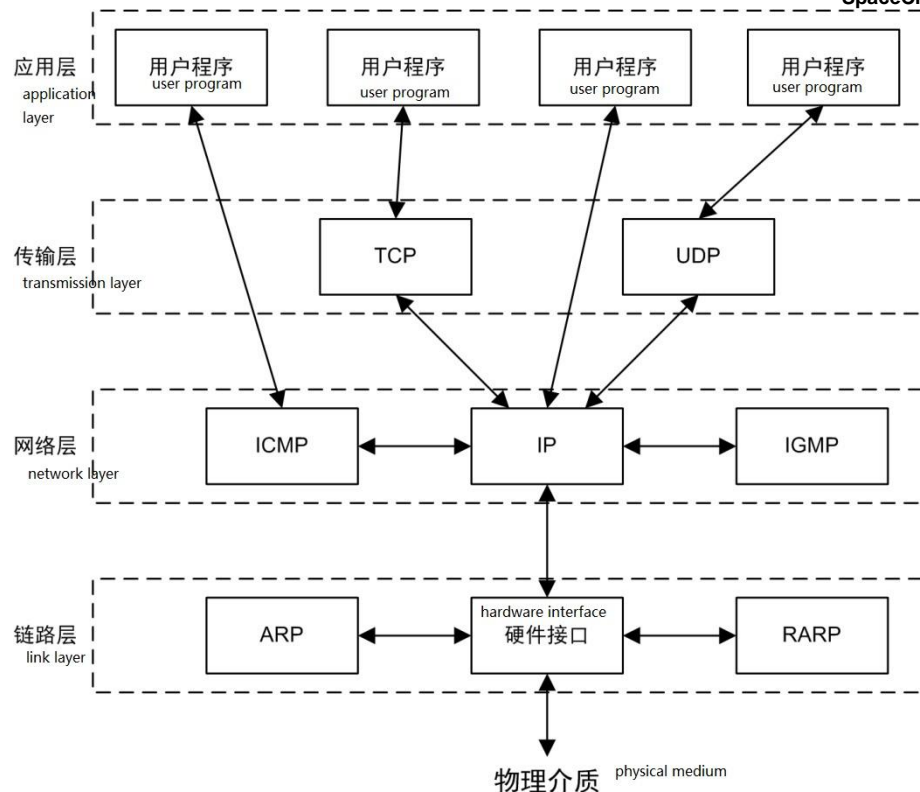


Figure 15.7 Layer Protocols in the protocol hierarchy

IPv4 (Internet Protocol version 4). IPv4 (usually called as IP) has been the main protocol of the Internet protocol family since the early 1980s. It uses 32-bit addresses to provide the passing packet service to TCP, UDP, ICMP and IGMP.

IPv6 (Internet Protocol version 6). IPv6 was designed in the mid-1990s to replace IPv4. The main change is use of 128-bit address.

TCP (Transmission Control Protocol) is a kind of connection-oriented protocol. It provides the users with the reliable full-duplex byte stream. TCP socket is a kind of stream socket. TCP cares for specific details such as acknowledgment, timeout and retransmissions.

UDP (User Datagram Protocol) is a kind of connectionless protocol. UDP socket is a kind of datagram socket. UDP datagrams are not guaranteed to eventually reach their destination.

IP (Internet Protocol). IP is the main protocol at the network layer, and is used by both TCP and UDP simultaneously. Each set of TCP and UDP data is transmitted in the Internet through the IP layer in the end system and each intermediate router.

ICMP (Internet Control Message Protocol). ICMP processes error and control messages between the router and the host (for example, the **ping** command to check whether the network is connected is the process of ICMP protocol work).

IGMP (Internet Group Management Protocol). IGMP is used for multi-cast. It is used to multi-cast a UDP datagram to multiple hosts.

ARP (Address Resolution Protocol). ARP maps IPv4 address to hardware address (such as Ethernet address). ARP is generally used for broadcast networks, such as Ethernet, token ring, etc., but not for point-to-point network.

RARP (Reverse Address Resolution Protocol). It maps hardware address to IPv4 address. It is sometimes used for booting diskless nodes.

Figure 15.8 lists all protocols involved in running the FTP protocol client-server mode.

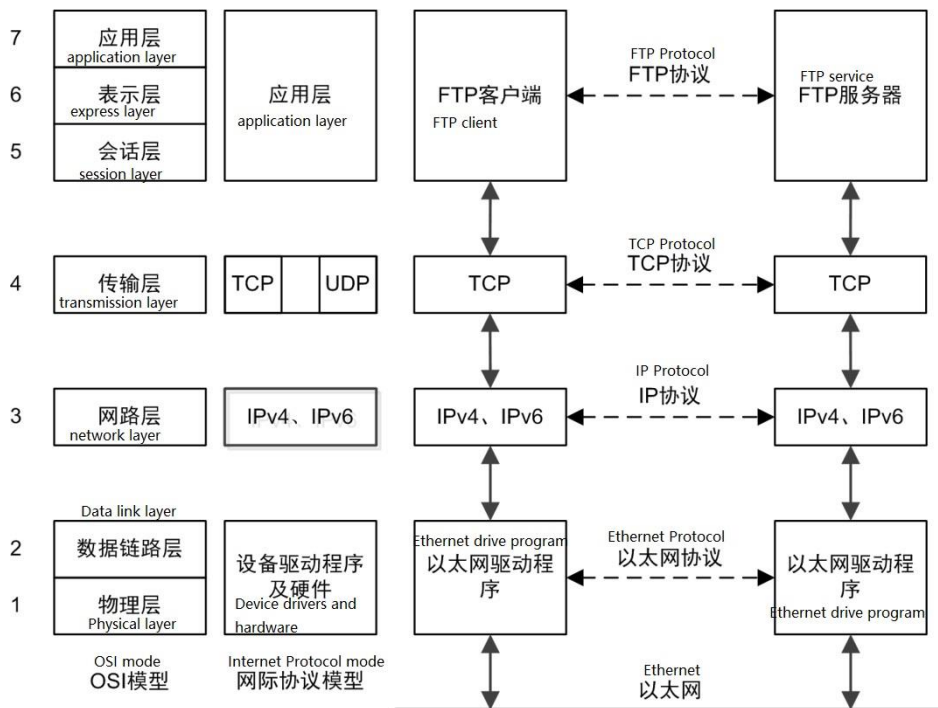


Figure 15.8 FTP communication model

15.2.2 IP address

Each interface on the Internet must have a unique Internet address (also called as IP address). The IP address is 32-bit long and has a certain structure. There are five different types of Internet address formats, as shown in Figure 15.9.

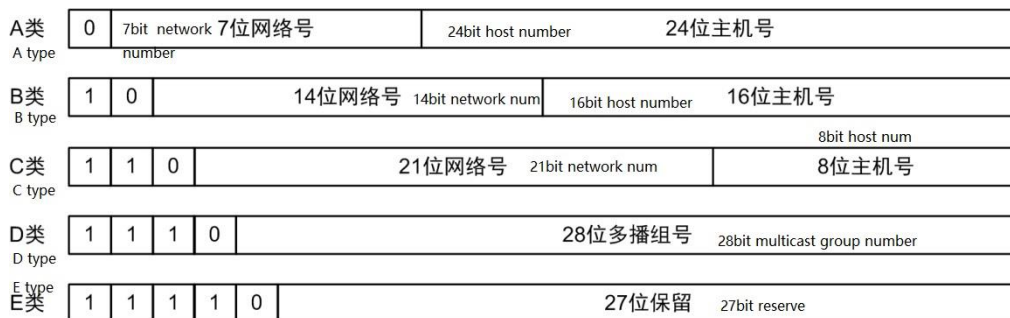


Figure 15.9 Five address representation methods

These 32-bit addresses are usually written as four decimal numbers, where each integer corresponds to one byte. This representation method is called as "dotted decimal notation". The easiest way to distinguish various types of addresses is to see its first decimal integer.

It shall be pointed out that multi-homed host has multiple IP addresses, and each interface corresponds to an IP address. Since each interface on the Internet must have a unique IP address, there must be a regulatory agency which assigns IP addresses to network accessed to the Internet. The regulatory agency is the Internet Network Information Center, called as InterNIC. The five address scopes are shown in Table 15.6. For example: 192.168.1.15 belongs to Class C address.

Table 15.6 Five address scopes

Type	Scope
Class A	0.0.0.0 to 127.255.255.255
Class B	128.0.0.0 to 191.255.255.255
Class C	192.0.0.0 to 223. 255.255.255
Class D	224.0.0.0 to 239. 255.255.255
Class E	240.0.0.0 to 247. 255.255.255

15.2.3 Data encapsulation

When the application transmits data via TCP, the data is sent to the protocol stack, and passes through each layer until it is sent to the network as a series of bit streams. Each layer will add some header information to the received data (sometimes tail information shall also be added). The process is shown in Figure 15.10.

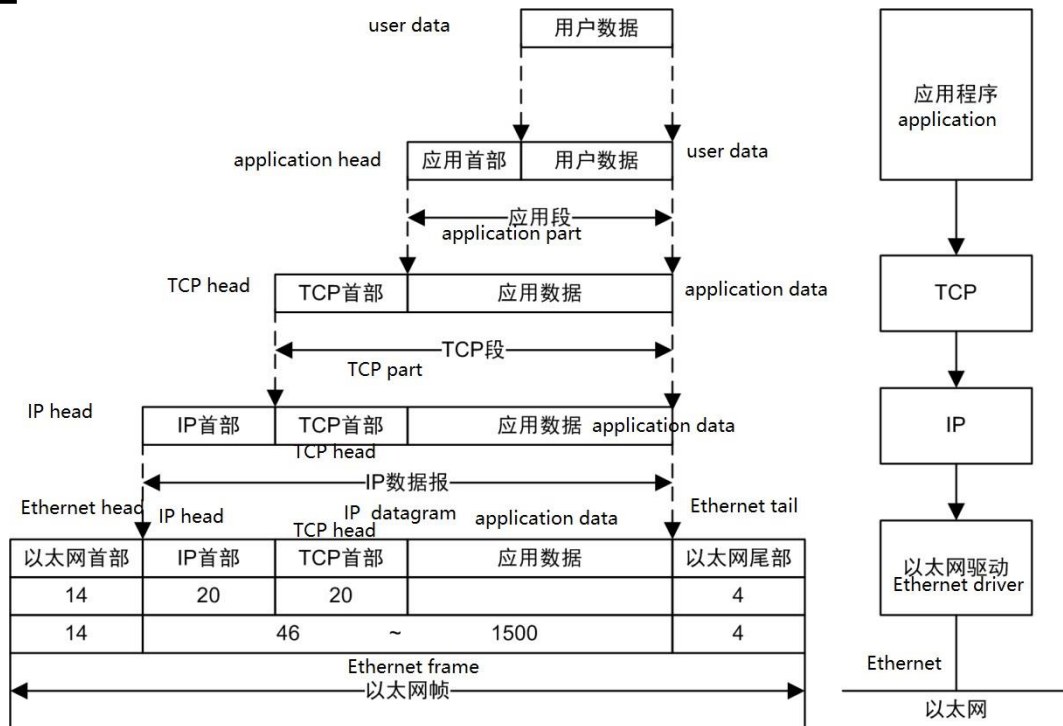


Figure 15.10 Protocol encapsulation of data stream

The bottom of Figure 15.10 shows the typical length of each protocol header. For example, the Ethernet header is 14 octet^①, the IP header is 20 octet and so on. The physical characteristics of the Ethernet data frame must be 46 and 1500 octet.

The data unit send by TCP to IP is called as TCP segment. The data unit send by IP to the network interface layer is called as IP datagram. The bit stream transmitted via the Ethernet is called as Frame.

UDP data is basically the same with TCP data. The only difference is that the information unit passed from UDP to IP is called as UDP datagram, but the header of UDP is 8 bytes. Since TCP, UDP, ICMP and IGMP send data to IP, IP must add some kind of identity to the generated IP header to indicate which layer the data belongs to. For this purpose, IP stores a value of 8 bits in length in the header, called as the protocol domain, 1 for the ICMP protocol, 2 for the IGMP protocol, 6 for the TCP protocol, and 17 for the UDP protocol.

15.2.4 Data demultiplexing

When the destination host receives an Ethernet data frame, the data starts to rise from the bottom of the protocol stack, while removing the message header added in each layer protocol. Each layer protocol must check the protocol identifier in the message header to determine the upper-layer protocol for receiving data. This process is called as data demultiplexing, as shown in Figure 15.11.

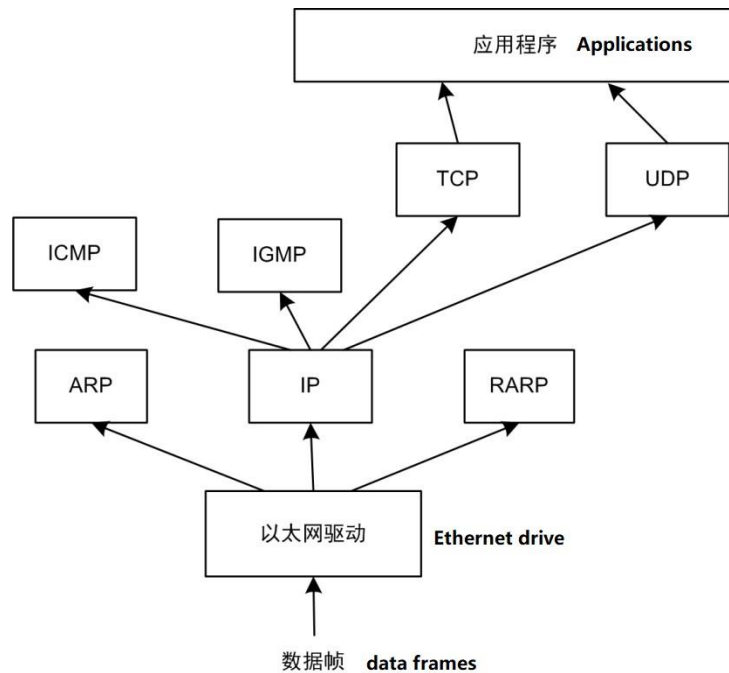


Figure 15.11 Ethernet data frame demultiplexing process

15.2.5 Port number

TCP and UDP use 16-bit port numbers to identify the application program. So how to select these port numbers? The servers are generally identified through the well-known port number. For example, for each TCP/IP implementation, TCP port number of FTP server is 21, TCP port number of each Telnet server is 23, and UDP port number of each TFTP (Simple File Transfer Protocol) server is 69. The service provided by any TCP/IP implementation adopts the well-known port numbers between 1 and 1023. These well-known port numbers are managed by the Internet Assigned Numbers Authority (IANA).

15.2.6 Link layer

TCP/IP supports a variety of different link layer protocols, depending on the hardware used in the network, such as Ethernet, token ring network, FDDI (Fiber Distributed Data Interface) and RS-232 serial line. It can be seen from Figure 15.7 that in the TCP/IP protocol family, the link layer mainly has the following purposes:

- Send and receive IP datagrams for IP modules;
- Send ARP request and receive ARP reply for ARP module;
- Send RARP request and receive RARP reply for RARP.

802.3 for the entire CSMA/CD network, 802.4 for the token bus network, and 802.5 for the token ring network. 802.2 and 802.3 define a frame format different from Ethernet. Encapsulation of Ethernet IP datagram is defined in RFC 894, and encapsulation of IP

datagram of IEEE 802 network is defined in RFC 1042. The encapsulation format is shown in Figure 15.12.

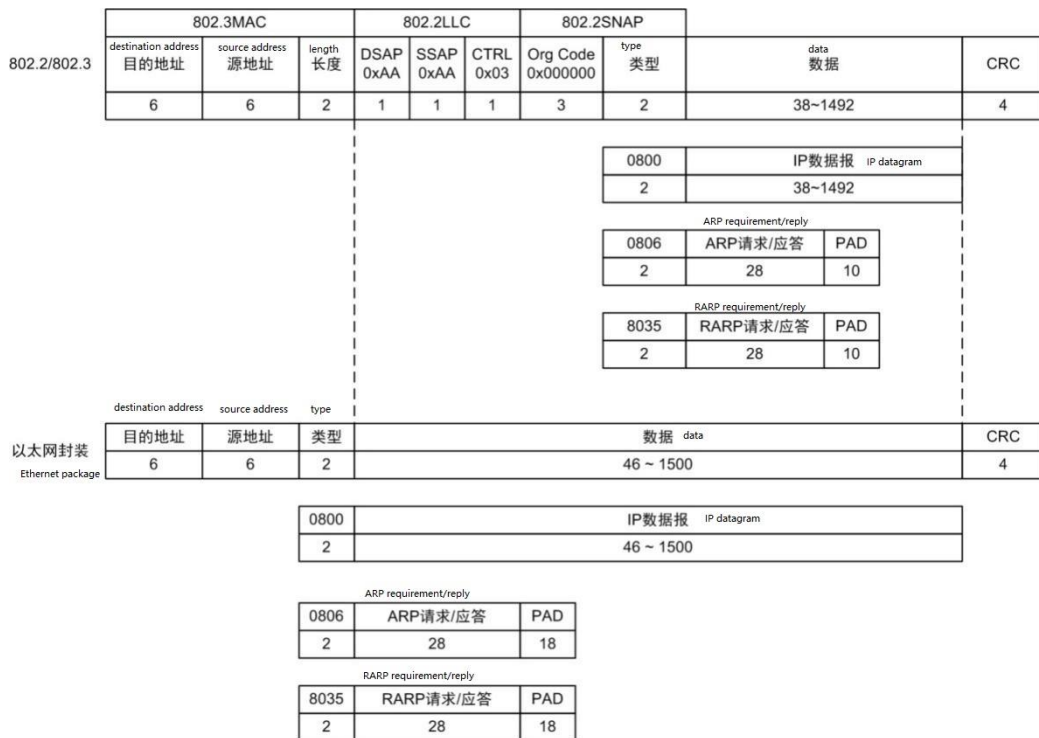


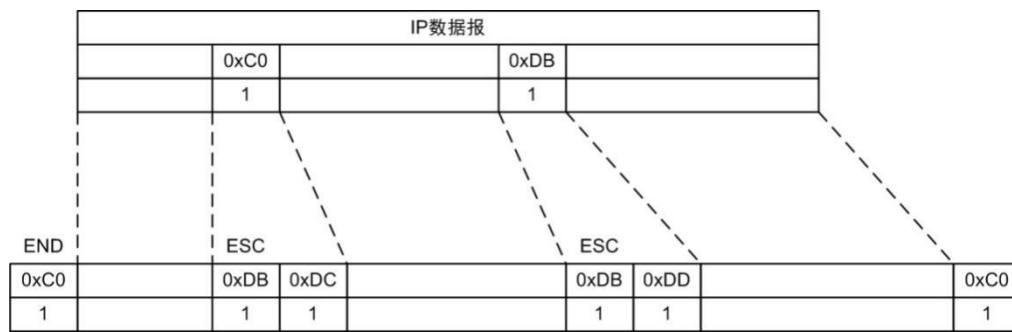
Figure 15.12 Encapsulation format of Ethernet frame

The full name of SLIP is Serial Line IP. It is a simple form of encapsulating IP datagram on the serial line, which is described in detail in RFC 1055. SLIP is suitable for RS-232 serial port and high-speed modem of each computer in the home for access to the Internet.

Frame format defined in SLIP:

- The IP datagram ends with a special character called as END (0xC0). At the same time, in order to prevent the line noise before arrival of datagram from being deemed as datagram content, most implementations also transmit an END character at the beginning of the datagram (if there is any line noise, then the END character will end this erroneous message. In this way, the current message can be transmitted correctly. After the previous error message is sent to the upper layer, it will be found that its content is meaningless and is discarded).
- If a character in the IP packet is END, then two bytes 0xDB and 0xDC shall be successively transmitted to replace it. The 0xDB special character is called as SLIP's ESC character, but its value is different from the ASCII code's ESC character (0x1B).
- If a character in the IP message is an ESC character of SLIP, then two bytes 0xDB and 0xDC are successively transmitted to replace it.

As shown in Figure 15.13, an IP message contains one END character and one ESC character. In this example, the total number of bytes transmitted on the serial line is the length of the original IP message plus 4 bytes.



(数据报=datagram)

Figure 15.13 SLIP message format

SLIP is a simple frame-packing method, with some drawbacks worth mentioning:

- Each end must know the other party's IP address. There is no way to notify the other end of the local IP address;
- There is no type field in the data frame (similar to the type field in Ethernet). If a serial line is used for SLIP, it cannot use other protocols at the same time;
- SLIP does not add the checksum in the data frame (similar to the CRC field in Ethernet). If the message transmitted by SLIP is influenced by the line noise, causing error, it can only be discovered through the upper layer protocol (another way is that the new modem can detect and correct the error message). In this way, it is important that the upper layer protocol provides some form of CRC.

PPP (point-to-point protocol) modifies all defects in the SLIP protocol. PPP includes the following three parts:

- Method of encapsulating IP datagrams on the serial link. PPP supports asynchronous mode with 8-bit data and without parity check (such as the serial interface commonly existing in most computers), and also supports the bit-oriented synchronous link;
- Link Control Protocol (LCP) to establish, configure and test the data link. It allows both parties to communications to negotiate to determine different options;
- Network Control Protocol (NCP) system for different network layer protocols.

PPP format encapsulation, as shown in Figure 15.14.

negotiations, and the protocol field is reduced from 2 bytes to 1 byte. Compare the PPP frame format with the SLIP frame format above, we will find that PPP adds only 3 extra bytes: 1 byte is reserved for the protocol field, and the other 2 are used for the CRC field. In addition, by using the IP network control protocol, most products can adopt Van Jacobson message header compression method (corresponding to CSLIP compression) through negotiations, so as to reduce the header length of IP and TCP.

In a word, PPP has the following advantages compared with SLIP:

- PPP supports running multiple protocols on the single serial line, not just the IP protocol;
- Each frame has cyclic redundancy check;
- Both parties to communications can dynamically negotiate IP addresses (by using the IP network control protocol);
- Similar to CSLIP, TCP and IP message headers are compressed;
- The link control protocol can set multiple data link options. The price to pay for these advantages is to add 3 bytes at the header of each frame, passing of several frames of negotiation data and more complicated implementation are required when the link is created.

Both Ethernet and 802.3 have a limit on the length of the data frame, and the maximum values are 1500 octet and 1492 octet respectively. The characteristic of the link layer is called MTU (maximum transmission unit).

If the IP layer has a datagram to be sent, and the length of the data is larger than that of MTU of the link layer, the IP layer shall be fragmented, and the datagram is divided into several pieces. Therefore, each one is smaller than the MTU in length.

When two hosts on the same network communicate with each other, MTU of the network is very important. However, if communication between two hosts shall pass multiple networks, the link layer of each network may have the different MTU. What's important is not the MTU value of the network which two hosts at, but is the minimum MTU of two host paths, which is called as the path MTU.

The path MTU between two hosts is not necessarily a constant. It depends on the route chosen at that time. However, route selection is not necessarily symmetric. Therefore, the path MTU is not necessarily consistent at both directions.

15.2.7 IP Internet protocol

IP is the most core protocol in the TCP/IP protocol family. All TCP, UDP, ICMP, and IGMP data are transmitted in the IP datagram format. Many people who are new to TCP/IP are surprised to find that IP provides unreliable and connectionless datagram delivery services.

Unreliable means that it cannot guarantee that the IP datagram can reach the destination successfully. IP only provides the best transmission service. In case of a certain error, such as a certain router temporarily running out of the buffer zone, IP has a simple error processing algorithm: discard the datagram, and then send the ICMP message to the source end. Any required reliability must be provided by the upper layer (such as TCP).

Term connectionless means that the IP does not maintain any status information about subsequent datagrams. Processing to each datagram is mutually independent. This also indicates that the IP datagram can be received not in the sending order. If a source sends two continuous datagrams (first A, then B) to the same sink, each datagram selects the route independently, and may select a different routes. Therefore, B may arrive before A arrives.

The format of the IP datagram is shown in Figure 15.15. The ordinary IP header is 20 bytes unless the option field is included.

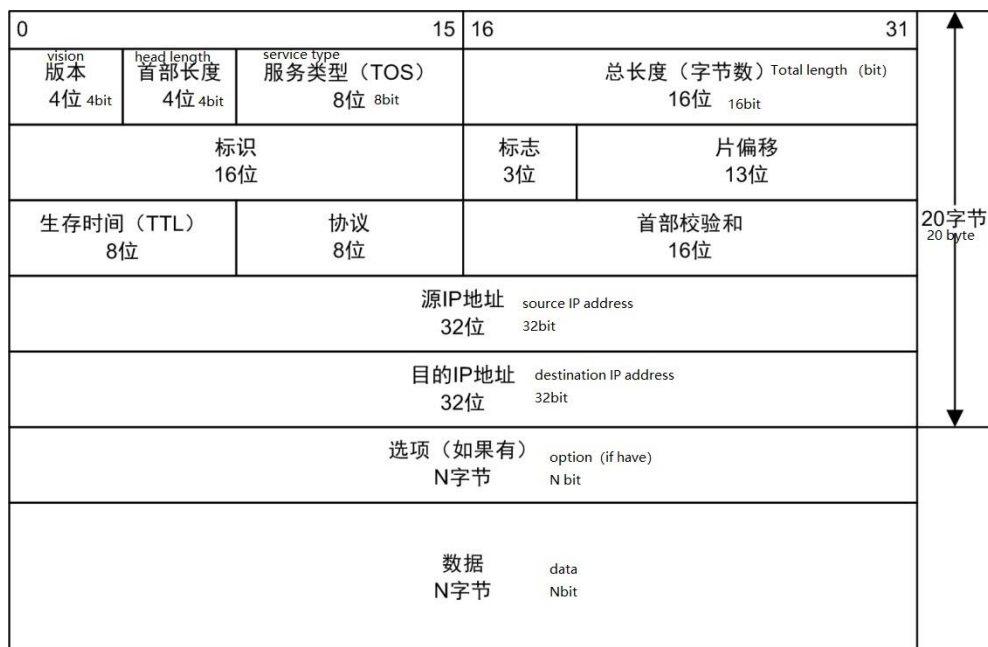


Figure 15.15 IP datagram format encapsulation

The most significant bit is on the left, recorded as 0 bit; the least significant bit is on the right, recorded as 31 bits. The 32-bit values of 4 bytes are transmitted in the following order: firstly 0 to 7 bit, secondly 8 to 15 bit, then 16 to 23 bit, and finally 24 to 31 bit. This transmission order is called big endian. All binary integers in the TCP/IP header shall be transmitted in the network in this order. Therefore, it is also called as the network endian. The machine which stores binary integers in other forms, such as the little endian format, must convert the header to the network endian before data transmission.

The version number of the current protocol is 4. Therefore, IP is called as IPv4 sometimes. The header length refers to the number of 32-bit characters occupied by the header, including any option. It is a 4-bit field. Therefore, the maximum length of the header is 60 bytes. The value of the ordinary IP datagram (without any option) field is 5. The type of service (TOS) field

includes a 3 bit priority subfield (ignored), 4 bit TOS subfield, and 1 bit unused bit but must be set to zero. TOS of 4 bit represents respectively: minimum delay, maximum throughput capacity, maximum reliability and minimum cost. Only 1 bit can be set in 4 bit. If all 4 bits are 0, it means general service.

The physical network layer generally limits the maximum length of each data frame sent. When the IP layer receives an IP datagram to be sent at any time, it shall judge which local interface the data is sent to, and inquires the interface to get MTU thereof. IP compares MTU with the datagram length, which shall be fragmented if necessary. Fragments can occur on the original sending host, or on the intermediate router.

After an IP datagram is fragmented, it is assembled only after arriving at the destination address. Reassembly is performed by the destination IP layer, the purpose is to make fragmentation and reassembly transparent to the transport layer (TCP and UDP), except for some possible leapfrog operations. The datagram fragmented may be fragmented again (maybe more than once). The data included in the IP header provides enough information for fragmentation and reassembly.

For each IP datagram sent from the sending end, its identity field contains a unique value. The value is copied to each slice when the datagram is fragmented, and the flag field uses one of the bits to represent "more slices." Except for the last slice, every slice which constitutes the datagram must set the bit as 1. The slice offset field refers to the position at the beginning of the slice offset original datagram. In addition, when the datagram is fragmented, the total length of each slice is changed as the length of the slice. Finally, there is a bit in the flag field called the "unfragmented" bit. If this bit is set as 1, IP will not fragment the datagram.

When IP datagram is fragmented, each slice will become a packet with its own IP header, and is independent from other packets during route selection. Therefore, when these slices of the datagram arrive at the destination, it may be out of order. However, there is enough information in the IP header to allow the receiving end to properly assemble the datagram fragments.

15.2.8 ARP address desorption protocol

When a host sends the Ethernet data frame to another host on the same LAN, the destination interface is determined according to the 48-bit Ethernet address. The device driver never checks the destination IP address in the IP datagram. Address resolution provides mapping for these two different address forms (32-bit IP address and any type of address used by the data link layer).

When the IP address is resolved on the Ethernet, the format of the ARP request and answer packet is shown in Figure 15.16 (ARP can be used for other types of networks, and can resolve addresses beyond the IP address, and the first four fields immediately following the frame type field appoint the type and length of the last four fields).

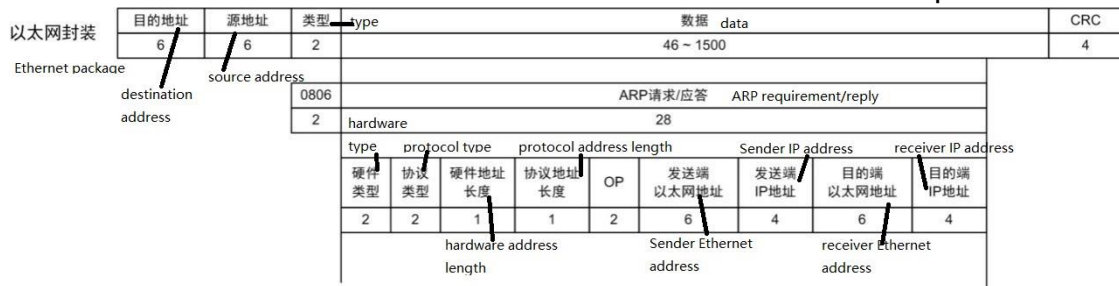


Figure 15.16 ARP format encapsulation

The first two fields in the Ethernet header are the Ethernet source and destination addresses. The special address with the destination address of 1 is the broadcast address, and all Ethernet interfaces on the cable must receive broadcast data frames. The two-byte long Ethernet frame type indicates the type of the following data. For ARP request or answer, the value of this field is 0x0806.

The hardware type field indicates the type of the hardware address. The value of 1 indicates the Ethernet address, and the protocol type field indicates the protocol address type to be mapped. Its value is 0x0800, i.e., indicating the IP address. Its value is the same with the value of the type field in the Ethernet data frame including the IP datagram.

The length of the hardware address and the length of the protocol address indicate the length of the hardware address and the protocol address respectively, in bytes. For the ARP request or answer to the IP address on the Ethernet, their values are 6 and 4 respectively.

The operation field indicates four operation types: ARP request (the value is 1), ARP answer (the value is 2), RARP request (the value is 3) and RARP answer (the value is 4).

For an ARP request, all other fields beyond the destination hardware address have filling values. After receiving an ARP request message where the destination is the host, the system will fill the hardware address, then replace the two sending addresses with the two destination addresses, set the operation field as 2, and send it back.

The key to high-efficient operation of ARP is that there is an ARP cache on each host. This cache stores the mapping record between the nearest Internet address to the hardware address.

The **arp** command can be used to view the ARP cache table, as shown below:

```
# arp -a
FACE INET ADDRESS    PHYSICAL ADDRESS  TYPE
en1 192.168.7.40     00:ff:ff:6f:a7:a0 dynamic
```

The 48 bit Ethernet address is indicated by 6 hexadecimal numbers, separated with the colon.

15.2.9 ICMP message control protocol

ICMP is deemed as a constituent part of the IP layer. It passes the error message and other information to be noticed, the ICMP message is usually used by the IP layer or the higher layer protocol (TCP or UDP), and some ICMP messages return the error messages to the user progress. ICMP message is shown in Figure 15.17.

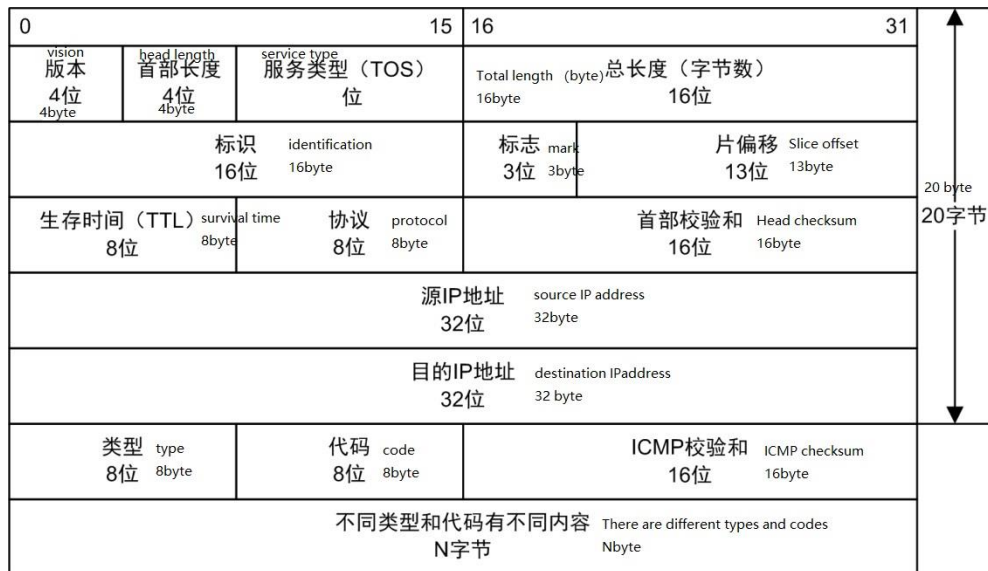


Figure 15.17 ICMP message format encapsulation

Various types of ICMP messages are shown in Table 15.7, different types are determined jointly by the type field and code field in the message.

Table 15.7 ICMP message type description

Type	Code	Description	Query	Mistake
0	0	Echo reply (ping reply)	●	
	0	Network unreachable		●
	1	Host unreachable		●
	2	Protocol unreachable		●
	3	Port unreachable		●
	4	Fragmentation is required, but the non-sliced bit is set		●
	5	Source station routing failed		●
	6	Destination network does not know		●
3	7	Destination host does not know		●
	8	Source host is isolated		●
	9	Destination network is forcibly prohibited		●
	10	Destination host is forcibly prohibited		●
	11	Network is unreachable due to service type TOS		●
	12	Host is unreachable due to service type TOS		●
	13	Communication is forcibly prohibited due to filtering		●
	14	Communication is forcibly prohibited due to filtering		●
	15	Priority suspension takes effect		●
4	0	Source end closed (basic flow control)		●
5	0	Redirection to network		●

	1	Redirection to host	●
	2	Service type and network redirection	●
	3	Service type and host redirection	●
8	0	Request echo (ping)	●
9	0	Router notice	●
10	0	Router request	●
	0	The survival time during transmission is 0	●
11	1	The survival time during datagram assembly is 0	●
	0	Bad IP header (including various mistakes)	●
12	1	Lack necessary options	●
13	0	Timestamp request	●
14	0	Timestamp reply	●
15	0	Information request (no longer used)	●
16	0	Information reply (no longer used)	●
17	0	Address mask request	●
18	0	Address mask reply	●

The following situations will not lead to ICMP error message:

- ICMP error message (however, ICMP query messages may generate ICMP error messages);
- IP datagram with the destination address as the broadcast address or multi-cast address;
- Datagram broadcast as a link layer broadcast;
- Not the first slice of IP fragmentation;

- The source address is not the datagram of the single host. That is to say, the source address cannot be zero address, loopback address, broadcast address or multi-cast address.

These rules are designed to prevent broadcast storm brought by previously allowed respond of ICMP error message to broadcast grouping.

15.2.10 UDP user datagram protocol

UDP is a simple datagram-oriented transport protocol. Unlike stream character-oriented protocols, such as TCP, all data generated by the application may not be related to the single IP datagram actually sent. UDP does not provide reliability: it sends data from the application to the IP layer, but does not guarantee that it can reach the destination. The UDP datagram is encapsulated in the format of an IP datagram, as shown in Figure 15.18.

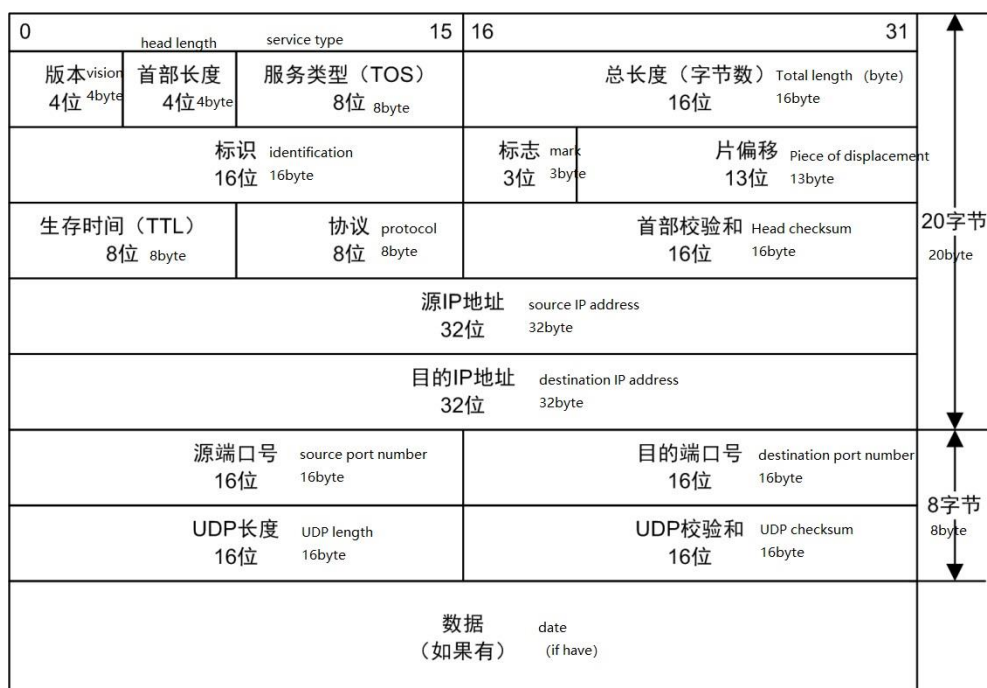


Figure 15.18 UDP data format encapsulation

The UDP length field refers to the byte length of the UDP header and UDP data. UDP checks and overwrites UDP headers and UDP data.

15.2.11 TCP transmission control protocol

Although both TCP and UDP use the same network layer (IP), TCP provides the application layer with service completely different from UDP. TCP provides a connection-oriented and reliable byte stream service. Connection-oriented means that two applications using TCP (usually a client and a server) must establish a TCP connection before

exchanging data with each other. The process is very similar to making a call: dial and wait for the other person to pick up the phone and say “hello,” and then explain who it is. Only two parties communicate with each other in a TCP connection.

TCP provides reliability via the following ways:

- The application data is divided into data block suitable for sending deemed by TCP. This is completely different from UDP, the length of the datagram generated by the application program will remain unchanged. and the information unit passed from TCP to IP is called a message segment or segment;
- When TCP sends a segment, it starts a timer, and waits for the destination to confirm receiving of the segment. If a confirmation cannot be received timely, the message segment will be resent;
- When TCP receives data sent from the other end of TCP connection, it will send a confirmation, which is not sent immediately, but usually be delayed by a little time.
- TCP will maintain the checksum of its header and data. This is an end-to-end checksum to detect any change in the data during transmission. If the checksum of the received segment has mistakes, TCP will discard the message segment and does not confirm receiving of the segment (hope sending end timeout and retransmission);
- TCP segments are transmitted as IP datagrams, while arrival of IP datagrams may be out of order. Therefore, arrival of TCP segments may also be out of order. If necessary, TCP will reorder the received data and pass the received data to the application layer in the correct order;
- Since IP datagrams will be duplicated, the receiving end of TCP must discard duplicate data.
- TCP can also provide traffic control, and each side of the TCP connection has the buffer space with fixed size. The receiving end of TCP only allows the other end to send data the buffer zone of the receiving end can accept, so as to prevent the faster host from causing buffer area overflow of the slower host.

TCP data is encapsulated in an IP datagram, as shown in Figure 15.19.

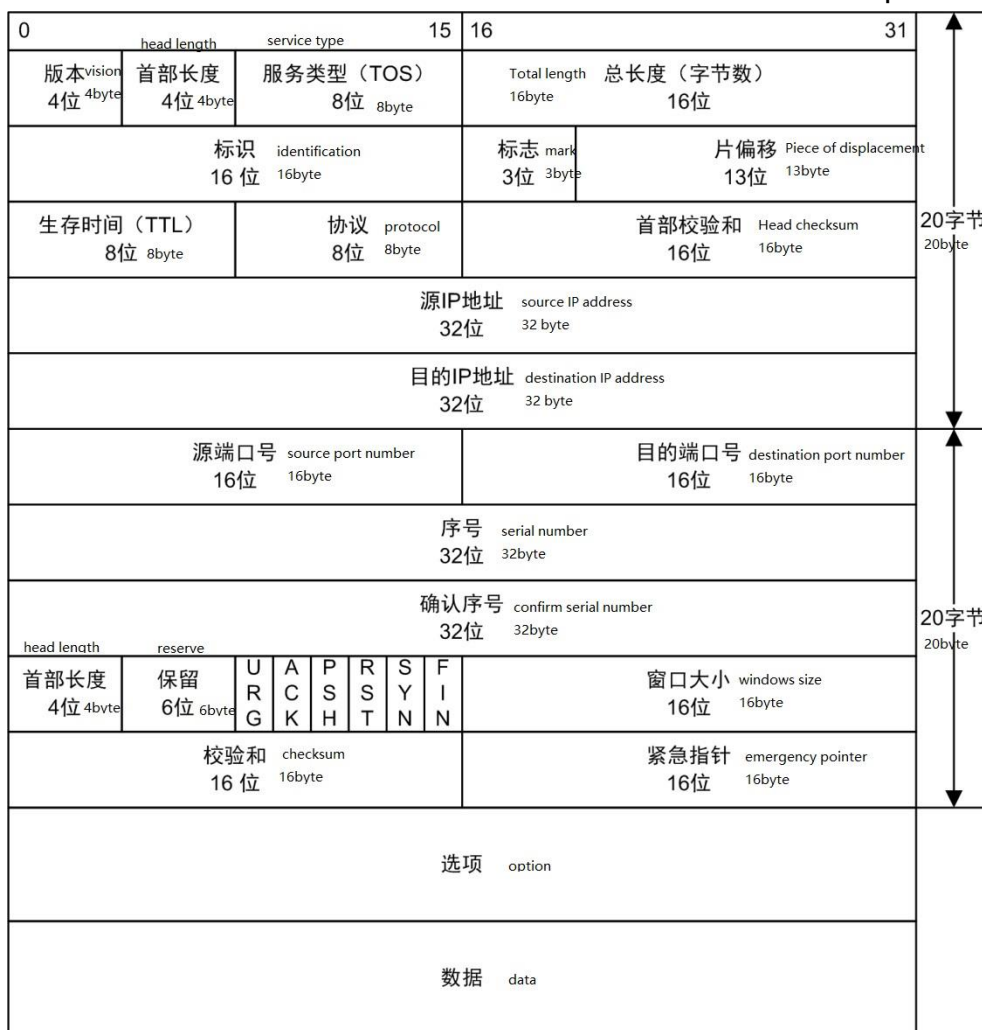


Figure 15.19 TCP data format encapsulation

Each TCP segment contains port numbers of the source end and the active end, which is used to find the sending and receiving end applications. These two values plus the source IP address and the destination IP address in the IP header uniquely confirm a TCP connection. Sometimes, an IP address and a port number are also called a socket, and four tuples, including client IP address, the client port number, server IP address and server port number may uniquely confirm both sides of each TCP connection in the network.

The serial number is used to identify the data byte stream sent from the TCP sending end to the TCP receiving end, which represents the first data byte in this message segment. If the byte stream is seen as one-way flow between two applications, TCP counts each byte with the serial number. The serial number is the 32-bit unsigned number, which starts from 0 after reaching the maximum value.

The SYN flag becomes 1 when a new connection is established. The serial number segment contains the initial sequence number (ISN) of the connection selected by the host. The sequence number of the first byte of data to be sent by the host is the ISN plus 1, because the SYN flag consumes a sequence number.

Since each transmitted byte is counted, the acknowledgment sequence number contains the next sequence number expected to be received at the end of the acknowledgment. Therefore, the acknowledgment sequence number shall be the sequence number of data bytes successfully received last time plus 1. The sequence number field is valid only when the ACK flag is 1. Sending ACK does not need any cost, because the 32-bit acknowledgment sequence number field is same with that of the ACK flag, always a part of the TCP header. Therefore, we see that once a connection is established, the field is always set, and the ACK flag is always set as 1.

TCP provides full-duplex service for the application layer, which means that data can be transmitted independently in two directions. Therefore, each end of the connection must maintain the serial number of the transmitted data in each direction. TCP can be expressed as a sliding window protocol without selective acknowledgment or denies. We say that TCP lacks selective acknowledgment because the acknowledgment sequence number in the TCP header indicates that the sender has successfully received the bytes. However, the bytes referred by the acknowledgment sequence number are not included. The selected part in the data stream cannot be confirmed at present. For example, if 1-1024 bytes have been successfully received, the next message segment contains bytes with sequence numbers from 2049 to 3072, and the receiving end cannot acknowledge the new message segment. All it can do is send back an ACK with an acknowledgment sequence number of 1025, which also cannot deny a message segment. For example, if the message segment containing 1025 to 2048 bytes is received, but its checksum is wrong, what the TCP receiving end can do is send back an ACK with an acknowledgment sequence number of 1025.

The length of the header gives the number of 32-bit characters in the header, the value is required because the length of the optional field is variable, occupying 4 bits. Therefore, TCP has the header with a maximum of 60 bytes. However, the normal length is 20 bytes without the optional field. There are 6 flag bits in the TCP header, and many of them can be set as 1 simultaneously.

6 bit flag in the TCP header:

- URG (urgent pointer) is valid;
- ACK acknowledgment sequence number is valid;
- The PSH receiver shall submit the message segment to the application layer as soon as possible.
- RST reconstruction connection;
- The SYN synchronization serial number is used to initiate a connection;
- The FIN sending end completes the sending task.

Figure 15.20 shows the process of establishing a TCP connection:

- The request end (usually called the client) sends a SYN segment to indicate the server port to be connected by the client, and the initial sequence number (ISN);

- The server sends back the SYN segment containing the initial sequence number of the server as an answer. At the same time, the acknowledgment sequence number is set as the customer's ISN plus 1 to confirm the customer's SYN message segment. A SYN will occupy a sequence number;
- The client must set the acknowledgment sequence number as the server's ISN plus 1 to confirm the server's SYN message segment.

This process is also known as three-way handshake.

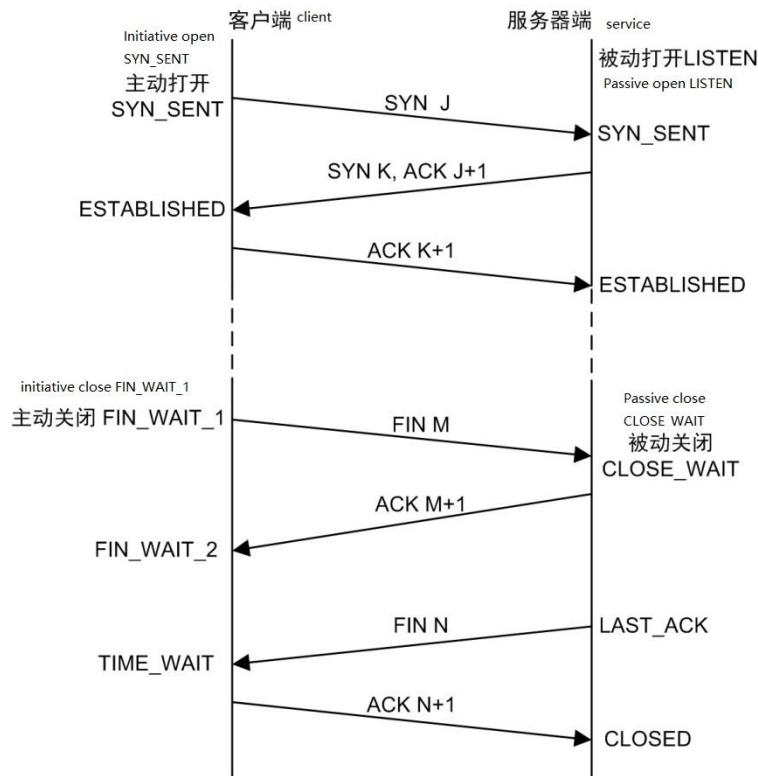


Figure 15.20 Establishment and suspension of TCP connection

Three-way handshake is required for establishing a connection, while four-way handshake is required for terminating a connection. This is caused by the half-close of TCP. Since a TCP connection is full-duplex (i.e., data can be passed in two directions simultaneously), each direction must be closed independently. The principle is that a FIN can be sent to terminate the direction connection after one side finishes its data sending task. When a FIN is received at an end, it must notify the other end of the application layer that the data transfer in that direction has been terminated. Sending FIN is usually the result of the application layer closing.

Receiving a FIN only means that there is no data flow in this direction. A TCP connection can still send data after receiving a FIN. This is possible for the semi-closed applications, although only a few TCP applications do this in practical application. The normal shutdown process is shown in Figure 15.20. Firstly, a side performing closing (i.e., send the first FIN) will

perform active close, while the other side (receive the FIN) will perform passive close. Usually a party completes active close while the other side completes passive close.

The TIME_WAIT state is also called as the 2MSL wait state. Each specific TCP implementation must select MSL (Maximum Segment Lifetime) of a message segment. It is the longest time in the network before any segment is discarded. We know that the time is limited, because TCP segments are transmitted in the network as IP datagrams, while IP datagrams have TTL fields limiting the lifetime. RFC 793 indicates that the MSL is 2 minutes. However, the common value in implementation is 30 seconds, 1 minute, or 2 minutes. The limit on the TTL of IP datagrams is based on the hop count instead of the timer. For MSL value given by specific implementation, the processing principle is: when TCP performs an active close and sends back the last ACK, the connection must stay at the TIME_WAIT state for 2 times the MSL. Therefore, TCP can resend the last ACK to prevent the ACK from loss (the other side times out and resends the last FIN). Another result of such 2MSL wait is that the socket defining TCP connection (the client's IP address and port number, the server's IP address and port number) cannot be reused. The connection can only be used after the 2MSL is ended.

15.3 Network communication instance

The client-server model is also called the master-slave architecture, or C/S structure for short. It is a network architecture which distinguishes the client from the server. Each client application instance can send a request to a server. There are many different types of servers, such as file servers, terminal servers, mail servers and so on. Although the purpose of their existence is different, the basic structure is the same.

This method is applied for many different types of applications in different ways, and the most common one is the web pages currently used on the Internet. For example, when you browse the SylixOS website on your computer, your computer and web browser are regarded as a client, and the host constituting the SylixOS website is regarded as the server. When your web browser requests an appointed article from the SylixOS site, the SylixOS site server finds all information required for the article from the database, combines it into a web page, and sends it back to your browser.

Server-side features: passive roles, wait for requests from clients, process requests and return results.

Client features: active role, send request and wait for response to the request.

15.3.1 UDP instance

There is essential difference between writing applications with UDP and TCP, the reason is the difference between the two transport layers: UDP is the connectionless and unreliable datagram protocol, which is different from connection-oriented reliable byte stream provided by TCP. However, some occasions is more suitable to use UDP. Some popular applications

written with UDP are DNS (Domain Name System), NFS (Network File System) and SNMP (Simple Network Management Protocol).

The client does not need to establish connection with the server, but the destination address shall be appointed (server address), and the client only sends datagrams to the server.

The server does not accept connections from clients, but only waits for data from a certain client.

The typical UDP client and server side functions are shown in Figure 15.21.

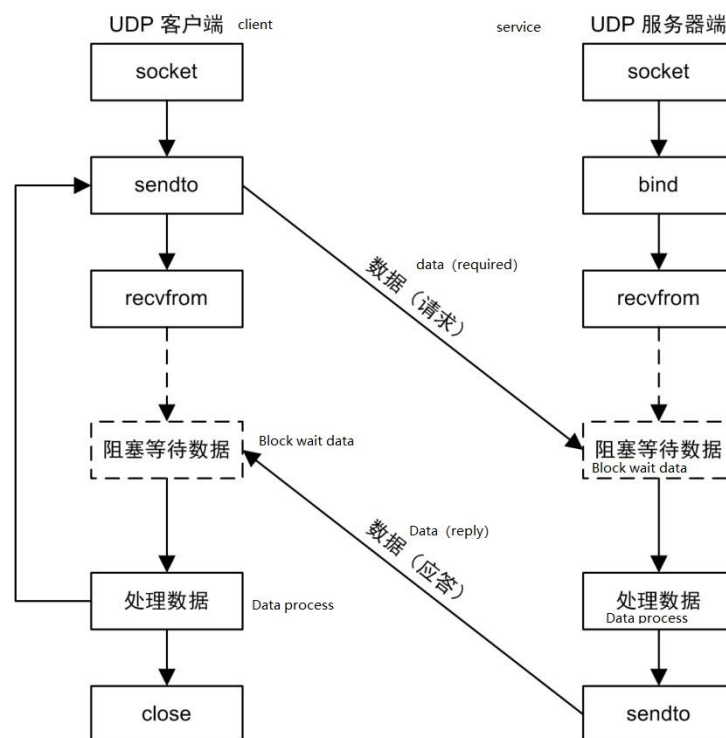


Figure 15.21 UDP program socket function

The bind function assigns a local protocol address to a socket, and the meaning of the protocol address depends only on the protocol itself. For Internet protocols, the protocol address is combination of the 32-bit IPv4 address or the 128-bit IPv6 address and the 16-bit TCP or UDP port. Calling the bind function can appoint the IP address or port, and both can be appointed or not appointed.

```
#include <sys/socket.h>
int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

Prototype analysis of Function bind:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);

- Parameter **name** is a pointer to the sockaddr structure type of the specific protocol domain;
- Parameter **namelen** represents the length of **name** structure.

Figure 15.21 shows how to use `recvfrom` and `sendto` in the typical UDP client-side and server-side programs (similar to the standard `read` and `write` functions, but three additional parameters are required).

```
#include <sys/socket.h>
ssize_t recvfrom(int s, void *mem, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

Prototype analysis of Function `recvfrom`:

- For success of the function, return the number of bytes of the data read. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);
- Parameter **mem** is the pointer to the read buffer zone;
- Parameter **len** represents the byte length of the read data;
- Parameter **flags** is used to specify the message type. If you do not care about this parameter, you can set it to 0. If you need to care about this parameter, please configure its value to the following value:
 - ◆ MSG_PEEK: pre-read data but do not delete data;
 - ◆ MSG_WAITALL: wait until all data arrives before return;
 - ◆ MSG_OOB: out-of-band data;
 - ◆ MSG_DONTWAIT: non-blocking receiving data;
 - ◆ MSG_MORE: there is more data to send.
- Parameter **from** used to indicate the protocol address of the UDP datagram sender (for example, IP address and port number);
- Parameter **fromlen** is used to specify a pointer to the size of **from** address.

Since UDP is connectionless, it is also possible that the `recvfrom` function returns a value of 0. If the **from** parameter is a null pointer, then the corresponding length parameter **fromlen** must also be a null pointer, indicating that we do not care about the protocol address of the data sender.

```
#include <sys/socket.h>
ssize_t sendto(int s, const void *data, size_t size, int flags,
               const struct sockaddr *to, socklen_t tolen);
```

Prototype analysis of Function `sendto`:

- For success of the function, return the number of bytes of the data read. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);
- Parameter **data** is a pointer to the write data buffer zone;
- The parameter **size** indicates the byte length of data written;
- Parameter **flags** is used to specify the message type. If you do not care about this parameter, you can set it to 0. If you need to care about this parameter, please configure its value to the following value:
 - ◆ MSG_PEEK: pre-read data but do not delete data;
 - ◆ MSG_WAITALL: wait until all data arrives before return;
 - ◆ MSG_OOB: out-of-band data;
 - ◆ MSG_DONTWAIT: non-blocking receiving data;
 - ◆ MSG_MORE: there is more data to send.
- Parameter **to** is used to indicate the protocol address of the UDP datagram receiver (for example, IP address and port number);
- Parameter **toLen** specifies the length of **to** address.

The sendto function is feasible to write a datagram of length 0, which results in a datagram with IP header (usually 20 bytes for IPv4, 40 bytes for IPv6) and an 8-byte UDP header but without data.

The UDP echo program model is shown in Figure 15.22. The client and the server follow this flow to complete reception and echo of the retroflection data.

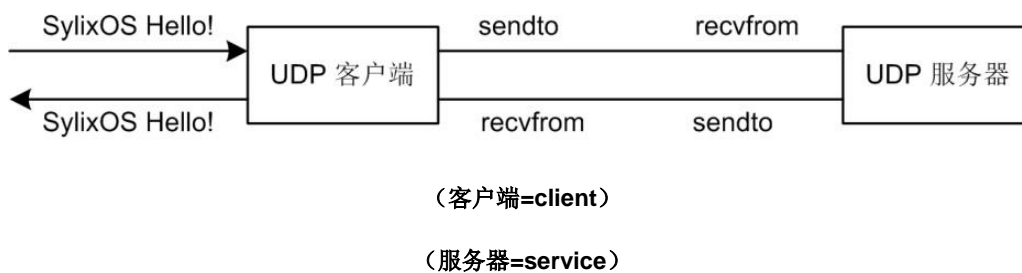


Figure 15.22 UDP echo program model

The client program uses the sendto function to send "SylixOS Hello!" to the server, uses recvfrom to read back the server's echo, and finally outputs the received echo information "SylixOS Hello!".

The server program uses the recvfrom function to read in the "SylixOS Hello!" data from the client, and sends the received data to the client program via sendto.

Program List 15.2 UDP echo server program

```

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

#define __UDP_ECHO_TYPE_CLIENT      1      /* Client mode          */
#define __UDP_ECHO_TYPE_SERVER     2      /* Server mode         */
/* Current mode selection */
#define __UDP_ECHO_TYPE            (__UDP_ECHO_TYPE_SERVER)
/* Client IP address     */
#define __UDP_ECHO_IP_CLIENT       "192.168.1.16"
/* Server IP address     */
#define __UDP_ECHO_IP_SERVER       "192.168.1.17"
#define __UDP_ECHO_PORT_CLIENT     8000   /* Client port number   */
#define __UDP_ECHO_PORT_SERVER     8001   /* Server port number   */
#define __UDP_ECHO_BUFF_SIZE_CLIENT 257   /* Client receive buffer size */
#define __UDP_ECHO_BUFF_SIZE_SERVER 257   /* Server receive buffer size */
static int __UdpEchoServer (void)
{
    int          iRet      = -1;          /* Operation result      */
    int          sockFd    = -1;         /* socket Descriptor     */
/* Address structure size */
    socklen_t    uiAddrLen = sizeof(struct sockaddr_in);
    register ssize_t sstRecv = 0;       /* Received data length  */
/* Receive buffer */
    char          cRecvBuff[__UDP_ECHO_BUFF_SIZE_SERVER] = {0};
    struct sockaddr_in sockaddrinLocal;  /* Local address         */
    struct sockaddr_in sockaddrinRemote; /* Remote address        */

    fprintf(stdout, "UDP echo server start.\n");
/* Create socket */
    sockFd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockFd < 0) {
/* operation failed */
        printf("UDP echo server socket error.\n");
        return (-1);
/* Error return */
    }
/*
 * Initialize the local address structure
 */
/* Clear address information */
    memset(&sockaddrinLocal, 0, sizeof(sockaddrinLocal));
/* Address structure size */
    sockaddrinLocal.sin_len      = sizeof(struct sockaddr_in);
    sockaddrinLocal.sin_family   = AF_INET; /* Address family */
/* net address */

```

```

sockaddrinLocal.sin_addr.s_addr = INADDR_ANY;
                                /* Binding server port      */
sockaddrinLocal.sin_port      = htons(__UDP_ECHO_PORT_SERVER);
iRet = bind(sockFd,
            (struct sockaddr *)&sockaddrinLocal,
            sizeof(sockaddrinLocal)); /* Bind local address and port */
if (iRet < 0) { /* Binding operation failed */
    close(sockFd); /* Close the socket that has been
created */
    fprintf(stderr, "UDP echo server bind error.\n");
    return (-1); /* Error return */
}
for (;;) {
                                /* Clear the receive buffer */
memset(&cRecvBuff[0], 0, __UDP_ECHO_BUFF_SIZE_SERVER);

sstRecv = recvfrom(sockFd,
                  (void *)&cRecvBuff[0],
                  __UDP_ECHO_BUFF_SIZE_SERVER,
                  0,
                  (struct sockaddr *)&sockaddrinRemote,
                  &uiAddrLen); /* Receive data from the remote */
if (sstRecv <= 0) { /* Failed to receive data */
    if ((errno != ETIMEDOUT ) &&
        (errno != EWOULDBLOCK)) { /* Non-timeout and non-blocking */
        close(sockFd); /* Close the socket that has been
created */
        fprintf(stderr, "UDP echo server recvfrom error.\n");
        return (-1);
    }
    continue;
}
sendto(sockFd,
       (const void *)&cRecvBuff[0],
       sstRecv,
       0,
       (const struct sockaddr *)&sockaddrinRemote,
       uiAddrLen);
}
return (0);
}

```

Program List 15.3 UDP echo client program

```
#include <stdio.h>
```

```

#include <string.h>
#include <sys/socket.h>

#define __UDP_ECHO_TYPE_CLIENT      1      /* Client mode          */
#define __UDP_ECHO_TYPE_SERVER     2      /* Server mode         */
/* Current mode selection */
#define __UDP_ECHO_TYPE            (__UDP_ECHO_TYPE_CLIENT)
/* Client IP address     */
#define __UDP_ECHO_IP_CLIENT       "192.168.1.16"
/* Server IP address     */
#define __UDP_ECHO_IP_SERVER       "192.168.1.17"
#define __UDP_ECHO_PORT_CLIENT     8000   /* Client port number   */
#define __UDP_ECHO_PORT_SERVER     8001   /* Server port number   */

#define __UDP_ECHO_BUFF_SIZE_CLIENT 257   /* Client receive buffer size */
#define __UDP_ECHO_BUFF_SIZE_SERVER 257   /* Server receive buffer size */

static int __UdpEchoClient (void)
{
    int          sockFd      = -1;        /* socket Descriptor     */
/* Address structure size */
    socklen_t    uiAddrLen  = sizeof(struct sockaddr_in);
    register ssize_t sstRecv = 0;        /* Received data length */
    register ssize_t sstSend = 0;        /* Received data length */
/* String to send        */
    const char   *pcSendData = "SylixOS Hello!\n";
/* Receive buffer        */
    char          cRecvBuff[__UDP_ECHO_BUFF_SIZE_CLIENT] = {0};
    struct sockaddr_in sockaddrinRemote; /* Remote address       */

    fprintf(stdout, "UDP echo client start.\n");
/* Create socket        */
    sockFd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockFd < 0) {
        fprintf(stderr, "UDP echo client socket error.\n");
        return (-1);
    }
/*
 * Initialize the remote address structure
 */
    memset(&sockaddrinRemote, 0, sizeof(sockaddrinRemote));
/* Address translation error */
    if (!inet_aton(__UDP_ECHO_IP_SERVER, &sockaddrinRemote.sin_addr)) {

```

```

        close(sockFd);                /* Close the socket that has
been created */
        fprintf(stderr, "UDP echo client get addr error.\n");
        return (-1);                /* Error return */
    }

                                        /* Address structure size */
    sockaddrinRemote.sin_len = sizeof(struct sockaddr_in);
    sockaddrinRemote.sin_family = AF_INET; /* Address family */
                                        /* Bind server port */
    sockaddrinRemote.sin_port = htons(__UDP_ECHO_PORT_SERVER);
    for (;;) {
        fprintf(stdout, "Send Data: %s", pcSendData);
        sstRecv = strlen(pcSendData); /* Get the length of the send string
*/
        sstSend = sendto(sockFd,
                        (const void *)pcSendData,
                        sstRecv,
                        0,
                        (const struct sockaddr *)&sockaddrinRemote,
                        uiAddrLen); /* Send data to the specified server
*/
        if (sstSend <= 0) { /* Failed to send data */
            if ((errno != ETIMEDOUT ) &&
                (errno != EWOULDBLOCK)) { /* Non-timeout and non-blocking */
                close(sockFd); /* Close the socket that has been
created */

                fprintf(stderr, "UDP echo client sendto error.\n");
                return (-1); /* Error return */
            }
            continue; /* Rerun after timeout or
non-blocking */
        }
        memset(&cRecvBuff[0], 0, __UDP_ECHO_BUFF_SIZE_CLIENT);
        sstRecv = recvfrom(sockFd,
                        (void *)&cRecvBuff[0],
                        __UDP_ECHO_BUFF_SIZE_SERVER,
                        0,
                        (struct sockaddr *)&sockaddrinRemote,
                        &uiAddrLen); /* Receive data from the far end*/
        if (sstRecv <= 0) { /* Failed to receive data */
            if ((errno != ETIMEDOUT ) &&
                (errno != EWOULDBLOCK)) { /* Non-timeout and non-blocking */

```



```
        close(sockFd);                /* Close the socket that has
been created */
        fprintf(stderr, "UDP echo client recvfrom error.\n");
        return (-1);                  /* Error return */
    }
    continue;                          /* Rerun after timeout or
non-blocking */
}
fprintf(stdout, "Recv Data: ");
cRecvBuf[sstRecv] = 0;
fprintf(stdout, "%s\n", &cRecvBuf[0]);
sleep(5);                             /* Sleep for a while */
}
return (0);
}
```

15.3.2 TCP instance

The typical TCP client and server communication process is shown in Figure 15.23. The server is started first, the client is started sometime later, and it tries to connect to the server. We assume that the client sends a request to the server, the server processes the request, and sends a response back to the client. This process continues until the client closes the local connection, and sends an end notification to the server. After the server receives the end notification, it closes the server's local connection, and can either finish the operation or continue to wait for new client connection.

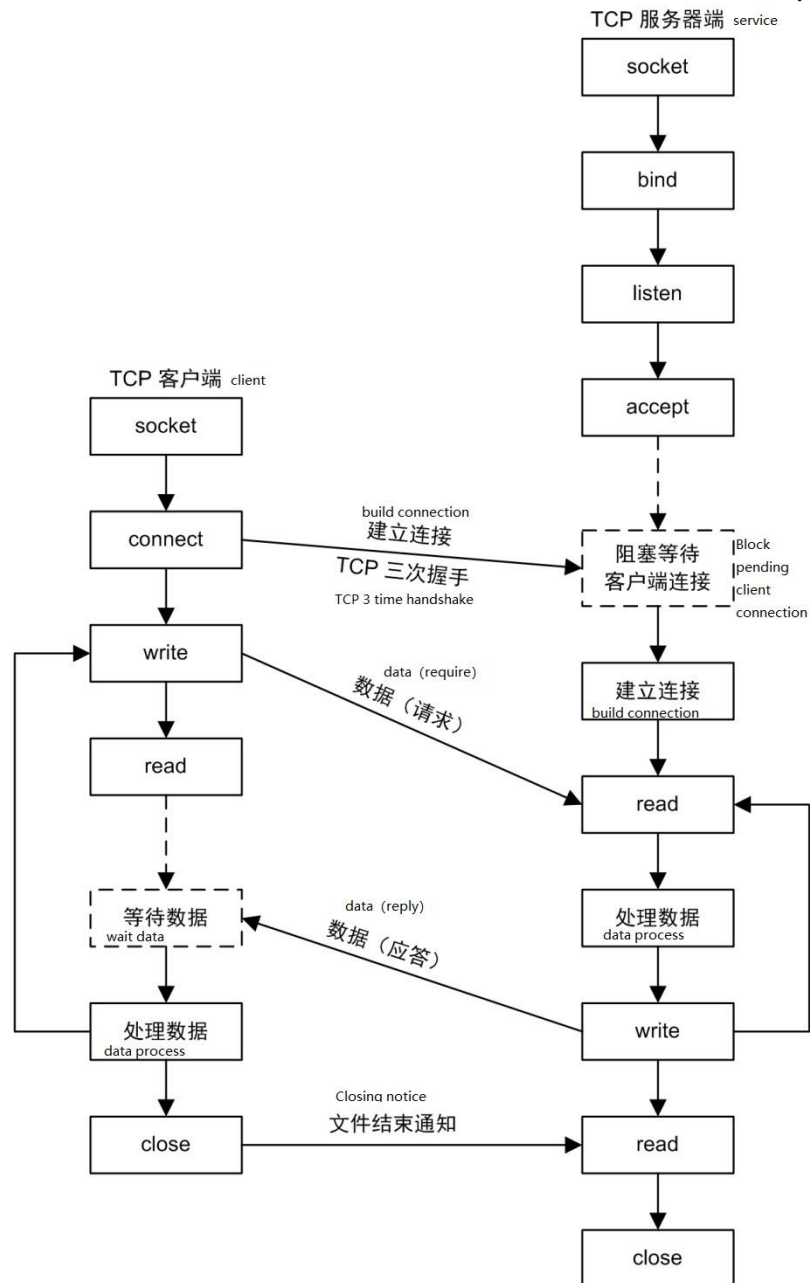


Figure 15.23 TCP Program socket function

The TCP client uses the connect function to establish connection with the TCP server.

```
#include <sys/socket.h>
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

Prototype analysis of Function connect:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter `s` is the socket (socket function returns);

- Parameter **name** is a pointer to the sockaddr structure type of the specific protocol domain;
- Parameter **namelen** represents the length of **name** structure.

The **name** (socket address structure) must contain the IP address and port number of the server. The TCP socket calls the connect function to stimulate three-way handshake of TCP, and only returns when the connection is established successfully or when an error occurs.

The listen function is only called by the TCP server, and indicates that a connection request to the socket can be accepted.

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

Prototype analysis of Function listen:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);
- Parameter **backlog** indicates the maximum number of connections which the corresponding socket can accept.

The accept function is only called by the TCP server to return a completed connection.

```
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Prototype analysis of Function accept:

- For success of the function, return the non-negative and connected socket descriptor. For failure, return -1 and set the error number;
- Parameter **s** is the socket (socket function returns);
- Parameter **addr** is used to return the protocol address structure information of the connected peer (client);
- Parameter **addrlen** is used to return the size of the connected protocol address structure.

We call the first **s** of accept as the listening socket (created by the socket, and then used as the first parameter to the bind function and the listen function), saying that its return value is the connected socket. It is important to distinguish the two sockets. A server usually only creates a listening socket. It exists for the lifetime of the server. The system creates a connected socket for each received client connection. (In other words, TCP's three-way handshake has been completed). When the server completes the service of a certain client, the corresponding connected socket is closed.

The getsockname function is used to return the local protocol address associated with a certain socket.

```
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

Prototype analysis of Function getsockname:

- For success of the function, return non-0. For failure, return -1;
- Parameter *s* is the socket (socket function returns);
- Parameter *name* is used to return the local protocol address structure information;
- Parameter *addrlen* is used to return the size of the local protocol address structure.

On a TCP client without calling the bind function, after the connect function returns successfully, the getsockname function is used to return the local IP address and the local port number of the connection.

When the bind function is called with the port number 0 (inform the system to select the local port number), the getsockname function returns the local port number specified by the system.

On a TCP server after calling the bind function with a wildcard IP address, once the connection with a client is established (accept returns successfully), the getsockname function can be used to return the local IP address of the connection. In such a call, the socket parameter must be the connected socket, but not the listening socket.

The getpeername function is used to return the remote protocol address associated with a certain socket.

```
#include <sys/socket.h>
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

Prototype analysis of Function getsockname:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *s* is the socket (socket function returns);
- Parameter *addr* is used to return the protocol address structure information of the connected peer (client);
- Parameter *addrlen* is used to return the size of the connected protocol address structure.

Summary of TCP communication: all clients and servers start from calling the socket. It returns a socket. The client calls the connect function. The server then calls the bind, listen and accept functions. The socket is usually closed by using the standard close function. However, you can also use the shutdown function to close the socket. Most TCP servers are concurrent. They serve each pending client connection individually, and most UDP servers are iterative.

The echo program model is shown in Figure 15.24 The client and the server follow this flow to complete reception and echo of the retroreflection data.

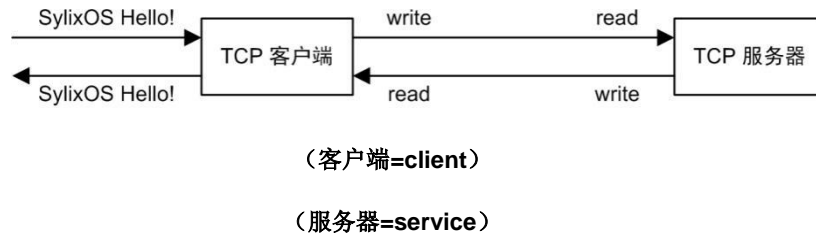


Figure 15.24 TCP echo program model

The client program sends "SylixOS Hello!" to the server, uses read to read back the server's echo, and finally outputs the received echo information "SylixOS Hello!".

The server program reads data from the client by using read, and sends the received data to the client program via write.

Program List 15.4 TCP echo server program

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

#define __TCP_ECHO_TYPE_CLIENT      1      /* Client mode */
#define __TCP_ECHO_TYPE_SERVER     2      /* Server mode */
#define __TCP_ECHO_TYPE            2      /* Current mode selection */
#define __TCP_ECHO_IP_CLIENT       "192.168.1.16" /* Client IP address */
#define __TCP_ECHO_IP_SERVER       "192.168.1.17" /* Server IP address */
#define __TCP_ECHO_PORT_CLIENT     8100    /* Client port number */
#define __TCP_ECHO_PORT_SERVER     8101    /* Server port number */
#define __TCP_ECHO_BUFF_SIZE_CLIENT 257    /* Client receive buffer size */
#define __TCP_ECHO_BUFF_SIZE_SERVER 257    /* Server receive buffer size */

static int __TcpEchoServer (void)
{
    int          iRet          = -1;
    int          sockFd        = -1;
    int          sockFdNew     = -1;
    socklen_t    uiAddrLen     = sizeof(struct sockaddr_in); /* Address structure size */
    register ssize_t sstRecv   = 0; /* Received data length */
    char         cRecvBuff[__TCP_ECHO_BUFF_SIZE_SERVER] = {0}; /* Receive buffer */
    struct sockaddr_in sockaddrinLocal; /* Local address */
```

```

    struct sockaddr_in  sockaddrinRemote;        /* Remote address
*/

    fprintf(stdout, "TCP echo server start.\n");

    sockFd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sockFd < 0) {
        fprintf(stderr, "TCP echo server socket error.\n");
        return (-1);
    }
    /*
    * Initialize the local address structure
    */
    memset(&sockaddrinLocal, 0, sizeof(sockaddrinLocal));
    sockaddrinLocal.sin_len          = sizeof(struct sockaddr_in);
                                         /* Address structure size */
    sockaddrinLocal.sin_family       = AF_INET; /* Address family */
    sockaddrinLocal.sin_addr.s_addr = INADDR_ANY;
                                         /*Bind server port */
    sockaddrinLocal.sin_port        = htons(__TCP_ECHO_PORT_SERVER);
    iRet = bind(sockFd,
                (struct sockaddr *)&sockaddrinLocal,
                sizeof(sockaddrinLocal)); /* Bind local address and port */
    if (iRet < 0) { /* Bind operation failed */
        close(sockFd); /* Close the socket that has been
created */
        fprintf(stderr, "TCP echo server bind error.\n");
        return (-1); /* Error return */
    }
    listen(sockFd, 2);
    sockFdNew = accept(sockFd, (struct sockaddr *)&sockaddrinRemote, &uiAddrLen);
    if (sockFdNew < 0) {
        close(sockFd); /* Close the socket that has been
created */
        fprintf(stderr, "TCP echo server accept error.\n");
        return (-1); /* Error return */
    }
    for (;;) {
                                         /* Clear the receive buffer */
        memset(&cRecvBuff[0], 0, __TCP_ECHO_BUFF_SIZE_SERVER);
                                         /* Receive data from the far end */
        sstRecv = read(sockFdNew,
                       (void *)&cRecvBuff[0],
                       __TCP_ECHO_BUFF_SIZE_SERVER);
    }

```

```

        if (sstRecv <= 0) {                                /* Failed to receive data
*/
        if ((errno != ETIMEDOUT ) &&
            (errno != EWOULDBLOCK)) {                    /* Non-timeout and non-blocking */
            close(sockFdNew);                             /* Shut down a connected socket
*/
            fprintf(stderr, "TCP echo server recvfrom error.\n");
            return (-1);                                  /* Return error */
        }
        continue;                                        /* Re-run after timeout or
non-blocking */
    }
                                                    /* Send retroreflective data back to
the far end */
    write(sockFdNew, (const void *)&cRecvBuff[0], sstRecv);
}
return (0);
}

```

Program List 15.5 TCP echo client program

```

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

#define __TCP_ECHO_TYPE_CLIENT    1    /* Client mode */
#define __TCP_ECHO_TYPE_SERVER    2    /* Server mode */
                                        /* Current mode selection */
#define __TCP_ECHO_TYPE
                                        /*Client IP address */
#define __TCP_ECHO_IP_CLIENT    "192.168.1.16"
                                        /* Server IP address */
#define __TCP_ECHO_IP_SERVER    "192.168.1.17"
#define __TCP_ECHO_PORT_CLIENT    8100 /* Client port number */
#define __TCP_ECHO_PORT_SERVER    8101 /* Server port number */
#define __TCP_ECHO_BUFF_SIZE_CLIENT    257 /* Client receive buffer size */
#define __TCP_ECHO_BUFF_SIZE_SERVER    257 /* Server receive buffer size */

static int __TcpEchoClient (void)
{
    int          iRet          = -1;    /* Operation result return value */
    int          sockFd        = -1;    /* socket Descriptor */
                                        /* Address structure size */
    socklen_t    uiAddrLen    = sizeof(struct sockaddr_in);
    register ssize_t    sstRecv    = 0; /* Received data length */
}

```

```

register ssize_t    sstSend    = 0;    /* Received data length */
                                /* String to send */
const char         *pcSendData = "SylixOS Hello!\n";
                                /* Receive buffer */
char               cRecvBuff[__TCP_ECHO_BUFF_SIZE_CLIENT] = {0};
struct sockaddr_in sockaddrinRemote; /* Remote address */

fprintf(stdout, "TCP echo client start.\n");

sockFd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sockFd < 0) {
    fprintf(stderr, "TCP echo client socket error.\n");
    return (-1);
}

/*
 * Initialize the remote address structure
 */
                                /* Clear address information */
memset(&sockaddrinRemote, 0, sizeof(sockaddrinRemote));
                                /* Address translation error */
if (!inet_aton(__TCP_ECHO_IP_SERVER, &sockaddrinRemote.sin_addr)) {
    close(sockFd); /* Close the socket that has been
created */
    fprintf(stderr, "TCP echo client get addr error.\n");
    return (-1); /* error return */
}
                                /* address structure size */
sockaddrinRemote.sin_len = sizeof(struct sockaddr_in);
sockaddrinRemote.sin_family = AF_INET; /* address family */
                                /* bind server port */
sockaddrinRemote.sin_port = htons(__TCP_ECHO_PORT_SERVER);
iRet = connect(sockFd,
               (const struct sockaddr *)&sockaddrinRemote,
               uiAddrLen);
if (iRet < 0) { /* operation failed */
    fprintf(stderr, "TCP echo client connect error.\n");
    return (-1); /* Error return */
}

for (;;) {
    fprintf(stdout, "Send Data: %s", pcSendData);
    sstRecv = strlen(pcSendData); /* Get the length of the send string
*/

```



```
sstSend = write(sockFd,
                (const void *)pcSendData,
                sstRecv);          /* Send data to the specified server
*/
if (sstSend <= 0) {                /* Failed to send data */
    if ((errno != ETIMEDOUT ) &&
        (errno != EWOULDBLOCK)) { /* Non-timeout and non-blocking
*/
        close(sockFd);            /* Close the socket that has been
created */
        fprintf(stderr, "TCP echo client write error.\n");
        return (-1);              /* Error return
*/
    }
    continue;                      /* Re-run after timeout or non-blocking
*/
}

/* Clear the receive buffer */
memset(&cRecvBuff[0], 0, __TCP_ECHO_BUFF_SIZE_CLIENT);
/* Receive data from the far end */
sstRecv = read(sockFd,
               (void *)&cRecvBuff[0],
               __TCP_ECHO_BUFF_SIZE_SERVER);
if (sstRecv <= 0) {                /* Failed to receive data */
    if ((errno != ETIMEDOUT ) &&
        (errno != EWOULDBLOCK)) { /* Non-timeout and non-blocking
*/
        close(sockFd);            /* Close the socket that has been
created */
        fprintf(stderr, "TCP echo client read error.\n");
        return (-1);              /*error return */
    }
    continue;                      /* Re-run after timeout or non-blocking
*/
}

fprintf(stdout, "Recv Data: ");
cRecvBuff[sstRecv] = 0;
fprintf(stdout, "%s\n", &cRecvBuff[0]);
sleep(5);                          /* Sleep for a while */
}
return (0);
}
```

15.3.3 Raw socket (RAW) instance

The raw socket can provide the following functions which are not normally provided by TCP and UDP sockets.

- Use the raw socket to read and write ICMPv4, IGMPv4 and IGMPv6 grouping. For example: *ping* command program;
- The special IPv4 datagram can be read and written with the raw socket. Recall the protocol field in Figure 15.15. Most kernels only process 1 (ICMP), 2 (IGMP), 6 (TCP) and 17 (UDP) datagrams, but the protocol field may also be other values. For example, the OSPF routing protocol does not use TCP or UDP, but uses IP directly, and sets the IP datagram protocol field to 89. Therefore, these datagrams contain protocol fields which are completely unknown to the kernel and shall be implemented by using the raw socket. These also apply to IPv6.

1. Creation of RAW socket

In order to create a raw socket the following steps are involved:

- When the second parameter of the socket function is SOCK_RAW, a raw socket is created. The third parameter is usually not 0. The following code is to create an IPv4 raw socket:

```
int sockfd;  
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

Note: the protocol parameter value is the IPPROTO_XXX constant value, such as IPPROTO_ICMP.

- The socket option can be set;
- The bind function can be called on the raw socket, but it is not common. This function is only used to set the local address, and the port number has no meaning for a raw socket;
- The connect function can be called on the raw socket, but it is not commonly used. The connect function sets only the destination address. For the output, after calling the connect function, we can call the write function or the send function instead of the sendto function because the destination address has been already specified.

2. Output of RAW socket

Usually the raw socket can be output by calling the sendto function or the sendmsg function and specifying the destination IP address. If the socket has already been connected by calling the connect function, you can also call the write function, writev function or send function.

3. Input of RAW socket

For raw socket input, you need to consider which IP packet received will be passed to the original socket. These need to comply with the following rules:

- The received TCP packet and UDP packet will never be passed to any raw socket. If you wish to read IP datagram containing TCP packet or UDP packet, they must be read at the link layer (see Section 15.2.6).
- When the kernel processes ICMP message, most of ICMP packets will be passed to the original socket;
- After the kernel processes the IGMP message, all IGMP packets will be passed to a raw socket;
- All IP datagrams with protocol fields that the kernel does not recognize will be passed to a raw socket;
- If the datagram arrives as a fragment, the packet will be passed to the original socket after all fragments arrive and are reassembled.

After the kernel prepares a datagram to be passed, the kernel will check all raw sockets to find all matching sockets. Each matching socket will receive a copy of the IP datagram. The datagram will only be sent to the specified socket when the following conditions are satisfied:

- If the specified protocol parameter is not zero when the original socket is created, the **protocol** field of the received datagram shall match this value. Otherwise, the datagram will not be sent to the socket;
- If a local IP address is bound to this raw socket, the destination IP address of the received datagram shall match the binding address; otherwise, the datagram will not be sent to the original socket;
- If this primitive socket specifies a peer's IP address by calling the connect function, the source IP address of the received datagram shall match the connected address. Otherwise, the datagram will not be sent to the original socket.

The following program uses the raw socket to send TCP network datagrams. IP header and TCP header of the datagram is constructed by us, and completed by the `ip_packet_ctor` function and the `tcp_packet_ctor` function respectively.

Program List 15.6 Use of raw socket

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <inet/lwip/ip4.h>
#include <inet/lwip/tcp_impl.h>

#define DEST_PROT      (4000)
#define PACKET_LEN    (128)
```

```
#define IPVERSION      (0x4)
#define TTLVAL        (255)

struct tcphdr {
    u16_t source;
    u16_t dest;
    u32_t seq;
    u32_t ack_seq;
    u16_t res1:4, doff:4, fin:1, syn:1, rst:1,
        psh:1, ack:1, urg:1, ece:1, cwr:1;
    u16_t window;
    u16_t check;
    u16_t urg_ptr;
};

void ip_packet_ctor (struct ip_hdr *iphdr, struct sockaddr_in *dest)
{
    int ip_len = sizeof(struct ip_hdr) + sizeof(struct tcphdr);

    IPH_VHL_SET(iphdr, IPVERSION, (sizeof(struct ip_hdr) >> 2));
    IPH_TOS_SET(iphdr, 0);
    IPH_LEN_SET(iphdr, htons(ip_len));
    IPH_ID_SET(iphdr, 0);
    IPH_OFFSET_SET(iphdr, 0);
    IPH_TTL_SET(iphdr, TTLVAL);
    IPH_PROTO_SET(iphdr, IPPROTO_TCP);
    IPH_CHKSUM_SET(iphdr, 0);
    iphdr->dest.addr = dest->sin_addr.s_addr;
}

void tcp_packet_ctor (struct tcphdr *tcphdr, u16_t srcprot, struct sockaddr_in
*dest)
{
    tcphdr->source = htons(srcprot);
    tcphdr->dest = dest->sin_port;
    tcphdr->seq = 3;
    tcphdr->ack_seq = 0;
    tcphdr->check = 0;
    tcphdr->doff = 5;
    tcphdr->syn = 1;
}

u16_t tcp_chksum (u16_t *addr, int len)
{
```

```
int      nleft = len;
int      sum   = 0;
u16_t   *temp  = addr;
short   ans   = 0;

while (nleft > 1) {
    sum += *temp++;
    nleft -= 2;
}

if (nleft == 1) {
    *(unsigned char *)&ans = *(unsigned short *)temp;
    sum += ans;
}

sum = (sum >> 16) + (sum & 0xffff);
sum += (sum >> 16);
ans = ~sum;

return (ans);
}

void send_packet (int sockfd, unsigned short srcport,
                 char *src, struct sockaddr_in *dest)
{
    struct ip_hdr   *iphdr;
    struct tcphdr   *tcphdr;
    char            buf[PACKET_LEN] = {0};
    struct in_addr  srcaddr;
    int             ip_len;

    ip_len = sizeof(struct ip_hdr) + sizeof(struct tcphdr);

    if (src) {
        inet_aton(src, &srcaddr);
    }

    iphdr = (struct ip_hdr *)buf;
    ip_packet_ctor(iphdr, dest);

    tcphdr = (struct tcphdr *) (buf + sizeof(struct ip_hdr));
    tcp_packet_ctor(tcphdr, srcport, dest);

    while (1) {
```

```
        iphdr->src.addr = (src == NULL) ? random() : srcaddr.s_addr;
        tcphdr->check = tcp_chksum((unsigned short *)tcphdr,
                                   sizeof(struct tcp_hdr));
        sendto(sockfd, buf, ip_len, 0, (struct sockaddr *)dest,
               sizeof(struct sockaddr_in));
    }
}

int main (int argc, char *argv[])
{
    int          sockfd;
    struct sockaddr_in  destaddr;

    if (argc < 2) {
        fprintf(stderr, "%s src-addr dest addr.\n", argv[0]);
        return (-1);
    }

    sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    if (sockfd < 0) {
        perror("socket");
        return (-1);
    }

    bzero(&destaddr, sizeof(destaddr));
    destaddr.sin_family      = AF_INET;
    destaddr.sin_len        = sizeof(destaddr);
    destaddr.sin_port       = htons(DEST_PROT);

    if (inet_aton(argv[2], &destaddr.sin_addr) == 0) {
        fprintf(stderr, "destination addr don't found.\n");
        return (-1);
    }

    send_packet(sockfd, DEST_PROT, argv[1], &destaddr);

    return (0);
}
```

15.4 Introduction to DNS

DNS is the abbreviation of Domain Name System. It is a core service of the Internet, and serves as a distributed database which can map domain names and IP addresses to each

other, enabling people to access the Internet more easily without remembering the IP address which can be read directly by the machine.

```
#include <sys/socket.h>
#include <netdb.h>
int  getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);
void freeaddrinfo(struct addrinfo *ai);
```

Prototype analysis of Function `getaddrinfo`:

- For success of the function, return 0. For failure, return non-0 value;
- Parameter ***nodename*** is the address character string;
- Parameter ***servname*** is the service name;
- Parameter ***hints*** enters address information;
- Output parameter ***res*** returns result address information.

函数 `freeaddrinfo` 原型分析:

Prototype analysis of Function `freeaddrinfo`:

- Parameter ***ai*** is the address information structure returned by the `getaddrinfo` function.

The `getaddrinfo` function returns one or more address information for the `addrinfo` structure. These address structures can be released by calling the `freeaddrinfo` function. The `addrinfo` structure in SylixOS is as shown below:

```
struct addrinfo {
    int          ai_flags;          /* Input flags.                */
    int          ai_family;        /* Address family of socket.    */
    int          ai_socktype;      /* Socket type.                 */
    int          ai_protocol;     /* Protocol of socket.         */
    socklen_t   ai_addrlen;       /* Length of socket address.    */
    struct sockaddr *ai_addr;     /* Socket address of socket.    */
    char        *ai_canonname;    /* Canonical name of service location. */
    struct addrinfo *ai_next;     /* Pointer to next in list.     */
};
```

- `ai_flags`: input flags, as shown in Table 15.8;
- `ai_family`: socket address family;
- `ai_socktype`: socket type, as shown in Table 15.2;
- `ai_protocol`: protocol, as shown in Table 15.1;
- `ai_addrlen`: length of the socket address;

- ai_addr: socket address;
- ai_canonname: canonical name.

Table 15.8 addrinfo structure input flag

Sign	Instructions
AI_PASSIVE	The socket address is used to listen for binding
AI_CANONNAME	Need a canonical name (as opposed to an alias)
AI_NUMERICHOST	Specify the host address numerically
AI_NUMERICSERV	Specify the service as the digital port number

An optional *hints* can be provided to select the address conforming to specific conditions. The *hints* is a template for filtering addresses, including ai_family, ai_flags, ai_protocol and ai_socktype fields. The remaining fields are divided into two cases in SylixOS. Firstly, if the program links to the external c library (libcextern), the remaining integer field must be 0, and the pointer field must be empty; secondly, if the application does not link the external c library, the remaining fields is not required.

```
#include <sys/socket.h>
#include <netdb.h>
int  getnameinfo(const struct sockaddr *addr, socklen_t len,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen, int flag);
```

Prototype analysis of Function getnameinfo:

- For success of the function, return 0. For failure, return non-0 error value;
- Parameter addr is the socket address;
- Parameter *len* is the length of socket address;
- Output parameter *host* returns the host name;
- Parameter *hostlen* is the length of parameter host buffer zone;
- Output parameter *serv* returns the service host name;
- Parameter *servlen* is the length of Parameter serv buffer zone;
- Parameter *flag* is the control flag, as shown in Table 15.10.

The getnameinfo function converts an address into a host name and service name. If *host* is non-NULL, it points to a buffer zone with a length of *hostlen* bytes to store the returned host name. Similarly, if *serv* is non-NULL, it points to a buffer zone with a length *servlen* bytes to store the returned service host name. In order to allocate space for *host* and *serv*, SylixOS contains the following constants:

Table 15.9 Constant values of character string length returned by the getnameinfo function

Constant value	Instructions	Value
NI_MAXHOST	Length of the returned host character string (parameter <i>hostlen</i>)	1025
NI_MAXSERV	Length of the returned service character string (parameter <i>servlen</i>)	32

Table 15.10 shows the configurable flag (parameter *flag*) which can change operation of the getnameinfo function.

Table 15.10 Flag of the getnameinfo function (flag)

Flag	Instructions
NI_NUMERICHOST	Return the numeric form of the host address, not the host name
NI_NUMERICSERV	Return the numeric form (port number) of the service address, not the name
NI_DGRAM	Service based on datagram but not stream
NI_NUMERICSERVICE	For IPv6, returns the numeric form of the range ID, not the name

DNS defines a message format for query and response. The overall format of this message is shown in Figure



16byte identification 16位标志	QR	opcode	AA	TC	RD	RA	zero	rcode
	1	4	1	1	1	1	3	4

Figure 15.25 DNS Format encapsulation

The 16-bit flag field is divided into several subfields:

- QR is 1 bit field: 0 indicates query message, and 1 indicates response message;
- Opcode is 4 bit field: usually 0 (standard query), other values are 1 (reverse query) and 2 (server status request);
- AA is a 1bit flag, which means "authoritative answer". The name server is authorized in this domain, and this bit only makes sense at respond;
- TC is 1 bit field indicating "truncated", which shows that the packet is longer than the allowable length. For example, when UDP is used, it means that when the total length of the response exceeds 512 bytes, only the first 512 bytes are returned;
- RD is a 1 bit field indicating "recursion desired". This bit can be set in a query and returned in the response. This flag tells the name server that this query must be processed, also known as a recursive query. If this bit is 0, and the requested name server does not have an authoritative answer, it returns a list of other name servers which can answer the query. This is called as iterative query. In the following example, we will see examples of these two types of queries;
- RA is 1bit field indicating "available recursion". If the name server supports recursive query, this bit is set as 1 in the response. Most name servers provide recursive query, except for some root servers;
- The 3bit field of zero must be 0;
- Rcode is a 4bit answer code field, as shown in Table 15.11.

Table 15.11 Answer code

Answer code	Instructions
0	No error
1	Message format error (the server cannot understand the requested error)
2	Server failure (the error cannot be handled for the reason of the server)
3	Name error (only meaningful for the authorized domain name resolution server, indicating that the resolved domain name does not exist)
4	Not realized
5	Deny (the server refused to respond because of the set policy)
6-15	Reserved value

In most queries, the query question segment contains questions, such as: specify the question. This section contains the "number of questions" question, and each question format is shown in Figure 15.26;

0	15	16	31
查询名			
查询类型		查询类	

(查询名=demand name)

(查询类=demand form)

Figure 15.26 Problem format

- The query name is encoded as some labels sequence. Each labels contains one byte to indicate the length of the subsequent character string, and the character string. 0 length and the empty character string indicate the end of the name;
- The query type is represented with 16 bit, the value can be any available type value, and the wildcard character represents all resource records;
- The query class is represented with 16 bit, as shown in Table 15.12.

Table 15.12 Query class

Class	Instructions
IN	Internet class
CSNET	CSNET class
CHAOS	CHAOS class
HESIOD	Designate MIT Athena Hesiod Class
ANY	Above wildcard character

The answer, authorization and additional information segments all share the same format: resource record, as shown in Figure 15.27.

0	15	16	31
域名 domain name			
类型 type		类 form	
生存时间 survival time			
资源数据长度 Resource data length			

Figure 15.27 Resource record format

- The domain name is the domain name contained in the resource record;
- Type represents 16bit resource record type;
- Class represents 16bit resource record class;
- The lifetime indicates the time that the resource record can be cached. If it is 0, it can only be transmitted and cannot be cached.
- The resource data length indicates the data length.

The DNS query process is as follows:

(1) The client sends the domain name resolution request, and sends the request to the local domain name server;

(2) When the local domain name server receives the request, it first queries the local cache. If there is such record, the local domain name server will directly return the query results;

(3) If there is no such record in the local cache, the local domain name server directly sends the request to the root name server, and queries its own cache. If there is no such record, it returns the address of the related lower-level domain name server;

(4) Repeat Step 3 until you find the correct record.

The following program shows how to use the `getaddrinfo` function. This program only implements the address information of those protocols that work with IPv4 (`ai_family` is `AF_INET`). The program limits the output to the `AF_INET` protocol family, that is to say, the `ai_family` field is set in the prompt.

Program List 15.7 Use of the `getaddrinfo` function

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

void family (struct addrinfo *ai)
{
    fprintf(stdout, "-----show family-----\n");

    switch (ai->ai_family) {

    case AF_INET:
        fprintf(stdout, "inet.\n");
        break;

    case AF_INET6:
        fprintf(stdout, "inet6.\n");
        break;

    case AF_UNIX:
        fprintf(stdout, "unix domain.\n");
        break;

    case AF_PACKET:
        fprintf(stdout, "packet domain.\n");
        break;

    case AF_UNSPEC:
        fprintf(stdout, "unspec.\n");
        break;

    default:
        fprintf(stderr, "unknown %d\n", ai->ai_family);
    }
}
```

```
void type (struct addrinfo *ai)
{
    fprintf(stdout, "-----show socktype-----\n");

    switch (ai->ai_socktype) {

    case SOCK_STREAM:
        fprintf(stdout, "stream.\n");
        break;

    case SOCK_DGRAM:
        fprintf(stdout, "datagram.\n");
        break;

    case SOCK_RAW:
        fprintf(stdout, "raw.\n");
        break;

    case SOCK_SEQPACKET:
        fprintf(stdout, "seqpacket.\n");
        break;

    default:
        fprintf(stderr, "unknown %d.\n", ai->ai_socktype);
    }
}

void protocol (struct addrinfo *ai)
{
    fprintf(stdout, "-----show protocol-----\n");

    switch (ai->ai_protocol) {

    case 0:
        fprintf(stdout, "default.\n");
        break;

    case IPPROTO_TCP:
        fprintf(stdout, "TCP.\n");
        break;

    case IPPROTO_UDP:
        fprintf(stdout, "UDP.\n");
        break;
    }
```

```
case IPPROTO_RAW:
    fprintf(stdout, "RAW.\n");
    break;

default:
    fprintf(stderr, "unknown %d.\n", ai->ai_protocol);
}
}

void flags (struct addrinfo *ai)
{
    fprintf(stdout, "-----show flags-----\n");

    if (ai->ai_flags == 0) {
        fprintf(stdout, " 0 \n");
    } else {
        if (ai->ai_flags & AI_PASSIVE) {
            fprintf(stdout, " AI_PASSIVE \n");
        }
        if (ai->ai_flags & AI_CANONNAME) {
            fprintf(stdout, " AI_CANONNAME \n");
        }
        if (ai->ai_flags & AI_NUMERICHOST) {
            fprintf(stdout, " AI_NUMERICHOST \n");
        }
        if (ai->ai_flags & AI_NUMERICSERV) {
            fprintf(stdout, " AI_NUMERICSERV \n");
        }
    }
}

int main (int argc, char *argv[])
{
    struct addrinfo      hints;
    struct addrinfo      *aip, *ailist;
    int                  ret;
    struct sockaddr_in   *inetaddr;
    const char           *addr = NULL;
    char                  buf[INET_ADDRSTRLEN];

    if (argc < 2) {
        fprintf(stderr, "%s [host]\n", argv[0]);
        return (-1);
    }
}
```

```
    }

    bzero(&hints, sizeof(hints));

    hints.ai_family      = AF_INET;
    hints.ai_socktype    = 0;
    hints.ai_flags       = AI_CANONNAME;

    ret = getaddrinfo(argv[1], NULL, &hints, &ailist);
    if (ret != 0) {
        fprintf(stderr, "getaddrinfo %s.\n", strerror(ret));
        return (-1);
    }

    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        family(aip);
        type(aip);
        protocol(aip);
        flags(aip);
        fprintf(stdout, "host: %s\n",
                aip->ai_canonname ? aip->ai_canonname : "=");
        if (aip->ai_family == AF_INET) {
            inetaddr = (struct sockaddr_in *)aip->ai_addr;
            addr = inet_ntop(AF_INET, &inetaddr->sin_addr, buf, INET_ADDRSTRLEN);
            fprintf(stdout, "ADDR[IPv4]: %s\n", addr ? addr : "NULL");
        }
    }

    freeaddrinfo(ailist);

    return (0);
}
```

Run the program under the SylixOS Shell, and the results are as displayed:

```
# ./getaddrinfo sylixos.com
-----show family-----
inet.
-----show socktype-----
unknown 0.
-----show protocol-----
default.
-----show flags-----
0
host: sylixos.com
```



```
ADDR[IPv4]: 106.39.47.146
```

15.5 AF_UNIX domain protocol

UNIX domain socket is an advanced IPC mechanism. IPC in the form can communicate between two processes running in the same computer system. Although Internet domain socket can be used for the same purpose, UNIX domain socket is more efficient. UNIX domain socket merely copies data, but does not perform protocol processing. Therefore, there is no need to add or delete network header, calculate the checksum, generate the serial number, or send the confirmation message.

UNIX domain socket in SylixOS provides three interfaces: stream (SOCK_STREAM), datagram (SOCK_DGRAM) and continuous datagram (SOCK_SEQPACKET). The UNIX domain datagram service is reliable, free of message loss or passing error. UNIX domain socket is like a mix of sockets and pipes. They can use their network-facing domain socket interfaces or use the socketpair function to create a pair of unnamed and interconnected UNIX domain sockets.

```
#include <sys/socket.h>
int socketpair(int domain, int type, int protocol, int sv[2]);
```

Prototype analysis of Function socketpair:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***domain*** is the protocol domain (only supports AF_UNIX);
- Parameter ***type*** is the protocol type;
- Parameter ***protocol*** is the protocol;
- Output parameter ***sv[2]*** returns the file descriptor group.

Although the interface is general enough, the function only supports the UNIX domain in SylixOS. A pair of interconnected UNIX domain sockets can play the role of full-duplex pipe: both ends are open for reading and writing.

We will find that the socket created by the socketpair function is unnamed, which means that the unrelated process cannot use them.

The Internet domain socket can bind an address to a socket by calling the bind function, and can bind an address to the UNIX domain socket. The difference is that the address used by the UNIX domain socket is different from that of the Internet socket.

In Section 15.1.2 we introduced that the address structure of the UNIX domain socket is sockaddr_un, and the sun_path member of the structure contains a path name. When we bind an address to the UNIX domain socket, the system will create a file of S_IFSOCK type with the path name.

This file is only used to notify the socket name to the client process, which cannot be opened or used by the application for communication.

If the file has existed when we try to bind the same address, the bind request will fail. The file is not automatically closed when the socket is closed. Therefore, it must be guaranteed that the file is unlinked before the application exits.

When both parties to communications are on the same host, use of UNIX domain sockets is usually twice as fast as TCP sockets. The UNIX domain socket can be used to pass the descriptor between two processes on the same host. The UNIX domain socket can provide the client's voucher to the server, which can provide additional security check.

The following shall be noticed when the UNIX domain socket is used:

- The path name used by the connect function must be a path name bound to an open UNIX domain socket, and the socket type must also be consistent;
- The UNIX domain stream socket is similar to the TCP socket, and both provide a byte stream interface without record boundary for the process.
- If the connect function call of the UNIX domain byte stream socket finds that the queue of the listening socket is full, an ECONNREFUSED error code will be returned immediately. This is different from TCP: if the queue of the listening socket is full, it will ignore the incoming SYN, and the initiator of the TCP connection will send several SYN retries.
- The UNIX domain datagram socket is similar to the UDP socket, and both provide an unreliable data service that retains record boundaries;
- The SylixOS UNIX domain socket implements the SOCK_SEQPACKET datagram, and this type guarantees the two-way function of connectivity and of record boundary retention.
- Unlike UDP, sending the datagram on an unbound UNIX domain socket will not bundle it with a path name (sending data on an unbound UDP socket will bundle a temporary port for the socket). It means that the receiver cannot send back the response datagram unless the sender of the datagram bundles a path name. Unlike TCP and UDP, calling the connect function on the UNIX domain datagram socket will not bundle a path name.

15.5.1 AF_UNIX instance

1. SOCK_STREAM type instance

The UNIX domain socket communication process of SOCK_STREAM type is as shown in Figure 15.23. The following program uses the UNIX domain socket to communicate between the server and the client. The server side waits for the client to send the character string

"client", the server side sends "ACK" to respond the client after the character string "client" is received successfully, and the client prints response results of the server side.

Program List 15.8 STREAM type client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

#define AF_UNIX_FILE "afunix.tmp"

int main (int argc, char *argv[])
{
    int          sockfd;
    struct sockaddr_un  addr;
    socklen_t    len;
    int          i, bytes, result;
    char         str[16];

    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sockfd < 0) {
        fprintf(stderr, "[client]socket error.\n");
        return (-1);
    }

    bzero(&addr, sizeof(addr));
    len = sizeof(addr);
    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, AF_UNIX_FILE);

    result = connect(sockfd, (struct sockaddr *)&addr, len);
    if (result < 0) {
        fprintf(stderr, "[client]connect error.\n");
        close(sockfd);
        return (-1);
    }
    fprintf(stdout, "[client]connect server success.\n");

    for (i = 0; i < 5; i++) {
        bytes = write(sockfd, "client", 7);
        if (bytes < 0) {
            fprintf(stderr, "[client]write error.\n");
            break;;
        }
    }
}
```

```
    }
    sleep(2);

    bytes = read(sockfd, str, 16);
    if (bytes < 0) {
        fprintf(stderr, "[client]read error.\n");
        break;
    }
    fprintf(stdout, "[client]receive ACK from server.\n");
}

close(sockfd);
return (0);
}
```

Program List 15.9 STREAM type server side

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <string.h>

#define AF_UNIX_FILE "afunix.tmp"

int main (int argc, char *argv[])
{
    int          ssockfd, csockfd;
    socklen_t    slen, clen;
    struct sockaddr_un saddr;
    int          i, bytes;
    char         str[16];

    unlink(AF_UNIX_FILE);

    ssockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (ssockfd < 0) {
        fprintf(stderr, "[server]socket error.\n");
        return (-1);
    }

    slen = sizeof(saddr);
    strcpy(saddr.sun_path, AF_UNIX_FILE);
    saddr.sun_family = AF_UNIX;
```

```
bind(ssockfd, (struct sockaddr *)&saddr, slen);
listen(ssockfd, 5);
fprintf(stdout, "[server]waiting for client connect...\n");

csockfd = accept(ssockfd, (struct sockaddr *)&saddr, &clen);
if (csockfd < 0) {
    fprintf(stderr, "[server]accept error.\n");
    close(ssockfd);
    return (-1);
}
fprintf(stdout, "[server]connect success.\n");

for (i = 0; i < 5; i++) {
    bytes = read(csockfd, str, 16);
    if (bytes < 0) {
        fprintf(stderr, "[server]read error.\n");
        break;
    }

    if (strncmp("client", str, 6) == 0) {
        fprintf(stdout, "[server]receiver from client is: %s\n", str);
    } else {
        fprintf(stderr, "[server]client send failed.\n");
        break;
    }
    sleep(1);

    fprintf(stdout, "[server]server reply ACK.\n");
    bytes = write(csockfd, "ACK", 4);
    if (bytes < 0) {
        fprintf(stderr, "[server]write error.\n");
        break;
    }
}

unlink(AF_UNIX_FILE);
close(csockfd);
close(ssockfd);

return (0);
}
```

2. SOCK_DGRAM type instance

The communication process of the UNIX domain socket of SOCK_DGRAM type is similar to that of UDP, as shown in Figure 15.21. The following is the server-client communication instance implemented with the SOCK_DGRAM type. Similar to the function of SOCK_STREAM type, the server side passively receives data sent from the client. If the server side receives Character "q", communication on request from the client is terminated, which is the termination program of the server side. The client sends one-time local current time every 1 second, and terminates the communication process after sending 5 times.

Program List 15.10 DGRAM type client

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stddef.h>
#include <time.h>

#define AF_UNIX_FILE "afunix.tmp"

int main (int argc, char **argv)
{
    int                sockfd;
    struct sockaddr_un  addr;
    int                count = 0;
    socklen_t          len;

    sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        fprintf(stderr, "[client]socket error.\n");
        return (-1);
    }

    bzero(&addr, sizeof(addr));

    len                = sizeof(addr);
    addr.sun_family    = AF_UNIX;
    strcpy(addr.sun_path, AF_UNIX_FILE);

    for (;;) {
        time_t          t;
        char             *str;
        ssize_t          ret;
```

```

        t = time(NULL);
    str = ctime(&t);
    if (str == NULL) {
        break;
    }

    if (count++ > 5) {
        str = "q";
    }

    ret = sendto(sockfd, str, strlen(str), 0, (struct sockaddr *)&addr, len);
    if (ret < 0) {
        fprintf(stderr, "[server]sendto error.\n");
        break;
    }
    fprintf(stdout, "[client]send %s", str);

    if (count > 6) {
        break;
    }

    sleep(1);
}

unlink(AF_UNIX_FILE);
close(sockfd);
return (0);
}

```

Program List 15.11 DGRAM type server side

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stddef.h>

#define AF_UNIX_FILE "afunix.tmp"

int main (int argc, char **argv)
{
    int                sockfd;
    struct sockaddr_un addr;

```

```
int          ret;
socklen_t    len;
size_t       size;
char         buf[BUFSIZ] = {0};

unlink(AF_UNIX_FILE);

sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
if (sockfd < 0) {
    fprintf(stderr, "[server]socket error.\n");
    return (-1);
}

bzero(&addr, sizeof(addr));
len          = sizeof(addr);
addr.sun_family = AF_UNIX;
strcpy(addr.sun_path, AF_UNIX_FILE);

ret = bind(sockfd, (struct sockaddr *)&addr, len);
if (ret < 0) {
    fprintf(stderr, "[server]bind error.\n");
    close(sockfd);
    return (-1);
}

for (;;) {
    size = recvfrom(sockfd, buf, BUFSIZ, 0, NULL, NULL);
    if (size > 0) {
        fprintf(stdout, "[server]recv: %s", buf);
    }

    if (strncmp("q", buf, 1) == 0) {
        break;
    }
}

unlink(AF_UNIX_FILE);
close(sockfd);
return (0);
}
```

3. SOCK_SEQPACKET type instance

The UNIX domain socket of SOCK_SEQPACKET type is a connection-oriented block message transfer. Therefore, the communication process is similar to that of the socket of SOCK_STREAM type.

Program List 15.12 SEQPACKET type client

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>

#define UNIX_PATH  "./AF_UNIX_SEQ"
#define SEND_STR   "af_unix seqpacket test."
#define SEND_Q     "q"

int main (int argc, char *argv[])
{
    int          sockfd;
    int          ret;
    struct sockaddr_un  unixaddr;
    socklen_t    len = sizeof(unixaddr);
    int          i;

    unlink(UNIX_PATH);
    sockfd = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (sockfd < 0) {
        perror("socket");
        return (-1);
    }

    unixaddr.sun_family = AF_UNIX;
    strcpy(unixaddr.sun_path, UNIX_PATH);
    unixaddr.sun_len    = SUN_LEN(&unixaddr);

    ret = connect(sockfd, (struct sockaddr *)&unixaddr, len);
    if (ret < 0) {
        perror("connect");
        return (-1);
    }

    for (i = 0; i < 10; i++) {
        sendto(sockfd, SEND_STR, strlen(SEND_STR), 0,
              (struct sockaddr *)&unixaddr, len);
        fprintf(stdout, "send msg: %s\n", SEND_STR);
        sleep(1);
    }
}
```

```
    }

    sendto(sockfd, SEND_Q, strlen(SEND_Q), 0,
           (struct sockaddr *)&unixaddr, len);

    return (0);
}
```

Program List 15.13 SEQPACKET type server side

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>

#define UNIX_PATH  "./AF_UNIX_SEQ"
#define QUIT_STR   "q"

int main (int argc, char *argv[])
{
    int          sockfd, csockfd;
    int          ret;
    char         buf[1024] = {0};
    struct sockaddr_un  unixaddr;
    socklen_t    len = sizeof(unixaddr);

    unlink(UNIX_PATH);

    sockfd = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (sockfd < 0) {
        perror("socket");
        return (-1);
    }

    unixaddr.sun_family = AF_UNIX;
    strcpy(unixaddr.sun_path, UNIX_PATH);
    unixaddr.sun_len    = SUN_LEN(&unixaddr);

    ret = bind(sockfd, (struct sockaddr *)&unixaddr, len);
    if (ret < 0) {
        perror("bind");
        return (-1);
    }
    listen(sockfd, 5);

    csockfd = accept(sockfd, (struct sockaddr *)&unixaddr, &len);
```

```
    if (csockfd < 0) {
        perror("accept");
        return (-1);
    }

    while (1) {
        ssize_t stlen;

        stlen = recvfrom(csockfd, buf, sizeof(buf), 0, NULL, NULL);
        if (stlen < 0) {
            continue;
        }

        if (!strncmp(buf, QUIT_STR, 1)) {
            fprintf(stdout, "server exit.\n");
            break;
        }

        fprintf(stdout, "buf: %s\n", buf);
    }

    return (0);
}
```

15.6 AF_PACKET link layer communication

Most operating systems provide applications with access to the data link layer at present, and the application program can access the link layer to monitor the packet received on the link layer. Therefore, we can monitor the network on the ordinary computer system via the program like tcpdump without the special hardware device. If the promiscuous mode of the network interface is used, we can even listen to all packets on the local cable, not just the packet with the host where the program is running as the destination address.

The socket of PACKET type shall be created when the data link layer packet is read in SylixOS, and the PACKET socket is used to send and receive the data frame on the link layer. Therefore, the application can complete implementation of each layer above the link layer in the user space. The PACKET socket is similar to TCP, UDP and UNIX in the definition mode:

```
int sockfd;

sockfd = socket(AF_PACKET, type, protocol);
```

PACKET socket definition needs to specify the socket function parameter **domain** as AF_PACKET (PF_PACKET), Parameter type supports SOCK_DGRAM and SOCK_RAW,

Parameter ***protocol*** contains the link layer protocol, and part of the commonly used protocols are shown in Table 15.13 (the more protocols are defined in the file <net/if_ether.h>).

Table 15.13 PACKET socket protocol

Agreement	Instructions	Value
ETH_P_IP	IP type data frame	0x0800
ETH_P_ARP	ARP type data frame	0x0806
ETH_P_RARP	RARP type data frame	0x8035
ETH_P_ALL	All types of data frames	0x0003

Specify Protocol ETH_P_XXX informs the data link layer to pass the different types of frames it receives to the PACKET socket. If the data link supports promiscuous mode (such as Ethernet), the promiscuous mode of the network device shall be set. The flag is got by calling the ioctl function (command SIOCGIFFLAGS), and the IFF_PROMISC flag is set, then the ioctl function (command SIOCSIFFLAGS) is called again to set a new flag (containing the IFF_PROMISC flag).

We mentioned above that the parameter ***type*** supports SOCK_RAW type, which contains the original packet of the link layer header information, i.e., a MAC header of ethhdr structure type shall be added for the type of socket when sent, and the structure is defined as follows:

```
struct ethhdr {
    u_char    h_dest[ETH_ALEN];    /* destination eth addr    */
    u_char    h_source[ETH_ALEN];  /* source ether addr      */
    u_short   h_proto;             /* packet type ID field   */
} __attribute__((packed));
```

- h_dest: destination MAC address of Ethernet;
- h_source: source MAC address of Ethernet;
- h_proto: frame type, as shown in 15.13;

The SOCK_DGRAM type has processed the header information of the link layer, i.e., the Ethernet header of the received data frame has been removed, and the application shall not add the header information when sending such data.

For created socket, data can be received and sent by calling the `recvfrom` function. Unlike UDP, the address structure of PACKET is the `sockaddr_ll` type, and the structure is defined as follows:

```
struct sockaddr_ll {
    u_char    sll_len;           /* Total length of sockaddr    */
    u_char    sll_family;       /* AF_PACKET                   */
    u_short   sll_protocol;     /* Physical layer protocol     */
    int       sll_ifindex;      /* Interface number            */
    u_short   sll_hatype;       /* ARP hardware type           */
    u_char    sll_pkttype;      /* packet type                  */
    u_char    sll_halen;        /* Length of address           */
    u_char    sll_addr[8];      /* Physical layer address      */
};
```

- `sll_len`: address structure length;
- `sll_family`: protocol family (`AF_PACKET`);
- `sll_protocol`: link layer protocol type, as shown in 15.13;
- `sll_ifindex`: network interface index number (such as 1 in `en1`);
- `sll_hatype`: device protocol type (For example, Ethernet is `ARPHRD_ETHER`);
- `sll_pkttype`: packet type, as shown in Table 15.14;
- `sll_halen`: physical address length (MAC address length);
- `sll_addr`: physical address.

The packet types supported by SylixOS are listed in the following table. It shall be noted that these types are only meaningful for the received packets.

Table 15.14 Packet type

Packet type	Instructions
<code>PACKET_HOST</code>	The destination address is the local host packet
<code>PACKET_BROADCAST</code>	Broadcast packet of the physical layer
<code>PACKET_MULTICAST</code>	A packet sent to the multicast address of the physical layer
<code>PACKET_OTHERHOST</code>	Packets sent to other hosts in promiscuous mode
<code>PACKET_OUTGOING</code>	Loopback packet

15.6.1 AF_PACKET instance

It is introduced above that the `AF_PACKET` protocol family supports `SOCK_DGRAM` and `SOCK_RAW` types. The former allows the kernel to process addition or removal of the Ethernet packet header, while the latter allows the application to have complete control to the Ethernet header. The protocol type must conform to one of the types defined in the header file

<net/if_ether.h> during socket calling (For example: ETH_P_IP in the case), and ETH_P_IP is used to process a set of protocols (TCP, UDP, ICMP and so on).

The instance program provides the method for AF_PACKET to read the raw data of the link layer (SOCK_RAW), which is similar to sniffer in function. The program calls the socket function to establish the socket firstly (the protocol field is AF_PACKET, and the protocol type is SOCK_RAW). Since the IP layer datagram shall be processed, the *protocol* is appointed as ETH_P_IP. Then the program calls the recvfrom function to start receiving the network packet. After successfully receiving the network packet, it firstly determines whether it is a complete packet (the length cannot be less than 42). If it is a complete packet, the MAC address and network address (IP address) shall be printed. Otherwise, discard the packet and exit the program.

Program List 15.14 AF_PACKET instance

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if_ether.h>

#define IPV4_VERSION    (0x4)
#define BUFSIZE        (2048)

int main (int argc, char *argv[])
{
    int          sockfd;
    struct ethhdr *ethheader;
    unsigned char *ip4header;

    sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
    if (sockfd < 0) {
        perror("socket");
        return (-1);
    }

    while (1) {
        ssize_t      len;
        unsigned char buf[BUFSIZE];

        fprintf(stdout, ".....\n");

        len = recvfrom(sockfd, (void *)buf, sizeof(buf), 0, NULL, NULL);
        if (len < 0) {
```

```
        continue;
    }

    fprintf(stdout, "recv %ld bytes.\n", len);

    /*
     * Check whether the packet contains the complete Ethernet header (14), IP
header (20), TCP/UDP header (20/8)
     */
    if (len < 42) {
        perror("recvfrom: ");
        fprintf(stderr, "Incomplete packet [%s]\n", strerror(errno));
        break;
    }

    /*
     * Print Ethernet header information.
     */
    ethheader = (struct ethhdr *)buf;
    fprintf(stdout, "Ethernet type: 0x%x\n", ntohs(ethheader->h_proto));
    fprintf(stdout, "Source MAC addr: "
            "%02x:%02x:%02x:%02x:%02x:%02x\n",
            ethheader->h_source[0], (ethheader->h_source[1]),
            (ethheader->h_source[2]), ethheader->h_source[3],
            (ethheader->h_source[4]), (ethheader->h_source[5]));
    fprintf(stdout, "Destination MAC addr: "
            "%02x:%02x:%02x:%02x:%02x:%02x\n",
            ethheader->h_dest[0], ethheader->h_dest[1],
            ethheader->h_dest[2], ethheader->h_dest[3],
            ethheader->h_dest[4], ethheader->h_dest[5]);

    /*
     * Print IP header information.
     */
    ip4header = buf + sizeof(struct ethhdr);
    if ((*ip4header) & IPV4_VERSION) == IPV4_VERSION) {
        fprintf(stdout, "Source host %d.%d.%d.%d\n",
                ip4header[12], ip4header[13],
                ip4header[14], ip4header[15]);

        fprintf(stdout, "Destination host %d.%d.%d.%d\n",
                ip4header[16], ip4header[17],
                ip4header[18], ip4header[19]);
    }
}
```

```

        fprintf(stdout, "Source, Dest ports %d,%d\n",
                (ip4header[20] << 8) + ip4header[21],
                (ip4header[22] << 8) + ip4header[23]);

        fprintf(stdout, "Layer[4] protocol %d\n", ip4header[9]);
    }
}

close(sockfd);

return (0);
}

```

Run the program in SylxOS Shell, and partial results are displayed as follows:

```

# ./packet_test
.....
recv 78 bytes.
Ethernet type: 0x800
Source MAC addr: 00:ff:ff:6f:a7:a0
Destination MAC addr: 08:08:3e:26:0a:5a
Source host 192.168.7.40
Destination host 192.168.7.30
Source, Dest ports 2048,1386
Layer[4] protocol 1
.....

```

It can be seen from the program running results that the Ethernet type is 0x800 which is the IP protocol type, and the MAC address and network address of the sender and the receiver can be clearly seen. The value of protocol is 1, and it can be concluded that the network packet is ICMP (1) packet. From more experimental results, the program only receives the packet with the destination address of 192.168.7.30, i.e., only receives the network packet sent to the host. Actually, the socket option can be set at the promiscuous mode (packets not sent to the machine can be received) to receive more packets.

15.6.2 AF_PACKET and mmap

The PACKET socket transmission mode introduced above is in the form of the buffer zone, and requires a function call for every packet captured, causing decline in transmission efficiency. For example, the function (such as libpcap) shall be called twice to get the timestamp of PACKET.

The PACKET MMAP mechanism solves this problem of low transmission efficiency. The PACKET MMAP mechanism will allocate a kernel buffer in the kernel space, then the user maps the buffer zone to user space by calling the mmap function, the kernel copies the

received packet to the kernel buffer zone, and the application can directly access the data in the buffer.

The PACKET MMAP mechanism provides a ring buffer zone which is mapped to the user space with configurable size. The size of the buffer zone is got from the member values in the `tpacket_req` structure, and the structure is defined as follows:

```
struct tpacket_req {
    u_int      tp_block_size;          /* Min size of contiguous block */
    u_int      tp_block_nr;           /* Number of blocks */
    u_int      tp_frame_size;         /* Size of frame */
    u_int      tp_frame_nr;           /* Total number of frames */
};
```

- `tp_block_size`: block size;
- `tp_block_nr`: block number;
- `tp_frame_size`: frame size;
- `tp_frame_nr`: frame number.

This ring buffer zone consists of `tp_block_nr` blocks, and each block contains `tp_block_size/tp_frame_size` frames. Where, each frame must be in the same block. The block size must be SylixOS page-aligned (the value obtained by the `getpagesize` function is 4K at default in SylixOS), and the frame size must be `TPACKET_ALIGNMENT == 16` (defined in `<netpacket/packet.h>`) byte alignment. It shall be noted that `tp_frame_nr` must be the same with $(tp_block_size/tp_frame_size)*tp_block_nr$. All blocks in SylixOS constitutes a contiguous physical memory. This structure relation is shown in Figure 15.28.

Each frame header of the ring buffer zone contains a `tpacket_hdr` structure, storing some information. The structure is divided into two versions of implementation, as shown below:

```
enum tpacket_versions {
    TPACKET_V1,
    TPACKET_V2
};
```

The following shows implementation of `TPACKET_V1` version, and the meanings are as follows:

```
struct tpacket_hdr {
    volatile u_long    tp_status;
    volatile u_int     tp_len;
    volatile u_int     tp_snaplen;
    volatile u_short   tp_mac;
    volatile u_short   tp_net;
    volatile u_int     tp_sec;
    volatile u_int     tp_usec;
};
```

- tp_status: frame status, as shown in Table 15.15:
- Tp_len: packet length (in case of the SOCK_DGRAM kernel, the length of the MAC header will be subtracted);
- Tp_snaplen: valid data length;
- Tp_mac: Ethernet frame offset position;
- Tp_net: NET datagram offset position;
- Tp_sec: timestamp (seconds);
- Tv_usec: timestamp (microseconds).

Table 15.15 frame state

frame state	Instructions
TP_STATUS_KERNEL	Indicate that the kernel can use this frame, that is to say, the application has no data to read
TP_STATUS_USER	Indicate that the application is readable. The kernel cannot use this frame at this time

The following shows implementation of the TPACKET_V2 version. The meanings are as follows:

```

struct tpacket2_hdr {
    volatile u_int32_t  tp_status;
    volatile u_int32_t  tp_len;
    volatile u_int32_t  tp_snaplen;
    volatile u_int16_t  tp_mac;
    volatile u_int16_t  tp_net;
    volatile u_int32_t  tp_sec;
    volatile u_int32_t  tp_nsec;
    volatile u_int16_t  tp_vlan_tci;
    volatile u_int16_t  tp_vlan_tpid;
};

```

- tp_status: frame status;
- Tp_len: packet length (in case of the SOCK_DGRAM kernel, the length of the MAC header will be subtracted);
- Tp_snaplen: valid data length;
- Tp_mac: Ethernet frame offset position;
- Tp_net: NET datagram offset position;
- Tp_sec: timestamp (second);

- Tp_nsec: timestamp (nanosecond);
- Tp_vlan_tci: two-byte tag control information (TCI) in the vlan;
- Tp_vlan_tpid: two-byte tag protocol identifier (TPID) in the vlan

Figure 15.28 shows the correspondence between block and frame implemented by SylixOS as well as the frame structure.

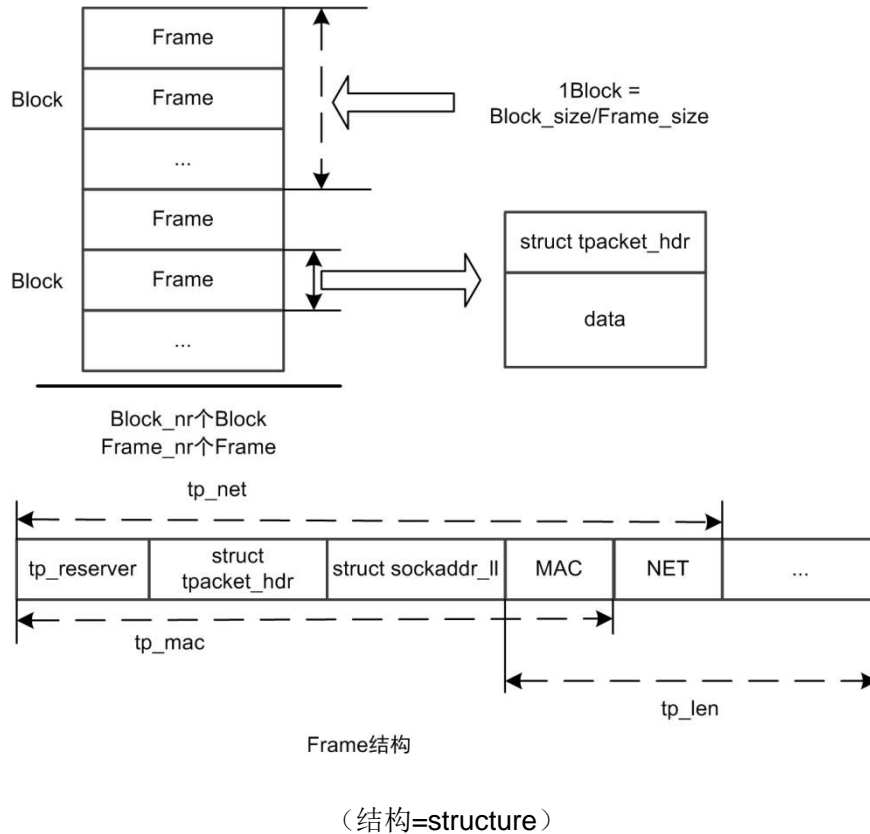


Figure 15.28 Block and fracme structure

In order to use the mmap way to perform PACKET communication correctly, the following process is needed:

- Create socket as follows:

```
int sockfd;
sockfd = socket(AF_PACKET, type, htons(ETH_P_ALL));
```

- Set the socket option to create a kernel ring buffer. The structure type of the req parameter is tpacket_req, as shown below:

```
setsockopt(sockfd, SOL_PACKET, PACKET_RX_RING, (void *)&req, sizeof(req));
```

- The application map and use buffer zone is as shown below:

```
mmap(0, size, PROT_READ|PROT_WRITE, MAP_SHARED, sockfd, 0);
```

Through the above process, you can create a PACKET communication based on MMAP mechanism. The program can wait for the buffer data to be read by calling the poll function. When the buffer zone is no longer needed, simply call the close function to close the created socket.

The following program instance shows how to use the MMAP mechanism:

Program List 15.15 AF_PACKET MMAP instance

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <net/if_ether.h>
#include <netpacket/packet.h>
#include <net/if_arp.h>
#include <sys/mman.h>
#include <poll.h>

#define BUF_SIZE    (16 * 1024 * 1024)          /* 16M          */
#define FREEM_SIZE  (2 * 1024)
#define BLOCK_SIZE  getpagesize()              /* 1 page       */

void show_packet (void *arg)
{
    struct tpacket_hdr *thdr = (struct tpacket_hdr *)arg;

    fprintf(stdout, "====packet header====\n");
    fprintf(stdout, "tpacket len: %d\n", thdr->tp_len);
    fprintf(stdout, "tpacket status: %lu\n", thdr->tp_status);
    fprintf(stdout, "tpacket snaplen: %d\n", thdr->tp_snaplen);
    fprintf(stdout, "tpacket mac offset: %d\n", thdr->tp_mac);
    fprintf(stdout, "tpacket net offset: %d\n", thdr->tp_net);
}

int process_packet (struct tpacket_hdr *thdr, int *idx)
{
    /*
     * If the status is TP_STATUS_KERNEL, there is no data.
     */
    if (thdr->tp_status == TP_STATUS_KERNEL) {
        return (-1);
    }

    show_packet((void *)thdr);
}
```

```
thdr->tp_len = 0;
thdr->tp_status = TP_STATUS_KERNEL;

(*idx)++;
(*idx) %= BUF_SIZE / FREAM_SIZE;

return (0);
}

int main (int argc, char *argv[])
{
    void                *buff = NULL;
    int                 sockfd;
    struct tpacket_req  req;
    int                 ret;
    int                 tpacket_verion = TPACKET_V1;
    int                 index          = 0, i;
    struct pollfd       pfd;

    sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
    if (sockfd < 0) {
        perror("socket");
        goto exit2;
    }

    /*
     * Set the TPACKET version to 1
     */
    ret = setsockopt(sockfd, SOL_PACKET, PACKET_VERSION,
                    (void *)&tpacket_verion, sizeof(int));
    if (ret < 0) {
        perror("setsockopt");
        goto exit2;
    }

    /*
     * Set buffer properties
     */
    req.tp_block_size  = BLOCK_SIZE;
    req.tp_frame_size  = FREAM_SIZE;
    req.tp_block_nr    = BUF_SIZE / req.tp_block_size;
    req.tp_frame_nr    = BUF_SIZE / req.tp_frame_size;

    ret = setsockopt(sockfd, SOL_PACKET, PACKET_RX_RING,
```

```
(void *)&req, sizeof(req));

if (ret < 0) {
    perror("setsocket");
    goto exit2;
}

buff = mmap(0, BUF_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, sockfd, 0);
if (MAP_FAILED == buff) {
    perror("mmap");
    goto exit2;
}

for (i = 0; i < req.tp_frame_nr; i++) {
    struct tpacket_hdr *thdr;

    /*
     * If data has been found before poll
     */
    thdr = (struct tpacket_hdr *) (buff + index * FREAM_SIZE);
    if (thdr->tp_status == TP_STATUS_USER) {
        goto proc_pkt;
    }

    pfd.fd = sockfd;
    pfd.events = POLLIN;
    pfd.revents = 0;

    ret = poll(&pfd, 1, -1);
    if (ret < 0) {
        perror("poll");
        goto exit1;
    }

proc_pkt:
    while (1) {
        thdr = (struct tpacket_hdr *) (buff + index * FREAM_SIZE);

        if (thdr->tp_status == TP_STATUS_KERNEL) {
            break;
        }

        show_packet((void *) thdr);

        thdr->tp_len = 0;
    }
}
```

```
thdr->tp_status = TP_STATUS_KERNEL;

    index++;
    index %= req.tp_frame_nr;
}
}

exit1:
    munmap(buff, BUF_SIZE);

exit2:
    close(sockfd);
    return (0);
}
```

From the above example, it can be seen that it is not required to call the `recv` function or the `recvfrom` function when the PACKET MMAP mechanism is used to receive the network data.

15.7 Network event detection

During network transmission, the network interface may be added or deleted. In the process of data transmission, the link is suddenly disconnected. These sudden situations usually cause fatal errors to the network. In order to reduce the losses caused by these problems, it is necessary for the application to take some countermeasures. For example, if the link is suddenly disconnected, it shall wait for the network to recover and retransmit. Some network protocols support the retransmission mechanism, but they all have a common feature: the number of retransmissions is limited. Therefore, there shall be a kind of mechanism to notify the application to retransmit when the network is recovered. This mechanism has the following advantages:

- After the network is disconnected, it does not occupy too much CPU time (polling detection of the network status);
- Be able to detect network recovery in a timely manner (similar to the interruption mechanism);
- Retransmission for the application is controllable (the network protocol is uncontrollable).

SylixOS implements this mechanism, which is called as the network event detection. The network event is detected by operating SylixOS standard I/O device `/dev/netevent`. This means that the device can be operated like an ordinary file to obtain event notification on the network. The only thing the application needs to know is the frame format of the event, as shown in Figure 15.29.

A SylixOS network event occupies the space of 24 bytes, which means that the application requires the space of at least 24 bytes to receive a network event. Among the 24 bytes, the first 4 bytes store the event types shown in Table 15.16. It shall be noted that the event type is stored in the first 4 bytes in big endian. 4 bytes behind the event type store a network interface name (such as en1), and other space store other data. Usually, the application can obtain a network event type in the following form (buf is the application receive buffer zone).

```
event = (buf[0] << 24) | (buf[1] << 16) | (buf[2] << 8) | (buf[3]);
```

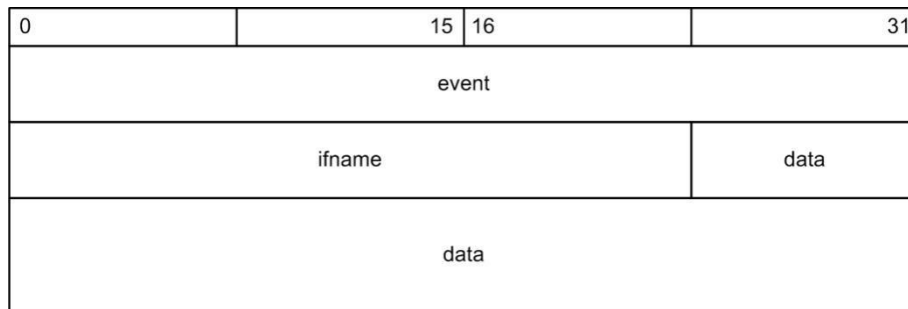


Figure 15.29 Network event frame

SylixOS supports the network event as shown in Table 15.16.

Table 15.16 Network event type

Network event type	Instructions
NET_EVENT_ADD	Network card addition
NET_EVENT_REMOVE	Network card deletion
NET_EVENT_UP	Network card enable
NET_EVENT_DOWN	Network card disable
NET_EVENT_LINK	Network card connected
NET_EVENT_UNLINK	Network card disconnected
NET_EVENT_ADDR	Network card address change
NET_EVENT_AUTH_FAI	Network card authentication failed

L

	NET_EVENT_AUTH_TO	Network card authentication timeout
	NET_EVENT_PPP_DEA	Connection stop
D		
	NET_EVENT_PPP_INIT	Enter initialization process
	NET_EVENT_PPP_AUT	Enter user authentication
H		
	NET_EVENT_PPP_RUN	Internet connectivity
	NET_EVENT_PPP_DISC	Incoming connection interrupted
ONN		
	NET_EVENT_WL_QUAL	Network card wireless environment changes (signal strength, etc.)
	NET_EVENT_WL_SCAN	Wireless network card AP scanning end

The following program shows how the application detects a network event. During program implementation, firstly, open the device `NET_EVENT_DEV_PATH`, install the `SIGALRM` signal, disable the network card 2 seconds later, and the read function to reads the network event and returns. Finally, print the network interface and event type.

Program List 15.16 Detecting network event

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <net/if_event.h>

void show_eventype (int event)
{
    switch (event) {

        case NET_EVENT_UP:
            fprintf(stdout, "event: up.\n");
            break;

        case NET_EVENT_DOWN:
            fprintf(stdout, "event: down.\n");
            break;
    }
}
```

```
case NET_EVENT_LINK:
    fprintf(stdout, "event: link.\n");
    break;

case NET_EVENT_UNLINK:
    fprintf(stdout, "event: unlink.\n");
    break;

default:
    fprintf(stdout, "event unknown\n");
}

system("ifup en1");
}

void sig_handler (int signum)
{
    system("ifdown en1");
}

int main (int argc, char *argv[])
{
    int         fd;
    char        buf[24];
    char        ifname[4];
    ssize_t     len;
    int         event;

    fd = open(NET_EVENT_DEV_PATH, O_RDONLY);
    if (fd < 0) {
        perror("open");
        return (-1);
    }

    signal(SIGALRM, sig_handler);

    alarm(2);

    len = read(fd, buf, 512);
    if (len < 0) {
        perror("read");
        return (-1);
    }
}
```

```

event = (buf[0] << 24) | (buf[1] << 16) | (buf[2] << 8) | (buf[3]);

ifname[0] = buf[4];
ifname[1] = buf[5];
ifname[2] = buf[6];
ifname[3] = buf[7];

fprintf(stdout, "ifname: %s\n", ifname);
show_eventype(event);

return (0);
}

```

Run the program in SylixOS Shell, and the running results are as follows:

```

# ./netevent_test
net interface "en1" set down.
ifname: en1
event: down.
net interface "en1" set up.

```

15.8 Standard network function library

15.8.1 ifconfig tool

ifconfig is the command used to display and configure the network device in SylixOS. The commands are described as follows:

[Command format]

```
ifconfig [netifname] [{inet | netmask | gateway}] [address]
```

[Common option]

```
inet : Specify to set the IPv4 address
netmask : Specify the setting subnet mask
gateway : Specify setting gateway

```

[Instructions for parameters]

```
netifname : The specified network interface name (such as en1)
address : The specified address (such as 192.168.1.33)

```

1. Set IP address

```
# ifconfig en1 inet 192.168.1.33
```

This command sets the IPv4 address of network interface en1 to 192.168.1.33.

2. Set gateway

```
# ifconfig en1 gateway 192.168.1.1
```

This command sets the gateway for network interface en1 to 192.168.1.1.

3. Set DNS

```
# ifconfig dns 0 192.168.1.254
```

This command sets the DNS 0 address to 192.168.1.254.

If the *ifconfig* parameter is the network interface, the network information of the interface will be printed. In particular, *ifconfig* does not specify any parameter to print information of all network interfaces in the system.

15.8.2 TFTP

1.Introduction to TFTP

TFTP (Trivial File Transfer Protocol) is a simple file transfer protocol. At the beginning of work, the TFTP client exchanges information with the server, and the client sends a read request or a write request to the server.

The first two bytes of the TFTP message indicate the operation code. For read request and write request, the file name field indicates the file on the server which the customer wants to read or write. This file field ends with 0 byte. The pattern field is an ASCII character string "netascii" or "octet" (any combination of uppercase and lowercase) and ends with 0 byte. "netascii" indicates that the data is composed of lines of ASCII characters, and the carriage return character followed by new line character (called CR/LF) is taken as the line terminator. The two end-of-line characters translate between this format and the line delimiter used by the local host. "octet" treats data as the byte stream of 8 bit per packet.

Each data packet contains a block number field, which is later used in the confirmation packet. Take reading a file as an example, the TFTP client needs to send a read request stating the file name and file mode to read. If this file can be read by this client, the TFTP server returns a data packet with block number 1. The TFTP client sends an ACK with block number 1 again. The TFTP server then sends data with block number 2. The TFTP client sends back ACK with block number 2. Repeat this process until the file is delivered. Except that the last data packet can contain less than 512 bytes of data, every other data packet contains 512 bytes of data. When the TFTP client receives a data packet of less than 512 bytes, one can know that it received the last data packet.

In the case of a write request, the TFTP client sends a write request indicating the file name and mode. If the file can be written by the client, the TFTP server returns an ACK packet with a block number of 0. The client sends the first 512 bytes of the file with a block number of 1. The server returns an ACK with a block number of 1. This type of data transmission is called as the stop-and-wait protocol (it is only used in some simple protocols, such as TFTP).

The last type of TFTP message is an error message, and its operation code is 5. It is used when the server cannot handle the read request or write request. Read and write errors during

file transfer can also result in the transmission of such messages and then stop the transmission. The error number field gives a numeric error code followed by an ASCII error message field which may contain additional information of the operating system. Since TFTP uses unreliable UDP, TFTP must handle packet loss and packet duplication. Packet loss can be resolved by the sender's timeout and retransmission mechanism. Like many UDP applications, there is no checksum in the TFTP packet. It assumes that any data errors will be checked and detected by UDP.

On the whole, TFTP is a simple protocol easy to implement: each packet size is fixed, and each packet has the confirmation mechanism which can achieve a certain degree of reliability. Of course, the disadvantages of TFTP are obvious: the transmission efficiency is not high, the sliding window mechanism is too simple, the window has only one packet size, and the timeout mechanism is not perfect.

Figure 15.30 shows five TFTP packet formats (messages with opcodes 1 and 2 use the same format).

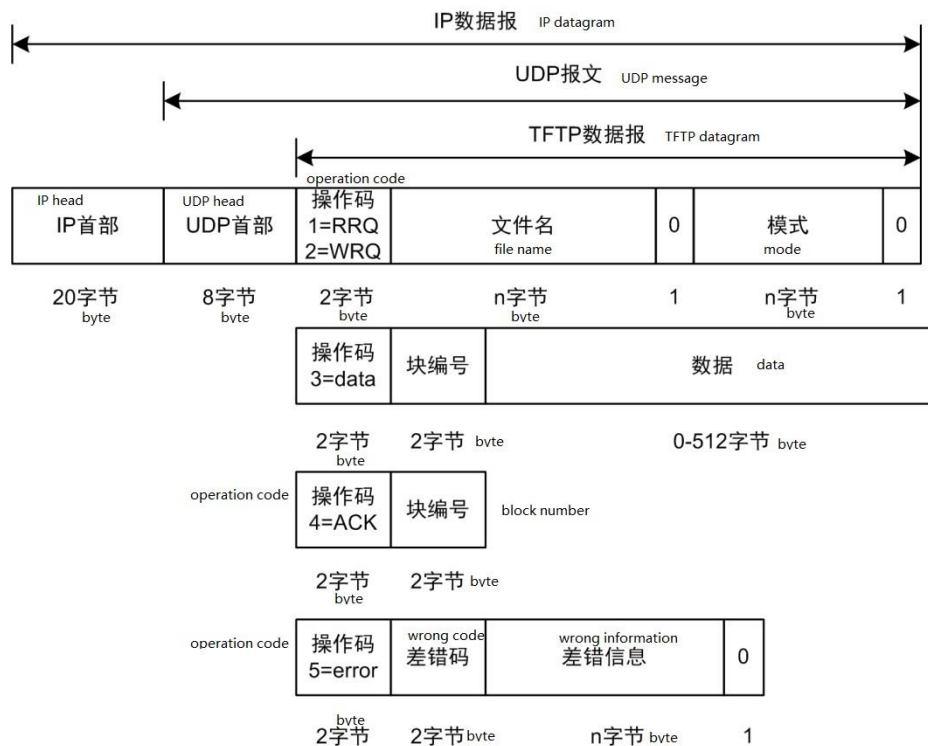


Figure 15.30 TFTP message format

2. TFTP command

The **tftp** command in SylixOS allows you to send and receive files via the TFTP protocol. The **tftpdpath** command can modify the default path of TFTP server.

[Command format]

```
tftpdpath [new path]
```

[Common option]

None

[Instructions for parameters]

new path : new path name

The following command sets the path name of the tftp server to /tmp/sylixos.

tftpdpath /tmp/sylixos**[Command format]****tftp** [-i] [Host] [{get | put}] [Source] [Destination]**[Common option]**

-i : Specify the TFTP mode as "octet"
get : Obtain a file from the TFTP server
put : Send a file to the TFTP server

[Instructions for parameters]

Host : server address
Source : source file name
Destination : Destination file name (this parameter can be empty when getting a file)

The following command gets the file sylixos.log from the TFTP server with IP 192.168.1.30.

tftp -i 192.168.1.30 get sylixos.log**15.8.3 FTP****1. Brief introduction to FTP**

FTP (File Transfer Protocol) is one of the protocols in the TCP/IP protocol family. Its purpose is to provide file sharing, that is to say, FTP completes copy between two computers. FTP uses two TCP connections to transfer a file.

- The control connection is established as a normal client server. The server passively opens the well-known port 21 for FTP, and then waits for the client's connection. The client actively opens TCP port 21 to establish connection. The control connection always waits for communication between the client and the server. The connection will send the command from the client to the server, and return the server's response;
- Each time a file is transferred between the client and the server, a data connection is created.

2. FTP data representation

The FTP protocol specification provides many options for controlling file transfer and storage. A choice must be made in each of the following four aspects.

- File types include ASCII file type, EBCDIC file type, binary file type and local file type (currently, SylixOS only supports ASCII file type and binary file type);
 - ◆
 - ◆ ASCII file type. The text file is transmitted in data connection in ASCII format. This requires the sender to convert the local text file to ASCII format, and the receiver to restore the ASCII code to the local text file. Each line transmitted in ASCII has a carriage return followed by a line feed. This means that the receiver must scan each byte to find CR and LF pairs;
 - ◆ EBCDIC file type, text file transfer requires EBCDIC at both ends;
 - ◆ Binary file type, data sent as a continuous bit stream, usually used to transfer binary files;
 - ◆ Local file type, which transfers the binary file between hosts with different byte sizes. The number of bits per byte is specified by the sender. For the system using 8bit bytes, transfer of the local file in 8bit bytes is equivalent to binary file transfer.
- Format control, this option is only valid for ASCII file type and EBCDIC file type;
 - ◆ Non-printing, the file cannot contain vertical format information;
 - ◆ Remote login format control, the file contains the remote login vertical format control explained to the printer;
 - ◆ Fortran carriage control, the first character of each line is the Fortran format control character.
- Structure:
 - ◆ The file structure (selected by default) file is considered as a continuous byte stream. There is no internal file structure;
 - ◆ Record structure, which is only used for the text file (ASCII or EBCDIC);
 - ◆ Page structure, each page is sent with a page number, so that the receiver can randomly store the pages.
- Transmission mode, which specifies how the file is transmitted in the data connection.
 - ◆ Stream mode, the file is transmitted in the form of byte stream. For the file structure, the sender prompts to close the data connection at the end of file. For the record structure, there is a dedicated two-byte serial code flag recording end and file end;
 - ◆ Block mode, the file is transmitted in a series of blocks, and each block has one or more header bytes;
 - ◆ Compression mode, a simple and full-length encoding compression method which compresses the same byte which appears consecutively.

The following is chosen in SylixOS:

- binary;

- Format, non-printing;
- Structure, file structure;
- Transmission mode, stream mode.

3. FTP protocol command

Command and answer are transmitted in ASCII code on the control connection between the client and the server. This requires that CR and LF pairs be returned at the end of each line (that is to say, each command or each answer).

Table 15.17 shows the SylixOS-supported FTP command.

Table 15.17 FTP command

FTP command	Instructions	FTP command	Instructions
USER	Specify the username on the remote system	RNFR	The first half of the file renaming process. The old path and file name of the file to be renamed
PASS	Send password to the remote user (used after USER command)	REST	Identify the data points in the file, from which point the file will continue to be transmitted
CWD	Change the current directory to the specified directory of the remote file system	RETR	This command causes the server to send a copy of the file specified in the pathname to the client
CDUP	Change the current directory to the root directory of the remote file system	STOR	Let the server receive a file from data connection
PWD	Return the name of the current working directory in the answer	APPE	Let the server prepare to receive a file and instruct it to attach the data to the specified file name. If the specified file does not exist, create it
ALLO	Allocate x bytes on the server before sending the file	SYST	Used to find out the type of operating system on the server
PORT	Specify an IP address and local port for data connection	MKD	Create a directory specified in the path name

PASV	Tell the server to listen for data connections on a non-standard port	RMD	Delete a directory specified in the path name
TYPE	Determine the data transmission mode	DELE	Delete the file specified in the path name on the server site
LIST	Let the server send a list to the client	MDTM	Update time information
NLST	Let the server send the customer a list of directories	SIZE	Send file size
NOOP	Do nothing	SITE	Provide some service features

4. FTP answer

The answers are all 3-digit figures in ASCII format, and are followed by the message option. The reason is that the software system needs to decide how to answer according to the digital code, but the option string is for manual processing. Since the client usually outputs digital response and message string, an interactive user can determine the meaning of the response by reading the message string (without remembering the meaning of all digital answer codes).

Each digit of the answer 3-digit code has a different meaning. The meanings of the first and second digits of the answer code are shown in Table 15.18.

Table 15.18 Meaning of first and second digits of the answer code

Answer	Instructions
1yz	Be sure to prepare the answer. It just starts when you expect another answer before sending another command
2yz	Certainly complete the answer. A new command can be sent
3yz	Affirmative intermediary answer. The command has been accepted but another command must be sent
4yz	Transient negative completion answer. The requested action did not occur, but the error status is temporary, so the command can be issued later
5yz	Permanent negative completion answer. The command is not accepted and will not be retried
x0z	Syntax error
x1z	Information
x2z	Connection. Answer means control or data connection
x3z	Identification and accounting. Answer used for registration or accounting command
x4z	Unspecified
x5z	File system state

The third digit gives the additional meaning of the error message. For example, here are some typical answers, all with a possible message string.

- 125: the data connection is already open; the transmission starts;
- 200: ready command;
- 214: help message (user-oriented);
- 331: the user name is ready, required to input the password;
- 425: can't open data connection;
- 452: wrong writing file;
- 500: syntax error (unrecognized command);
- 501: syntax error (invalid parameter);
- 502: unrealized MODE (mode command) type.

5. FTP connection management

The data connection has the following three major applications:

- Send a file from the client to the server;
- Send a file from the server to the client;
- Send a list of files or directories from the server to the client.

The FTP server sends the list of files back from the data connection instead of controlling multiple lines of answers on the connection. This avoids limitations of line finiteness in directory size, and it is easier for customers to save the directory list as a file instead of displaying the list on the terminal.

Earlier we said that the SylixOS transfer method is stream mode, and the end of file is a sign of closing data connection. This means that a brand new data connection must be established for each file transfer or directory list. The process is as follows:

- Because the customer issues the command to establish data connection, the data connection is established under the control of the customer;
- The client usually selects a temporary port number for data connection on the client host. The customer releases a passive open from the port;
- The client uses the PORT command to send the port number from the control connection to the server;
- The server receives the port number on the control connection and issues an active open to the port on the client host. The data connection end of the server uses port 20.

6. FTP command

The ***ftpd***s command in SylixOS can be used to view all the ftp information linked to SylixOS. The ***ftpdpath*** command can modify the default path of FTP server.

[Command format]

```
ftpdpath [new path]
```

[Common option]

None

[Instructions for parameters]

```
new path : new path name
```

The following command sets the default path name of the FTP server to /sylixos.

```
# ftpdpath /sylixos
```

[Command format]

ftpd

[Common option]

None

[Instructions for parameters]

None

The ***ftpd*** command is displayed as follows, and the result shows that a 192.168.1.30 FTP connection has been established.

```
# ftpd
ftpd show >>
ftpd path: /sylixos

      REMOTE                TIME                ALIVE (s)
-----
192.168.1.30  Sat Jan 09 12:01:00 2167                11

total ftp session : 1
```

15.8.4 Telnet

1. Brief introduction to Telnet

Telnet protocol is a simple remote login protocol. Its service process can be divided into the following three steps:

- The local user logs in to the remote system on the local terminal;
- The keyboard input on the local terminal is transmitted to the remote terminal one by one;
- Send the remote output back to the local terminal.

In the above process, the input / output are transparent to the kernel of the remote system, and the remote login service is also transparent to the user. This transparency is an important feature of Telnet.

Telnet provides three basic services. Firstly, a Network Virtual Terminal (NVT) is defined to provide a standard interface for the remote system. The client program do not have to know in detail all possible remote systems. They only need the program which uses the standard interface. Secondly, it includes a mechanism for client and server to negotiate options, and also provides a set of standard options. Finally, it handles both ends of the connection peer-to-peer. That is to say, both parties to the connection can be programs. In particular, the client is not necessarily not a user terminal. Any program is allowed as a client.

When the user calls Telnet, the application on the user's machine establishes a TCP connection with the remote server as a client, and communicates on this connection. At this point, the client accepts the keyboard message from the user's keyboard and sends it to the server. At the same time, it receives the character sent back from the server, and displays it on the user's screen.

The server itself does not directly process the messages transmitted from the client. Instead, the messages are sent to the operating system for processing, and the returned data is then passed on to the customer. That is to say, the server at this time, which we call "Pseudo Terminal", allows a run the program like the Telnet server to transfer characters to the operating system, and makes the characters seem to come from the local keyboard.

In order to provide interoperability between different operating systems and different types of computers, Telnet specifically provides a standard keyboard definition method called Network Virtual Terminal (NVT). The client program converts the keystrokes and command sequences from the user terminal into NVT format and sends them to the server. The remote server program converts the received data and commands from the NVT format to the format required by the remote system. For the returned data, the remote server converts the data from the format of the remote machine to the NVT format, and the local client converts the data from the NVT format to the format of the local machine.

SylixOS currently supports Telnet servers, which means that SylixOS device resources can be managed by connecting to the SylixOS target system via Telnet client program.

2. Telnet protocol command

Table 15.19 shows the common commands of Telnet protocol.

Table 15.19 Telnet protocol command

Telnet command	Instructions	Telnet command	Instructions
EOF	EOF character	EL	Delete line
SUSP	Hang the current process	GA	Continue
ABORT	Abnormally terminate process	SB	Suboption start
EOR	Record terminator	WILL	The sender wants to activate the option
SE	Suboption end	WONT	The sender wants to disable the option
NOP	No operation	DO	The sender wants the recipient to activate the option
DM	Data tag	DONT	The sender wants the receiver to disable the option
BRK	Interrupt	IAC	IAC
IP	Interrupt process	AYT	Whether the other party is running

AO	Abnormally terminate output	EC	Escape character
----	--------------------------------	----	------------------

The commonly used options are negotiated as follows:

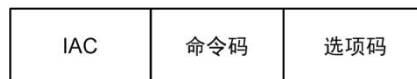
- WILL xxx: I want to have xxx features. Do you agree?
- WONT xxx: I don't want to have xxx features;
- DO xxx: I agree that you can have xxx features;
- DONT xxx: I do not agree that you have xxx features.

Option negotiation requires 3 bytes, first IAC, then WILL, DO, WONT, or DONT. The last identification byte is used to indicate the operation option, as shown in Table 15.20.

Table 15.20 Option identification

Option identification	Instructions
ECHO (1)	Echo
SGA (3)	Suppression continues
STATUS (5)	State
TM (6)	Timing mark
TTYTYPE (24)	Terminal type
NAWS (31)	Window size
TSPEED (32)	Port speed
LFLOW (33)	Remote flow control
LINEMODE (34)	Line mode
ENVIRON (36)	Environment variable

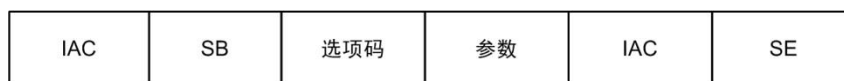
Option negotiation is the most complicated part of the Telnet protocol. When a party wants to execute an option, it needs to send a request to the other end. If the other party accepts the option, the option works at both ends. Otherwise, two ends maintain the original mode. The command format of Telnet is as shown in Figure 15.31. IAC is the reserved code in the Telnet protocol. Both parties use IAC to determine whether the received byte is data or command. The Telnet protocol command contains at least two characters (IAC and command code) in the byte sequence. Option negotiation has 3 bytes. The third byte is the negotiation option. When the negotiation option has sub-options, suboption negotiation shall be performed. The command format is shown in Figure 15.32.



(命令码=command code)

(选项码=option code)

Figure 15.31 command format of Telnet



(选项码=option code)

(参数=parameter)

Figure 15.32 Suboption negotiation command format

15.8.5 ping

1. Brief introduction to ping

The **ping** command is a tool to check whether the network connection of another host system on the network is normal. The working principle of the **ping** command is: to send an

ICMP packet to another host system on the network. If the specified system receives the packet, it sends the packet back to the sender.

Earlier we introduced that the ICMP protocol is an integral part of the IP layer. Therefore, the raw socket (SOCK_RAW) shall be used to send and receive the ICMP packet. The ICMP header format used by the *ping* command is as shown in Figure 15.33.

The identification part is used as the contracting label of the sender and the receiver. The sender sends the label to the receiver. After receiving the reply from the receiver, the sender checks the label to determine that the reply is concerned by itself. Count of ping recorded by the serial number.

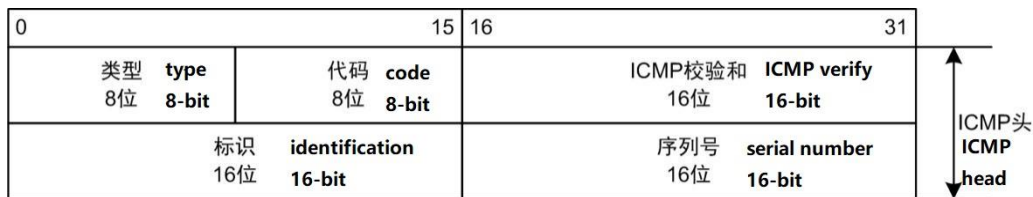


Figure 15.33 ping command ICMP header

2. ping command

In SylixOS, you can use the *ping* command to check whether the network is connected. The command is described as follows:

[Command format]

```
ping IP/hostname [-l datalen] [-n times] [-i ttl] [-w timeout]
```

[Common option]

```
-l : data length
-n : times
-i : TTL value
-w : timeout value
```

[Instructions for parameters]

```
IP : Destination IP address
hostname : Hostname (eg www.sylixos.com)
```

The following command checks whether www.sylixos.com is connected, displayed as follows:

```
# ping www.sylixos.com -n 3
Execute a DNS query...
Pinging www.sylixos.com [106.39.47.146]

Pinging 106.39.47.146

Reply from 106.39.47.146: bytes=32 time=0ms TTL=64
```



```

Reply from 106.39.47.146: bytes=32 time=0ms TTL=64
Reply from 106.39.47.146: bytes=32 time=0ms TTL=64

Ping statistics for 106.39.47.146:
    Packets: Send = 3, Received = 3, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

```

15.8.6 PPP

1. Brief introduction to PPP

The full name of PPP protocol is the Point-to-Point Protocol. It is a data link layer protocol which provides transmission of encapsulated network layer data packet on the point-to-point links. It is located at the second layer of the TCP/IP protocol stack, and is mainly designed to conduct point-to-point data transmission on the co-asynchronous link supporting full-duplex.

The PPP protocol is developed based on the SLIP (Serial Line IP). Because the SLIP protocol only supports asynchronous transmission mode, and there is no negotiation process (especially the network layer attributes, such as the IP addresses of both parties, cannot be negotiated). In the later development process, it is gradually replaced by PPP.

PPP is mainly composed of three types of protocols: Link Control Protocol Family (LCP), Network Layer Control Protocol Family (NCP) and PPP extension protocol family. The link control protocol is mainly used to establish, remove and monitor PPP data link. The network layer control protocol family is mainly used to negotiate the format and type of data packet transmitted on the data link. The PPP extension protocol family is mainly used to provide further support for PPP function, and PPP also provides authentication protocol family (PAP and CHAP) for network security.

2. PPP frame format

PPP frame format, as shown in Figure 15.34.

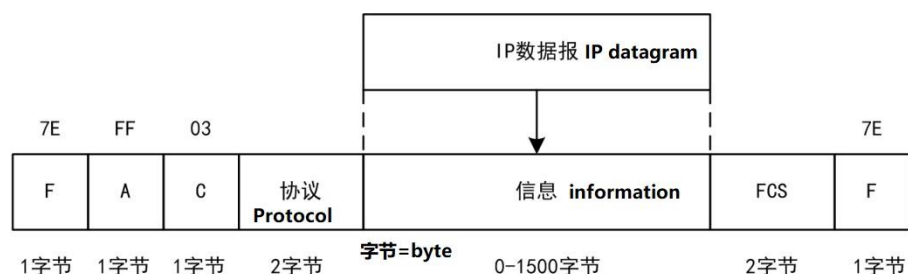


Figure 15.34 PPP frame

The field F stands for Flag. This flag identifies the start and end of a physical frame. The value is 0x7E; the field A represents Address. This address identifies the PPP broadcast

address. The value is 0xFF; the field C represents Control and is the control field. The value is 0x03. The FCS field is the frame check field. What truly belongs to PPP message contents is the F, A, C, protocol and information fields.

According to the different protocol fields, the information fields are also different, as shown in Table 15.21.

Table 15.21 Protocol and information

Agreement	Information
0x0021	IP datagram
0xC021	Link control protocol LCP
0x8021	Network control data NCP
0xC023	Security certification PAP
0xC025	LQR
0xC223	Security certification CHAP
0x8031	Bridge NC
0x802b	IPX control protocol

When bit 0x7E, which is the same as the flag field, appears in the information field, some measures must be taken. Since the PPP protocol is character-oriented, it cannot use the zero-bit insertion method used by HDLC (high-level data link control), but uses a special character padding. The specific approach is to convert each 0x7E byte which appears in the information field into a 2-byte endian (0x7D, 0x5E). If a 0x7D byte appears in the information field, it is converted into a 2-byte sequence (0x7D, 0x5D). If an ASCII control character appears in the information field, a 0x7D byte shall be added before the character. The purpose is to prevent these apparent ASCII control characters from being incorrectly interpreted as control characters.

3. PPP negotiation process

Before the link is established, PPP shall perform a series of negotiation processes. The process is as follows: PPP firstly performs LCP negotiation. Negotiation contents include MRU (maximum receiving unit), magic number, authentication mode, asynchronous character mapping and other options. After LCP negotiation succeeds, the link establish state is entered. If CHAP or PAP verification is configured, the CHAP or PAP verification stage is entered. After verification is passed, the network stage will be entered for negotiation, such as negotiation of IPCP, IPXCP and BCP. Failure in negotiation at any stage will tear down the link. The magic word is mainly used to detect link self-loop. PPP sends the Echo Request and Echo Reply packets to detect self-loop and maintain the link state. If the magic word in maximum allowed self-loop number of Echo Request message is the same with the last magic word, then the network is judged to have self-loop. If the link sends self-loop, the corresponding measures shall be taken to reset the link. In addition, when LCP sends config request, it can also detect self-loop. After LCP detects self-loop, and a certain number of packets are sent, the link will also be reset. If the Echo Request packet sent by PPP is lost, the link will be reset after the

maximum number of consecutive lost packets is lost, so as to avoid excessive invalid data transmission.

4. PAP authentication process

PAP is the two-way handshake protocol, which authenticates the user with the user name and password. The PAP authentication process is as follows: when the links at both ends can transmit data to each other, the authenticatee sends the local user name and password to the authenticator. The authenticator checks whether there is this user and the password is correct based on the local user table (or radius server). If it is correct, ACK message will be sent to the peer to inform that the peer has been allowed to enter the next phase of negotiation; otherwise, NAK message shall be sent to inform peer verification failure. The link will not be closed directly at the moment. Only when the number of verification failures reaches a certain value, the link can be closed to prevent unnecessary LCP renegotiation process due to mis-transmission, network interference and the like. The characteristic of PAP is to pass user name and password in plain language on the network. If it is intercepted during transmission, it may pose great threat to network security. Therefore, it is suitable for the environment with relatively low requirements for network security.

5. CHAP verification process

CHAP is the three-way handshake protocol. Its characteristic is that it only transmits the user name on the network but does not transmit the user password. Therefore, it is more secure than PAP. The CHAP verification process is as follows: firstly, the verifying party sends some randomly generated packets to the verified party, and simultaneously sends the host name of the local end to the verified party. When receiving the challenge from the peer end to the local end, the verified party searches for the user's password based on the host name of the verifying party and the local user table in the packet. If the the user with the same host name of the verified party in the user table, the message ID and the user's key are used to generate a response by using the Md5 algorithm, and then the response and its own host name are sent back. After receiving this response, the verifying party uses the message ID, the reserved password (key) and the random message to obtain the result by using the Md5 algorithm, compares it with the verified party, and returns the corresponding results based on the comparison results.

6. PPP API

The following function can be used in SylixOS to create PPPoS-type network card

```
#include <SylixOS.h>
INT  API_PppOsCreate(CPCHAR      pcSerial,
                    LW_PPP_TTY   *ptty,
                    PCHAR        pcIfName,
                    size_t        stMaxSize);
```

Prototype analysis of Function API_PppOsCreate:

- For success of the function, return 0. For failure, return -1 and set the error number;

- Parameter ***pcSerial*** is the serial interface device name;
- Parameter ***ptty*** is the working parameter of the serial port;
- Output parameter ***pcIfName*** returns the network interface name;
- Parameter ***stMaxSize*** is the size of the buffer zone.

The following function will create the PPPoE type network card.

```
#include <SylixOS.h>
INT API_PppOeCreate(CPCHAR pcEthIf, PCHAR pcIfName, size_t stMaxSize);
```

Prototype analysis of Function API_PppOeCreate:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcEthIf*** is the Ethernet network card name;
- Output parameter ***pcIfName*** returns the network device name;
- Parameter ***stMaxSize*** is the size of the buffer zone;

The following function will create PPPoL2TP network card.

```
#include <SylixOS.h>
INT API_PppOl2tpCreate(CPCHAR pcEthIf, CPCHAR pcIp,
                      UINT16 usPort, CPCHAR pcSecret,
                      size_t stSecretLen, PCHAR pcIfName,
                      size_t stMaxSize);
```

Prototype analysis of Function API_PppOl2tpCreate:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcEthIf*** is the Ethernet name;
- Parameter ***pcIp*** is the server address;
- Parameter ***usPort*** is the server port;
- Parameter ***pcSecret*** is security-related;
- Parameter ***stSecretLen*** is the size of the buffer zone;
- Output parameter ***pcIfName*** returns the network device name;
- Parameter ***stMaxSize*** is the size of the buffer zone.

The following function dials the connection.

```
#include <SylixOS.h>
INT API_PppConnect(CPCHAR pcIfName, LW_PPP_DIAL *pdial);
```

Prototype analysis of Function API_PppConnect:

- For success of the function, return 0. For failure, return -1 and set the error number;

- Parameter ***pcIfName*** is the network device name;
- Parameter ***pdial*** is the dialing parameter.

The following function disconnects and deletes the connection.

```
#include <SylixOS.h>
INT API_PppDisconnect(CPCHAR pcIfName, BOOL bForce);
INT API_PppDelete(CPCHAR pcIfName);
```

Prototype analysis of Function API_PppDisconnect:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcIfName*** is the network device name;
- Parameter ***bForce*** indicates whether it is forcibly disconnected.

Prototype analysis of Function API_PppDelete:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcIfName*** is the network device name;

The following function gets the device connection status.

```
#include <SylixOS.h>
INT API_PppGetPhase(CPCHAR pcIfName, INT *piPhase);
```

Prototype analysis of Function API_PppGetPhase:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pcIfName*** is the network device name;
- Output parameter ***piPhase*** returns the connection status.

15.8.7 Network address translation

1. Brief introduction to NAT

NAT is the process of converting the IP address in the IP datagram header to another IP address. In practical applications, NAT is mainly used to implement private network access to external network. This method of mapping the majority of private IP addresses by using a small number of public IP addresses can relieve the pressure of depletion of the IP address space to some extent.

The private network address (hereinafter referred to as the private network address) refers to the IP address of the internal network or the host. The public network address (hereinafter referred to as the public network address) refers to a globally unique IP address

on the Internet. IANA (Internet Assigned Number Authority) specifies that the following IP addresses are reserved as private network addresses and are not allocated on the Internet.

- Class A private address: 10.0.0.0 to 10.255.255.255;
- Class B private address: 172.16.0.0 to 172.31.255.255;
- C private addresses: 192.168.0.0 to 192.168.255.255.

If other network segments outside the above three ranges are selected as internal network addresses, it may cause confusion during intercommunication with other networks.

2. NAT translation mechanism

An address port mapping table is maintained in the router. All messages passing through the router and requiring address translation will be modified accordingly through the mapping table for translation between <private address + port> and <public address + port>. The translation process is as follows:

- When the internal network host sends the message to the outside, the router replaces the source IP address and port of the message with the external network address and port of the router;
- When the external message enters the internal network, the router will search the address port mapping table, and convert the destination address and port of the message to the real destination address.

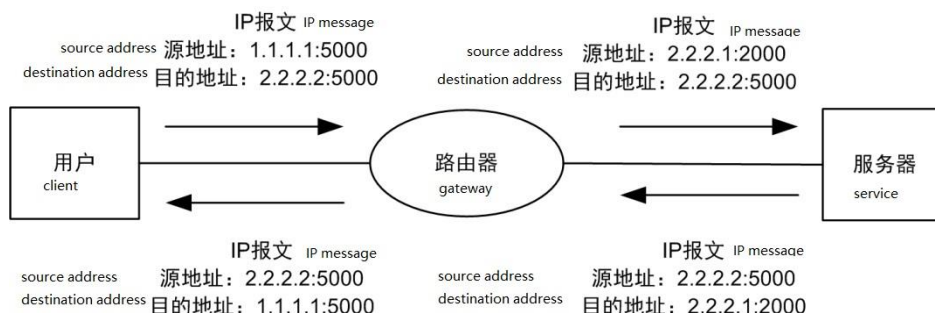


Figure 15.35 Schematic diagram of NAT

As shown in Figure 15.35, the process for internal users to access external servers is as follows:

- The user sends the message with source address of 1.1.1.1:5000 and destination address of 2.2.2.2:5000 to the server;
- When the process sent by the user to the server passes the router, the message source address is changed from 1.1.1.1:5000 to 2.2.2.1:2000;
- After receiving the user's message, the server sends the message back to the user. The source address of the message is 2.2.2.2:5000, and the destination address is 2.2.2.1:2000;

- When the message sent from the server to the user passes through the router, the address is changed from 2.2.2.1:2000 to 1.1.1.1:5000.

The address translation process described above is transparent to the terminal (users and servers in Figure 15.35). For the external server, it thinks that the client's IP address is 2.2.2.1 and does not know that this address of 1.1.1.1. Therefore, NAT "hides" topology of the private network.

When an internal network host accesses an external network, the number of hosts on the internal network is large, and there is only one external IP address, address translation may appear to be inefficient. Solving this problem requires a private network with multiple external addresses.

The NAT implemented in SylixOS uses the address pool to solve the above problems. The address pool is a collection of legal IP addresses (public network IP addresses). Users can configure the appropriate IP address pool based on the number of legal IP addresses they have, the number of internal network hosts and the actual application. When the host accesses the external network from the internal network, an IP address will be selected from the address pool as the translated message source address.

NAT address translation enables a large number of internal hosts to use a small number of public IP addresses to access external network resources, and provides "privacy" protection for internal hosts. However, address translation has the following disadvantages:

- The data message involving IP addresses cannot be encrypted. Otherwise, address translation of the data message cannot be performed. In an application layer protocol, the message cannot be encrypted if they contain addresses or ports to be translated. For example, the encrypted FTP connection cannot be used. Otherwise, FTP's port command cannot be converted correctly;
- Network debugging becomes more difficult. For example, if a host on an internal network tries to attack other networks, it is difficult to indicate which machine is malicious, because the host's IP address is blocked.

3. NAT API

In SylixOS, the following function is called to initialize the NAT library.

```
#include <SylixOS.h>
VOID Lw_Inet_NatInit(VOID);
```

The following function is called to start NAT translation and stop NAT translation.

```
#include <SylixOS.h>
INT Lw_Inet_NatStart(CPCHAR pcLocalNetif, CPCHAR pcApNetif);
INT Lw_Inet_NatStop(VOID);
```

Prototype analysis of Function Lw_Inet_NatStart:

- For success of the function, return 0. For failure, return -1 and set the error number;

- Parameter ***pcLocalNetif*** is the local internal and external network interface;
- Parameter ***pcApNetif*** is the external network interface.

4. NAT command

You can use the ***nats***[®] command to view NAT information in SylixOS, and use the ***nat*** command to start and stop the NAT network. The command is indicated as follows:

[Command format]

```
nat [-stop] / {[LAN netif] [WAN netif]}
```

[Common option]

```
-stop : Stop the NAT network
```

[Instructions for parameters]

```
LAN netif : Local network interface
WAN netif : External network interface
```

The following command will set wl2 as the local network interface and en1 as the external network interface.

```
# nat wl2 en1
```

[Command format]

```
nats
```

[Common option]

```
None
```

[Instructions for parameters]

```
ip addr : IP address
```

```
# nats
```

```
NAT networking show >>
```

```
LOCAL IP      LOCAL PORT ASS PORT  PROTO  IDLE(min)  STATUS
-----
```

15.8.8 SylixOS network routing

1. Routing principle

IP routing selection is usually simple. If the destination host is directly connected to the source host (such as a point-to-point link) or both are on the shared network (Ethernet or token ring network), the IP datagram is sent directly to the destination host. Otherwise, the host sends the datagram to the default router, and the router forwards the datagram. Most hosts use this simple mechanism.

In the general system, IP can receive the datagrams (i.e., locally generated datagrams) from TCP, UDP, ICMP and IGMP and send them, or can receive datagrams (datagrams to be forwarded) from a network interface and send them. The IP layer has a routing table in memory. When it receives a datagram and sends it, it will search the table once. When the datagram comes from a network interface, IP firstly checks whether the destination IP address is one of the IP addresses of the machine or an IP broadcast address. If this is the case, the datagram is sent to the protocol module specified by the IP header protocol field for processing. If the destination of the datagram is not these addresses, then if the IP layer is set as the router function, the datagram is forwarded. Otherwise, the datagram is discarded.

Each entry in the routing table contains the following information:

- The next-hop router's IP address or the directly-connected network IP address. The next-hop router refers to a router on the directly connected network, through which the datagram can be forwarded. The next-hop router is not the ultimate destination, but it can forward the datagram delivered to the ultimate destination.
- Flag. One of the flags indicates whether the destination IP address is the network address or the host address, and the other flag indicates whether the next-hop router is the true next-hop router or a directly connected interface, as shown in Table 15.22.
- Specify a network interface for the datagram transmission;
- Network interface name (character string name).

IP routing is selected hop-by-hop. From information of this routing table, it can be seen that IP does not know the full path to any destination (of course, except for those destinations directly connected to the host). All IP routing selection only provides IP address of the next-hop router for datagram transmission. It assumes that the next-hop router is closer to the destination than the host which sends the datagram, and the next-hop router is directly connected to the host.

IP routing selection mainly completes the following functions:

- Search the routing table for entries which can exactly match the destination IP address (the network number and the host number must match). If found, the message is sent to the next-hop router or directly connected network interface specified by the entry (depending on the value of the flag field).
- Search the routing table, and address the network address which can match the destination IP address;
- Search the routing table for entries marked "default". If found, the message is sent to the next-hop router specified by the entry.

If none of these steps is successful, the datagram cannot be transmitted. If the datagram which cannot be transmitted comes from the local machine, a "host unreachable" or "network unreachable" error will be generally returned to the application program which generates the datagram.

Full host address matching is performed before the network number matches. The default route is selected only if they all fail. The default route as well as the ICMP indirect message sent by the next-hop router (if we select the wrong default route for the datagram) is the powerful feature of the IP routing selection mechanism.

Route matching always matches the local host in priority. If the local host fails to match, it matches the network address.

The following table describes the route flag:

Table 15.22 Route flag

Route flag	Instructions
U	This route can be used
G	Route to a gateway (router). If there is no such flag, the destination address is a direct connection
H	Route to a host. The destination address is a full host address. If this flag is not set, the route is to a network, and the destination address is a network address: a network number, or combination of network number and subnet
D	The route is created by the redirect message
M	The route has been modified by the redirect message

The flag G is very important because it distinguishes the indirect route and the direct route (the direct route is not set with this flag). The packet sent to the direct route not only has the destination IP address, but also has its link layer address. When sending to an indirect route, the IP address indicates the final destination address, and the link layer address indicates the gateway (the next-hop route).

H flag indicates that the destination address is a full host address. H flag is not specified to indicate that the destination address is a network address (the host number part is 0). When searching the routing table for a certain destination IP address, the host address entry must exactly match the destination address, but the network address entry only needs to match the network number and subnet number of the destination address.

Whenever an interface is initialized, a direct route is automatically created for the interface. For point-to-point link and loopback interface, the route is to the host (for example, H flag is set). For broadcast interface, such as Ethernet, routing is to the network.

If the route to the host or network is not directly connected, then the route table must be added. The **route** command describes how to add a route entry to the route table.

2. Routing API (old style interface)

The application program prior to SylixOS 1.5.6 can call the simple routing interface to operate the routing table, and the following function can add a piece of routing information to the routing table:

```
#include <sys/route.h>
int route_add (struct in_addr *pinaddr, int type, const char *ifname);
```

Prototype analysis of Function route_add:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter **pinaddr** is the routing address;

- Parameter ***type*** is the route type, as shown in Table 15.23;
- Parameter ***ifname*** is the network interface name.

Table 15.23 Route type

Route type	Instructions
ROUTE_TYPE_HOST	The destination is host
ROUTE_TYPE_NET	The destination is network
ROUTE_TYPE_GATEWAY	The destination is gateway
ROUTE_TYPE_DEFAULT	Default gateway

Calling the `route_change` function can change a piece of route information:

```
#include <sys/route.h>
int route_change(struct in_addr *pinaddr, int type, const char *ifname);
```

Prototype analysis of Function `route_change` prototype:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pinaddr*** is the routing address;
- Parameter ***type*** is the route type as shown in Table 15.23;
- Parameter ***ifname*** is the network interface name.

Call the `route_delete` function to delete a piece of route information:

```
#include <sys/route.h>
int route_delete(struct in_addr *pinaddr);
```

Prototype analysis of Function `route_delete`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***pinaddr*** is the routing address.

Calling the `route_getnum` function can get the total route entry in the routing table:

```
#include <sys/route.h>
int route_getnum(void);
```

Calling the `route_get` function can obtain routing information which satisfies certain conditions:

```
#include <sys/route.h>
int route_get(u_char flag, struct route_msg *msgbuf, size_t num);
```

Prototype analysis of Function `route_add`:

- This function returns the number of route entries obtained;
- Parameter ***flag*** is the type mask as shown in Table 15.23;
- Output parameter ***msgbuf*** is the routing information structure;

- Parameter *num* is the number of routing information structure caches.

The route entry information obtained by the `route_get` function will be stored in the `route_msg` structure. The structure is described as follows:

```
struct route_msg {
    u_char      rm_flag;          /* type                */
    int         rm_metric;        /* Not used temporarily */
    struct in_addr rm_dst;        /* Destination address information*/
    struct in_addr rm_gw;        /* Gateway information  */
    struct in_addr rm_mask;      /* Subnet mask information */
    struct in_addr rm_if;        /* Network interface address */
    char        rm_ifname[IF_NAMESIZE]; /* Network interface name */
};
```

Note: the route type is stored as OR in the member `rm_flag` of the `route_msg` structure.

3. route command

Use the **route** command to display or set system routing table information.

[Command format]

```
route [add | del | change] {-host | -net | -gw} [addr] [[dev] | if]
```

[Common option]

```
add : Add routing table entries
del : Delete routing table entries
change : Change routing table entries
-host : Host address
-net : website address
-gw : Gateway address
dev : Network equipment
```

[Instructions for parameters]

```
addr : IP address
if : Network device interface name
```

4. Output information meaning

```
# route
kernel routing tables
Destination      Gateway          Mask             Flag             Interface

build-in routing tables
Destination      Gateway          Mask             Flag             Interface
192.168.7.0     *                255.255.255.0   U                en1
192.168.7.30    *                255.255.255.0   UH               en1
127.0.0.0       *                255.0.0.0       U                lo0
```

127.0.0.1	*	255.0.0.0	UH	lo0
default	192.168.7.1	255.255.255.0	UG	en1

The meanings of various titles of the output information are as follows:

- Destination: routing destination address;
- Gateway: gateway address;
- Mask: subnet mask;
- Flag: route flag, as shown in Table 15.22;
- Interface: network interface.

The second line of the output reciprocal is the loopback interface. Its name is lo0. The route flag is “UH”. No “G” flag indicates that the route is not a gateway but a direct route. “H” flag indicates that the destination address (127.0.0.1) is a host address, but not a network address.

The last line is the default route. Each host has one or more default routes. This item indicates that if no specific route is found in the routing table, the packet is sent to the router 192.168.7.1, which means that the current host can use this routing table entry to access other systems through the Internet and the router 192.168.7.1. "UG" flag indicates that it is a gateway.

5. Routing API (new interface)

The application program after SylixOS 1.5.6 can manipulate route entries by calling standard IO ioctl function or write function.

The supported ioctl commands are as follows:

- SIOCADDRT: add a piece of routing information;
- SIOCDELRT: delete a piece of routing information;
- SIOCCHGRT: modify a piece of routing information;
- SIOCGETRT: get a piece of routing information;
- SIOCLSTRT: traverses IPv4 routing information;
- SIOCLSTRTM: get the entire IPv4 routing information.

Parameter ***pArg*** of the ioctl function is a pointer to the following type of structure. The structure is as follows:

```
struct rtenry {
    u_long          rt_pad1;
    struct sockaddr rt__dst;          /* Destination address */
    struct sockaddr rt_gateway;      /* Network gateway */
    struct sockaddr rt_genmask;      /* Subnet mask */
    char           rt_ifname[IF_NAMESIZE]; /* Network Interface */
}
```

```

    u_short          rt_flags;
    u_short          rt_refcnt;
    u_long           rt_pad3;
    void             *rt_pad4;
    short            rt_metric;          /* measure          */
    void             *rt_dev;            /* Unused equipment */
    u_long           rt_hopcount;        /* Hops count       */
    u_long           rt_mtu;             /* Routing MTU      */
    u_long           rt_window;
    u_short          rt_irtt;
    u_long           rt_pad5[16]};

```

The structure related with the command SIOCCLSTRT is as follows:

```

struct rtenry_list {
    u_long           rtl_bcmt;           /* Rtenrt cache size */
    u_long           rtl_num;           /* The number of routes in the
cache */
    u_long           rtl_total;         /* All routes          */
    struct rtenry   *rtl_buf;          /* Route cache         */
};

```

The structure related with the command SIOCCLSTRTM is as follows:

```

struct rt_msghdr_list {
    size_t          rtml_bsize;         /* Rtml_buf buffer size */
    size_t          rtml_rsize;         /* Number of cache routes */
    size_t          rtml_tsize;         /* The system returns all the
routes */
    struct rt_msghdr *rtml_buf;         /* Route cache          */
};

```

[Instance code]:

It has been known that various operations on route entries can be implemented through the ioctl function of the standard IO, including add, delete, modify and get route information.

In the following program, the operation of adding a route is implemented by filling the information in external structure *rtenry* of the route. Firstly, clear the structure information with the bzero function, set the host route to RTF_UP | RTF_HOST, fill in the destination address, subnet mask and gateway protocol cluster and protocol length, then write the desired destination address, subnet mask and gateway information to *sin_addr* through the inet_aton function, and finally set *metric* and *rt_ifname*.

Use the socket generated by AF_INET, and call SIOCADDRT command through the ioctl function to add the route.

```

#include <stdio.h>
#include <net/route.h>

```



```
#include <string.h>
#include <socket.h>

int main (int argc, char **argv)
{
    int          iSock;
    int          iRet;
    struct rtable rtable;

    bzero(&rtable, sizeof(struct rtable));

    rtable.rtable_flags = RTF_UP | RTF_HOST;          /* Set host routes      */

    rtable.rtable_dst.sa_len    = sizeof(struct sockaddr_in);
    rtable.rtable_dst.sa_family = AF_INET;

    rtable.rtable_genmask.sa_len    = sizeof(struct sockaddr_in);
    rtable.rtable_genmask.sa_family = AF_INET;

    rtable.rtable_gateway.sa_len    = sizeof(struct sockaddr_in);
    rtable.rtable_gateway.sa_family = AF_INET;

    inet_aton("192.168.1.32", &((struct sockaddr_in *)&rtable.rtable_dst->sin_addr);
    inet_aton("255.255.255.0", &((struct sockaddr_in
    *)&rtable.rtable_genmask->sin_addr);
    inet_aton("192.168.2.1", &((struct sockaddr_in
    *)&rtable.rtable_gateway->sin_addr);

    strncpy(rtable.rtable_ifname, "en1", IF_NAMESIZE);

    rtable.rtable_metric = 3;

    iSock = socket(rtable.rtable_dst.sa_family, SOCK_DGRAM, 0);
    if (iSock < 0) {
        return (PX_ERROR);
    }

    /*
     * Add a route using the SIOCADDRT command
     */
    iRet = ioctl(iSock, SIOCADDRT, &rtable);
    if (iRet < 0) {
        fprintf(stdout, "fail to ioctl\n");
        close(iSock);
    }
}
```

```

        return (-1);

    } else {
        close(iSock);
    }

    return (0);
}

```

Run the program in SylixOS Shell, and partial results are displayed as follows:

```

# ./route_ioctl
# route
IPv4 Route Table:
Destination      Gateway          Genmask          Flags Metric Ref    Use Iface
192.168.1.32    192.168.1.1    255.255.255.0   UH  3     0     0 en1
192.168.1.34    0.0.0.0         255.255.255.255 UH  4     0     0 en1
192.168.1.0     0.0.0.0         255.255.255.0   U   4     0     0 en1
127.0.0.1      0.0.0.0         255.255.255.255 UH  4     0     0 lo0
127.0.0.0      0.0.0.0         255.0.0.0       U   4     0     0 lo0

```

Check the routing information through the **route** command, and it can be seen that the route to be set has been added. That is to say, route entry with destination address 192.168.1.32, subnet mask 255.255.255.0, gateway 192.168.1.1, metric 3 and interface of **en1**.

The **rt_msghdr** structure shall be filled to add a route with the write function. Firstly, obtain the file descriptor generated by the routing socket AF_ROUTE. Note that only SOCK_RAW can communicate the bottom. When filling the **rt_msghdr** structure, you need to construct the **msg** structure, the **buf** array of the structure is used to store the destination address, subnet mask and gateway you want to set. It is also worth noting that when setting the destination address, mask and gateway, the corresponding protocol cluster and length also need to be filled.

After the information is filled, the write function is called to add the route.

```

#include <stdio.h>
#include <unistd.h>
#include <net/route.h>
#include <string.h>
#include <socket.h>

#define ROUND_UP(x, align) ((size_t)((((size_t)(x) + (align - 1)) & ~(align - 1)))
#define SO_ROUND_UP(len) ROUND_UP(len, sizeof(size_t))
#define SA_ROUND_UP(x) SO_ROUND_UP(((struct sockaddr *) (x))->sa_len)
#define SA_NEXT(t, x) (t)((PCHAR)(x) + SA_ROUND_UP(x))

```

```
#define SOCKADDRSET(X,R) \
if (msg.rtm.rtm_addr & (R)) \
{ \
    int len = SA_ROUND_UP(X); \
    memcpy(pnt, (caddr_t)(X), len); \
    pnt += len; \
}

struct {
    struct rt_msghdr rtm;
    char buf[512];
} msg;

int main (int argc, char **argv)
{
    int          iSock;
    ssize_t      iRet;
    caddr_t      pnt;
    struct sockaddr_in dest;
    struct sockaddr_in mask;
    struct sockaddr_in gate;

    iSock = socket(AF_ROUTE, SOCK_RAW, 0);
    if (iSock < 0) {
        fprintf(stdout, "fail to socket\n");
        return (-1);
    }
    static int msg_seq = 0;

    memset (&msg, 0, sizeof (struct rt_msghdr));

    msg.rtm.rtm_version = RTM_VERSION;
    msg.rtm.rtm_type = RTM_ADD;
    msg.rtm.rtm_seq = msg_seq++;
    msg.rtm.rtm_addr = RTA_DST | RTA_GATEWAY | RTA_NETMASK;
    msg.rtm.rtm_flags = RTF_HOST | RTF_UP;
    msg.rtm.rtm_index = 1;

    pnt = (caddr_t)msg.buf;

    inet_aton("192.168.1.32", &dest.sin_addr);
    dest.sin_family = AF_INET;
    dest.sin_len = sizeof(struct sockaddr_in);
```

```

    inet_aton("192.168.1.1", &gate.sin_addr);
    gate.sin_family = AF_INET;
    gate.sin_len = sizeof(struct sockaddr_in);

    inet_aton("255.255.255.0", &mask.sin_addr);
    mask.sin_family = AF_INET;
    mask.sin_len = sizeof(struct sockaddr_in);

    SOCKADDRSET (&dest, RTA_DST);
    SOCKADDRSET (&gate, RTA_GATEWAY);
    SOCKADDRSET (&mask, RTA_NETMASK);

    msg.rtm.rtm_msglen = pnt - (caddr_t) &msg;

    iRet = write(iSock, &msg, msg.rtm.rtm_msglen);
    if (iRet < 0) {
        fprintf(stdout, "fail to write error\n");
        return (PX_ERROR);
    }

    return (0);
}

```

Run the program in SylixOS Shell, and partial results are displayed as follows:

```

# ./route_write
# route
IPv4 Route Table:
Destination      Gateway          Genmask          Flags Metric Ref    Use Iface
192.168.1.32     192.168.1.1     255.255.255.0   UH    0    0      0 en1
192.168.1.34     0.0.0.0         255.255.255.255 UH    4    0      0 en1
192.168.1.0      0.0.0.0         255.255.255.0   U     4    0      0 en1
127.0.0.1        0.0.0.0         255.255.255.255 UH    4    0      0 lo0
127.0.0.0        0.0.0.0         255.0.0.0       U     4    0      0 lo0

```

Check the routing information through the **route** command, and it can be seen that the route to be set has been added. That is to say, route with destination address 192.168.1.32, subnet mask 255.255.255.0, gateway 192.168.1.1 and interface of **en1**. Since no metric is set, the default is 0.

6. AF_ROUTE get route message

The implemented function is that when the AF_ROUTE routing socket is created, call to the read function will block until the routing information changes, and the routing information changes in many ways, including plugging and unplugging the network cable, opening or closing the interface, and adding or deleting routes, which can cause changes in routing information.

Here is only an example that read will return all details of the route in time when the route is added. The point to note here is the conversion between pointers.

```
#include <stdio.h>
#include <unistd.h>
#include <net/route.h>
#include <string.h>
#include <socket.h>

struct rtm_type_parse {
    u_char    rtm_type;
    void      (*func)(void *);
};

#define NAMESIZE    2048

#ifndef ARRAY_SIZE
#define ARRAY_SIZE(x)    (sizeof(x) / sizeof((x)[0]))
#endif

#define ROUND_UP(x, align)    (size_t)((((size_t)(x) + (align - 1)) & ~(align - 1)))
#define SO_ROUND_UP(len)    ROUND_UP(len, sizeof(size_t))
#define SA_ROUND_UP(x)    SO_ROUND_UP(((struct sockaddr *) (x))->sa_len)
#define SA_NEXT(t, x)    (t)((PCHAR)(x) + SA_ROUND_UP(x))

#define SET_TAG()
printf("-----\n")

char *num2family[] = {
    [0] = "AF_UNSPEC",
    [1] = "AF_UNIX",
    [2] = "AF_INET",
    [10] = "AF_INET6",
    [16] = "AF_ROUTE",
    [17] = "AF_PACKET",
    [18] = "AF_LINK"
};

int main (int argc, char **argv)
{
    int                iSock;
    ssize_t            iRet;
    char                rBuf[NAMESIZE];
    struct rt_msghdr    *pMy;
```

```
    struct sockaddr_in *psockaddrin;
void          *p;
int           bits = 0;

iSock = socket(AF_ROUTE, SOCK_RAW, 0);
if (iSock < 0) {
    fprintf(stdout, "fail to socket\n");
    return (-1);
}
printf("iSock is %d\n", iSock);
printf("rtentry count is %d\n", (int )sizeof(struct rt_msghdr));

pMy = (struct rt_msghdr *)rBuf;

printf("here is 1\n");

iRet = read(iSock, rBuf, NAMESIZE);
if (iRet < 0) {
    fprintf(stdout, "fail to read\n");
    return (-1);
}

if (pMy->rtm_type == RTM_ADD) {
    if ((pMy->rtm_type != RTM_ADD) ||
        (pMy->rtm_version != RTM_VERSION)) {
        printf("<%s> type[%x] mismatch.\n", "RTM_GET", pMy->rtm_type);
    }

    SET_TAG();

    printf("<%s> rtm_msglen = %d.\n", "RTM_GET", pMy->rtm_msglen);
    printf("<%s> rtm_version = %d.\n", "RTM_GET", pMy->rtm_version);
    printf("<%s> rtm_pid = %d.\n", "RTM_GET", pMy->rtm_pid);
    printf("<%s> rtm_seq = %d.\n", "RTM_GET", pMy->rtm_seq);
    printf("<%s> rtm_errno = %d.\n", "RTM_GET", pMy->rtm_errno);
    printf("<%s> rtm_use = %d.\n", "RTM_GET", pMy->rtm_use);
    printf("<%s> rtm_inits = %lu.\n", "RTM_GET", pMy->rtm_inits);

    SET_TAG();

    p = (void *) (pMy + 1);
    psockaddrin = (struct sockaddr_in *)p;
```

```
        if (pMy->rtm_addrs & RTA_DST) {
            psockaddrin = (struct sockaddr_in *)p;

            printf("<%s> <RTA_DST> net family: %s\n", "RTM_GET",
num2family[psockaddrin->sin_family]);
            printf("<%s> <RTA_DST> net port: %d\n", "RTM_GET",
ntohs(psockaddrin->sin_port));
            printf("<%s> <RTA_DST> dest addr: %s\n", "RTM_GET",
inet_ntoa(psockaddrin->sin_addr));
            bits++;
            SET_TAG();
        }

        if (pMy->rtm_addrs & RTA_GATEWAY) {
            if (bits > 0) {
                psockaddrin = SA_NEXT(struct sockaddr_in *, psockaddrin);
            } else {
                psockaddrin = (struct sockaddr_in *)p;
            }

            printf("<%s> <RTA_GATEWAY> net family: %s\n", "RTM_GET",
num2family[psockaddrin->sin_family]);
            printf("<%s> <RTA_GATEWAY> net port: %d\n", "RTM_GET",
ntohs(psockaddrin->sin_port));
            printf("<%s> <RTA_GATEWAY> dest addr: %s\n", "RTM_GET",
inet_ntoa(psockaddrin->sin_addr));
            bits++;
            SET_TAG();
        }

        if (pMy->rtm_addrs & RTA_NETMASK) {
            if (bits > 0) {
                psockaddrin = SA_NEXT(struct sockaddr_in *, psockaddrin);
            } else {
                psockaddrin = (struct sockaddr_in *)p;
            }

            printf("<%s> <RTA_NETMASK> net family: %s\n", "RTM_GET",
num2family[psockaddrin->sin_family]);
            printf("<%s> <RTA_NETMASK> net port: %d\n", "RTM_GET",
ntohs(psockaddrin->sin_port));
            printf("<%s> <RTA_NETMASK> dest addr: %s\n", "RTM_GET",
inet_ntoa(psockaddrin->sin_addr));
            bits++;
        }
    }
}
```

```

        SET_TAG();
    }

    } else {
        fprintf(stdout, "you need to add others\n");
        return (-1);
    }

    return (0);
}

```

Run the program in SylixOS Shell, and partial results are displayed as follows:

In a terminal execution program, read will enter the blocking state;

```
# ./route_read
```

All information of the route will be output until the route is added at another terminal.

```

-----
<RTM_GET> rtm_msglen = 272.
<RTM_GET> rtm_version = 5.
<RTM_GET> rtm_pid = 8.
<RTM_GET> rtm_seq = 0.
<RTM_GET> rtm_errno = 0.
<RTM_GET> rtm_use = 0.
<RTM_GET> rtm_inits = 0.
-----
<RTM_GET> <RTA_DST> net family: AF_INET
<RTM_GET> <RTA_DST> net port: 0
<RTM_GET> <RTA_DST> dest addr: 192.168.1.32
-----
<RTM_GET> <RTA_GATEWAY> net family: AF_INET
<RTM_GET> <RTA_GATEWAY> net port: 0
<RTM_GET> <RTA_GATEWAY> dest addr: 192.168.2.1
-----
<RTM_GET> <RTA_NETMASK> net family: AF_INET
<RTM_GET> <RTA_NETMASK> net port: 0
<RTM_GET> <RTA_NETMASK> dest addr: 255.255.255.0
-----

```

7. New route command

In SylixOS, the **route** command is used to process information related to network routing. For example, an IPv4 or IPv6 routing information can be added, deleted, modified and acquired through an instruction, or all information of IPv4 or IPv6 route can be obtained by traversing.

[Command format]

```
route [add|del|chg] [-host|-net|dl|gw] [dest] [netmask] [gateway] {metric} [dev]
```

[Common option]

```
add : Add routing table entries  
del : Delete routing table entries  
chg : Modify the metric of the routing table entry  
-host : Host address  
-net : website address  
dest : target address  
netmask : Subnet mask  
gateway : Gateway address  
metric : measure  
dev : Network Device Interface
```

[Instructions for parameters]

```
dl : default , Mainly operate on default routing information  
gw : gateway , Mainly operate on default routing information
```

The following is an example of using the **route** command:

- Add a route to the host (network interface: en1);

```
route add -host 192.168.7.40 mask 255.0.0.0 123.0.0.0 dev en1
```

- Add a route to the network (network interface: en2), and set the metric to 3;

```
route add -net 180.149.132.47 mask 255.0.0.0 12.0.0.0 metric 3 dev en2
```

- Delete a network route;

```
route del -net 180.149.132.47 mask 255.0.0.0
```

- Delete a host route;

```
route del -host 192.168.7.40 dev en1
```

- Change the metric of a host route to 3;

```
route chg -host 192.168.7.40 mask 255.0.0.0 123.0.0.0 metric 3 dev en1
```

15.8.9 netstat

1. netstat command

In SylixOS, the **netstat** command is used to display network-related information, such as network connection, routing table, interface status, multicast member and so on. The command is indicated as follows:

[Command format]

```
netstat {[-wtux --A] -i | [hriqs]}
```

[Common option]

```
-h : Display help information
-r : Display routing table information
-i : Display interface information
-g : Display multicast group member information
-s : Display network status information
-w : Display raw socket information
-t : Display TCP information
-u : Display UDP information
-p : Display PACKET Socket Information
-x : Display UNIX domain socket information
-l : Display all LISTEN status information
-a : Show all socket information
```

[Instructions for parameters]

None

2. Output information meaning

In SylixOS, output after execution of the **netstat** command is as follows:

```
# netstat -a
--UNIX--:
TYPE      FLAG STATUS  SHUTD      NREAD MAX_BUFFER PATH

--PACKET--:
TYPE      FLAG PROTOCOL INDEX MMAP MMAP_SIZE TOTAL  DROP

--TCP LISTEN--:
LOCAL          REMOTE          STATUS  RETRANS  RCV_WND  SND_WND
*:23          *: *            listen    0        0        0
*:21          *: *            listen    0        0        0

--UDP--:
LOCAL          REMOTE          UDPLITE
*:69          *:0             no
*:137         *:0             no
*:161         *:0             no
```

As can be seen from the output results, there are four parts: UNIX, PACKET, TCP and UDP.

- UNIX part;
 - ◆ TYPE: UNIX domain socket type: stream, seqpacket, dgram;

- ◆ FLAG: I/O flag, such as NONBLOCK;
 - ◆ STATUS: current state (only for type stream): none, listen, connect, estab;
 - ◆ SHUTD: current shutdown state: rw, r, w, no;
 - ◆ NREAD: number of valid data bytes (unit bytes);
 - ◆ MAX_BUFFER: maximum size of the receive buffer zone
 - ◆ PATH: UNIX domain file path name.
- PACKET part;
 - ◆ TYPE: PACKET socket type: raw, dgram;
 - ◆ FLAG: I/O flag, such as NONBLOCK;
 - ◆ PROTOCOL: protocol type, as shown in Table 15.3;
 - ◆ INDEX: network interface index;
 - ◆ MMAP: whether mmap is performed;
 - ◆ MMAP_SIZE: size of the mapped memory;
 - ◆ TOTAL: total number of network packets;
 - ◆ DROP: number of discarded network packets.
 - TCP part:
 - ◆ LOCAL: local IP address and port number;
 - ◆ REMOTE: remote IP address and port number;
 - ◆ STATUS: TCP status is shown in Table 15.24.
 - ◆ RETRANS: retransmission count;
 - ◆ RCV_WND: receive window size;
 - ◆ SND_WND: send window size.
 - UDP part.
 - ◆ LOCAL: local IP address and port number;
 - ◆ REMOTE: remote IP address and port number;
 - ◆ UDPLITE: whether it is UDPLITE.

Table 15.24 TCP status

TCP status	Instructions
CLOSED	Initial closed state
LISTEN	Listening state, the server side can receive the connection
SYN_SENT	The client enters this state after sending SYN packet
SYN_RCVD	The server enters this state after receiving SYN packet
ESTABLISHED	Connection has been established
FIN_WAIT_1	The side actively closing connection sends the FIN message, and enters this state (generally difficult to see)
FIN_WAIT_2	Enter the end of the FIN_WAIT_1 state and enter this state when receiving ACK from the other end
CLOSE_WAIT	The other side requests to close the connection. Enter this state after responding to the ACK, and then it can perform other operations such as closing the local
CLOSING	The side initiating sending sends FIN message, and enters the state when receiving FIN message of the other party before receiving ACK message (this situation may be sent when both parties request to close at the same time).
LAST_ACK	After the passive closing sides sends the FIN packet, it finally waits for the other side's ACK packet to enter this state.
TIME_WAIT	In the FIN_WAIT_1 state, the state is entered when the message with FIN flag and ACK flag from the other side is received without entering FIN_WAIT_2 state.

3. Display network interface information

Execute the following command to display the network interface information.

```
# netstat -i
          |RECEIVE                               |TRANSMIT
FACE MTU RX-BYTES RX-OK RX-ERR RX-DRP RX-OVR TX-BYTES TX-OK TX-ERR TX-DRP TX-OVR FLAG
en1: 1500 2031404 25593 0 0 0 1790 27 0 0 0
UBLEth
lo0: 0 648 11 0 0 0 648 11 0 0 0
UL
```

The output information shows all network interfaces and their information in the system. The information includes: network MTU, number of bytes of data received, number of successfully received packets, number of received error packets, number of packets sent, network interface flags and so on. The network interface flags supported in SylixOS are shown in Table 15.25.

Table 15.25 Network interface flag

Flag	Instructions
U	Network interface U p
B	Support broadcast
P	The network interface is P oint-to-point connection
D	Network interface Dhcp is turned on
L	Network interface has been connected (Linkup)
Eth	The network interface is an Ethernet device which supports ARP (Etharp)
G	Network interface supports IGMP

15.8.10 npf

1. Brief introduction to npf

In order to provide protection for the internal network, it is necessary to check the packets passing through the firewall, such as checking the source address and destination address, port address and packet type, and determine whether the packet is a legitimate packet based on the data. If it does not meet the predefined rules, this packet will not be sent to the destination computer. Because the packet filtering technology requires that data packets for internal and external communications must pass through the computer which uses this technology to perform filtering, packet filtering technology is usually used on routers.

The network packet filter (npf) tool provided by SylixOS is a network filter which provides the following filtering rules:

- Filter link layer frame: specified as MAC rule, this rule needs to provide the destination computer's hardware address;
- Filter IP datagram: specified as IP rule, this rule needs to specify a range of IP addresses that you want to filter;
- Filtering transport layer packet: specified as UDP or TCP rule, this rule filters packets for a range of IP addresses and a range of port numbers.

Table 15.26 is the network packet filtering rule currently supported by SylixOS.

Table 15.26 Filter rules

Rules	Instructions	Value
LWIP_NPF_RULE_MAC	Filter Ethernet frame	0
LWIP_NPF_RULE_IP	Filter IP datagram	1
LWIP_NPF_RULE_UDP	Filter UDP data packet	2
LWIP_NPF_RULE_TCP	Filter TCP data packet	3

2. npf function

The following function initializes the SylixOS network packet filter.

```
#include <SylixOS.h>
INT Lw_Inet_NpfInit(VOID);
```

Prototype analysis of Function Lw_Inet_NpfInit:

- For success of the function, return 0. For failure, return -1 and set the error number;

The following function adds a new rule to the network packet filter.

```
#include <SylixOS.h>
PVOID Lw_Inet_NpfRuleAdd (CPCHAR pcNetifName,
                          INT iRule,
                          UINT8 pucMac[],
                          CPCHAR pcAddrStart,
                          CPCHAR pcAddrEnd,
                          UINT16 usPortStart,
                          UINT16 usPortEnd);
```

Prototype analysis of Function Lw_Inet_NpfRuleAdd:

- For success of the function, return the new rule. For failure, return LW_NULL and set the error number;
- Parameter ***pcNetifName*** is the network interface name;
- Parameter ***iRule*** is the corresponding rule, as shown in Table 15.26;
- Parameter ***pucMac*** is MAC address group of no-through traffic;
- Parameter ***pcAddrStart*** is the starting IP address of no-through traffic;
- Parameter ***pcAddrEnd*** is the termination IP address of no-through traffic;
- Parameter ***usPortStart*** is the starting port number of no-through traffic;
- Parameter ***usPortEnd*** is the termination port number of no-through traffic;

Parameters ***pcAddrStart*** and ***pcAddrEnd*** are used for Rules "UDP", "TCP" and "IP", Parameters ***usPortStart*** and ***usPortEnd*** are used for Rules "UDP" and "TCP", and Parameter ***pucMac*** is used for Rule "MAC".

The following function deletes a rule from the network packet filter.

```
INT Lw_Inet_NpfRuleDel (CPCHAR pcNetifName,
                       PVOID pvRule,
                       INT iSeqNum);
```

Prototype analysis of Function Lw_Inet_NpfRuleDel:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter ***pcNetifName*** is the network interface name;
- Parameter ***pvRule*** is the rule handle (NULL means that the rule sequence number is used);
- Parameter ***iSeqNum*** is the rule sequence number.

The following function can bind the network filter to the specified network card and unbind it from the specified network card.

```
#include <SylixOS.h>
INT Lw_Inet_NpfAttach (CPCHAR pcNetifName);
```

```
INT    Lw_Inet_NpfDetach (CPCHAR  pcNetifName);
```

Function prototype analysis:

- For success of the function, return 0. For failure, return -1 and set the error code;
- Parameter *pcNetifName* is the network interface name;

The following function can get network filter information.

```
ULONG  Lw_Inet_NpfDropGet (VOID);
ULONG  Lw_Inet_NpfAllowGet (VOID);
INT     Lw_Inet_NpfShow (INT  iFd);
```

Prototype analysis of Function Lw_Inet_NpfShow:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter *iFd* is the target file descriptor.

Calling the Lw_Inet_NpfDropGet function can get the number of discarded packets (including abandonment caused by regular filtration and insufficient cache), calling the Lw_Inet_NpfAllowGet function can get the number of packets running, and calling the Lw_Inet_NpfShow function can print detailed information of the network filter to the specified file (the file has been opened via the open function).

3. Npf comman

In SylixOS, filtering commands of the network packet include: *npfs*, *npfsattach*, *npfdetach*, *npfruleadd* and *npfruledel*, and instructions for commands are as follows:

The *npfattach* command adds the filtering function to the specified network interface.

[Command format]

```
npfattach [netifname]
```

[Common option]

None

[Instructions for parameters]

```
netifname : Network interface name
```

The following shows how to use the *npfattach* command:

```
# npfattach en1
attached.
```

The *npfdetach* command deletes the filtering function from the specified network interface.

[Command format]

```
npfdetach [netifname]
```


[Common option]

None

[Instructions for parameters]

netifname : Network interface name

The following shows how to use the *npfdetach* command:

```
# npfdetach en1
detached.
```

The *npfruleadd* command adds a new network filtering rule to the system, and the rules currently supported in SylixOS are as shown in Table 15.27.

[Command format]

```
npfruleadd [netifname] [rule] [args...]
```

[Common option]

None

[Instructions for parameters]

```
netifname : Network interface name;
rule : The rule name currently includes: "mac", "ip", "tcp", "udp";
args : It varies according to the rules, as shown in Table 15.27.
```

Table 15.27 Rules and parameters

Rules	Parameters
mac	MAC address, such as 08:08:08:08:08:08
ip	Starting IP address and termination IP address, such as 192.168.1.1 and 192.168.1.100
tcp	Starting IP address, termination IP address, starting port number and termination port number, such as 192.168.1.1, 192.168.1.100, 20 and 30
udp	Starting IP address, termination IP address, starting port number and termination port number

The following shows how to use the ***npfruleadd*** command:

```
# npfruleadd en1 ip 192.168.1.1 192.168.1.200
rule add ok
```

The ***npfruledel*** command deletes an existing rule.

[Command format]

```
npfruledel [netifname] [rule sequence num]
```

[Common option]

None

[Instructions for parameters]

```
netifname : Network interface name;
rule sequence num : Rule number.
```

The following shows how to use the ***npfruleadd*** command:

```
# npfruledel en1 0
delete
```

The ***npfs*** command indicates the filter rule information currently added.

[Command format]

```
npfs
```

[Common option]

None

[Instructions for parameters]

None

The following is the print information of the ***npfs*** command:

```
# npfs
NETIF ATTACH SEQNUM RULE ALLOW MAC      IPs      IPe      PORTs    PORTe
```

```

en1    YES    0 IP    NO     N/A    192.168.1.1 192.168.1.200 N/A    N/A
drop:82 allow:1306

```

It can be seen from the information indicated from the ***npfs*** command that a filter rule of “ip” type is added in SylixOS at present. The network interface is "en1", the sequence number is 0, the starting IP address of the filtering rule is 192.168.1.1, and the termination IP address is 192.168.1.200. Because the type of the filter rule is "ip", the start port number and termination port number do not exist.

15.9 Control interface of the standard network card

SylixOS supports the control function of the POSIX standard network card, the application can conveniently modify the behavior of the network interface via these functions. For example, activate the network card, turn off the network card, enable DHCP and so on.

In addition, the IP address and network card flag (such as enabling the promiscuous mode) can be got and set via the network ioctl command.

1. Network interface API

```

#include <net/if.h>
int if_down(const char *ifname);

```

Prototype analysis of Function if_down:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***ifname*** is the network interface name.

Calling the if_down function can turn off the specified network device. If the DHCP lease is enabled on the network interface, the DHCP lease will be disabled simultaneously.

```

#include <net/if.h>
int if_up(const char *ifname);

```

Prototype analysis of Function if_up:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***ifname*** is the network interface name.

Calling the if_up function can activate the specified network device. If the DHCP flag is set, DHCP lease will be enabled simultaneously.

```

#include <net/if.h>
int if_isup(const char *ifname);

```

Prototype analysis of Function if_isup:

- For success of the function, return 0 represents that the network card is not enabled, and 1 represents that the network card is enabled. For failure, return -1 and set the error number;
- Parameter ***ifname*** is the network interface name.

Calling the `if_isup` function can detect whether the specified network interface is at activated state.

```
#include <net/if.h>
int if_islink(const char *ifname);
```

Prototype analysis of Function `if_islink`:

- For success of the function, return 0 represents that the network card is not connected, and 1 represents that the network card is connected. For failure, return -1 and set the error number;
- Parameter ***ifname*** is the network interface name.

Calling the `if_islink` function can detect whether the specified network interface is connected. It shall be noted that the network interface packet can be sent after the activated network interface is successfully connected.

```
#include <net/if.h>
int if_set_dhcp(const char *ifname, int en);
int if_get_dhcp(const char *ifname);
```

Prototype analysis of Function `if_set_dhcp`:

- For success of the function, return 0. For failure, return -1 and set the error number;
- Parameter ***ifname*** is the network interface name.
- Parameter ***en*** is divided into two values: 0 represents disabling the DHCP lease, and 1 represents enabling the DHCP lease.

Prototype analysis of Function `if_get_dhcp`:

- For success of the function, get DHCP state. 0 represents disable, and 1 represents enable. For failure, return -1 and set the error number;
- Parameter ***ifname*** is the network interface name.

Calling the `if_set_dhcp` function can enable or disable the DHCP lease the specified network interface, and calling the `if_get_dhcp` function will get the DHCP state.

```
#include <net/if.h>
unsigned if_nametoindex(const char *ifname);
char *if_indextoname(unsigned ifindex, char *ifname);
```

Prototype analysis of Function `if_nametoindex`:

- The function returns the index number of the network interface;

- Parameter **ifname** is the network interface name.

Prototype analysis of Function if_indextoname:

- For success of the function, return the network interface name (such as en1). For failure, return LW_NULL;
- Parameter **ifindex** is the network interface index;
- Out parameter **ifname** returns the network interface name.

Calling the if_nameindex function will return the index number corresponding to the network interface **ifname**, calling the if_indextoname function will get the network interface name, the network interface name is stored in the buffer zone referred to by the parameter **ifname**, and the size of the buffer zone shall be at least IF_NAMESIZE (<net/if. h>).

```
#include <net/if.h>
struct if_nameindex *if_nameindex(void);
void if_freenameindex(struct if_nameindex *ptr);
```

Prototype analysis of Function if_nameindex:

- For success of the function, return the network interface structure array pointer. For failure, return NULL and set the error number;

Prototype analysis of Function if_freenameindex:

- Parameter **ptr** is the pointer returned by the if_nameindex function.

Calling the if_nameindex function will return the array of the if_nameindex structure, the array contains all local network interfaces and is allocated dynamically, and the memory can be released by calling the if_freenameindex function. If_nameindex structure, as shown below:

```
struct if_nameindex {
    unsigned    if_index;           /* Numeric index of interface      */
    char        *if_name;          /* Null-terminated name of the    */
                                   /* interface.                      */
    .....
};
```

- If_index: interface index;
- If_name: the network interface name ending with the null character, such as en1.

The following program prints all local network interfaces:

Program List 15.17 Print network interface

```
#include <net/if.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
```

```

struct if_nameindex *if_ni, *p;

if_ni = if_nameindex();
if (if_ni == NULL) {
    fprintf(stderr, "if_nameindex error.\n");
    return (-1);
}

for (p = if_ni; !(p->if_index == 0 && p->if_name[0] == '\0'); p++) {
    fprintf(stdout, "%u: %s\n", p->if_index, p->if_name);
}

if_freenameindex(if_ni);
return (0);
}

```

Run under the SylixOS Shell, and the results are as follows:

```

# ./if_test
1: en1
0: lo0

```

2. Network interface ioctl

The traditional ioctl function is used for any featured system interface which is common but not suitable to be classified in other categories. However, POSIX removes ioctl, and POSIX replaces ioctl with some standardized wrapper functions. For example: the function to operate the serial port: tcgetattr function, tcflush function and so on. Nevertheless, network programming still retains ioctl operation, so as to adapt to specific operations. For example, get the network interface information, set the network card flag and so on.

The first step of the network interface program is usually to get all network interfaces configured in the system from the kernel, which is implemented via request by the SIOCGIFCONF command. This command uses the ifconf structure, and the ifconf structure uses the ifreq structure. These two structures are as follows (both structures are defined in <net/if.h>):

```

struct ifreq {
#define IFHWADDRLEN      6
    union {
        char          ifrn_name[IFNAMSIZ];
    } ifr_ifrn;
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
    } ifr_ifru;
};

```

```
        struct sockaddr ifru_netmask;
    struct sockaddr ifru_hwaddr;
    short          ifru_flags;
    int            ifru_ifindex;
    int            ifru_mtu;
    int            ifru_metric;
    int            ifru_type;
    void           *ifru_data;
} ifr_ifru;
};
#define ifr_name          ifr_ifrn.ifrn_name
#define ifr_addr          ifr_ifru.ifru_addr
#define ifr_dstaddr      ifr_ifru.ifru_dstaddr
#define ifr_netmask      ifr_ifru.ifru_netmask
#define ifr_broadaddr    ifr_ifru.ifru_broadaddr
#define ifr_hwaddr       ifr_ifru.ifru_hwaddr
#define ifr_flags         ifr_ifru.ifru_flags
#define ifr_ifindex      ifr_ifru.ifru_ifindex
#define ifr_mtu          ifr_ifru.ifru_mtu
#define ifr_metric       ifr_ifru.ifru_metric
#define ifr_type         ifr_ifru.ifru_type
#define ifr_data         ifr_ifru.ifru_data
```

For the application, the meaning of the following members shall be understood:

- ifr_name: network interface name (such as: en1);
- ifr_addr: network interface address;
- ifr_dstaddr: destination address;
- ifr_netmask: subnet mask;
- ifr_broadaddr: broadcast address;
- ifr_hwaddr: hardware address;
- ifr_flags: network flag, as shown in Table 15.28;
- ifr_ifindex: network interface index;
- ifr_mtu: network MTU;
- ifr_metric: network measurement;
- ifr_type: network type;
- ifr_data: request date.

Table 15.28 Network interface flag

Entrance flag	Instructions
IFF_UP	Network equipment activated
IFF_BROADCAST	The broadcast address of the network device is valid
IFF_POINTOPOINT	The network devices is the point-to-point mode
IFF_RUNNING	Network equipment connected
IFF_MULTICAST	The network device supports IGMP
IFF_LOOPBACK	Loopback device
IFF_NOARP	The network interface does not support the APR protocol
IFF_PROMISC	Network settings support the promiscuous mode

```

struct ifconf {
    int          ifc_len;          /* size of buffer in bytes    */
    union {
        char     *ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
};
#define ifc_buf      ifc_ifcu.ifcu_buf /* buffer address            */
#define ifc_req      ifc_ifcu.ifcu_req /* array of structures       */

```

The following is the meaning of the ifconf structure member:

- ifc_len: buffer zone length;
- ifc_buf: buffer zone pointer;
- ifc_req: ifreq structure pointer.

Allocate a buffer zone and an ifconf structure before calling the ioctl function, and then initialize the ifconf structure. Figure 15.36 shows the memory layout of ifconf, and the third parameter of the ioctl function points to the ifconf structure. The ifreq structure returned by the kernel will be stored in the buffer zone where the ifc_buf points to shown in Figure 15.36, and ifc_len is updated to reflect the actual number of existing bytes in the buffer zone.

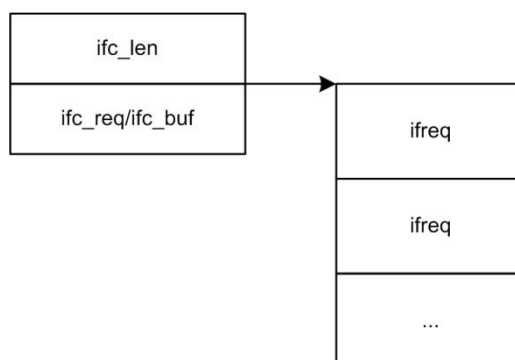


Figure 15.36 Relation between the ifconf structure and ifreq structure.

3. Network ioctl command

It is introduced above that different operations shall be performed via the ioctl command request in network programming, and the following commands are included in SylixOS:

- The SIOCGIFCONF command can get the list of local network interfaces. It shall be noticed that these network interfaces are AF_INET address family (IPv4), and the system will return the list of network interfaces with the ifconf structure. If the length returned by ifc_len is equal to the original incoming length, the size of the ifc_buf buffer shall be increased;
- The SIOCGSIZIFCONF command can get the buffer zone size required for the list of the network interface;
- The SIOCSIFFLAGS command can set the network flag, as shown in Table 15.28. The network card flag is set via the ifr_flags member of the ifreq structure;
- The SIOCGIFFLAGS command can get the network flag, as shown in Table 15.28. The network card flag is got via the ifr_flags member of the ifreq structure;
- The SIOCGIFTYPE command can get network interface types, which are usually defined in <net/if_type.h> as IFT_* types, such as IFT_PPP, IFT_LOOP. These network types are got via the ifr_type member of the ifreq structure;
- The SIOCGIFINDEX command can get the index value of the network interface. The network interface index is got via the ifr_ifindex member of the ifreq structure.
- The SIOCGIFMTU command can get the MTU value of the network interface. The network MTU is got via the ifr_mtu member of the ifreq structure (SylixOS does not support setting the MTU value from the application at present);
- The SIOCGIFHWADDR command can get the hardware address of the network card. The hardware address is got via the ifr_hwaddr member of the ifreq structure, the member is of the sockaddr structure type, the sa_data member in the structure stores the hardware address, and sa_family is of the ARPHRD_ETHER type.
- The SIOCSIFHWADDR command can set the hardware address of the network card, which usually requires hardware-driven support;
- The SIOCSIFADDR command can set the IP address of the network card (the command is valid for IPv4);
- The SIOCSIFNETMASK command can set the network mask (the command is valid for IPv4);
- The SIOCSIFDSTADDR command can set the destination IP address (the command is valid for IPv4);
- The SIOCSIFBRDADDR command can set the network broadcast address (the command is valid for IPv4);

- The SIOCGIFADDR command can get the IP address of the network card (the command is valid for IPv4);
- The SIOCGIFNETMASK command can get the network mask (the command is valid for IPv4);
- The SIOCGIFDSTADDR command can get the destination IP address (the command is valid for IPv4);
- The SIOCGIFBRDADDR command can get the network broadcast address (the command is valid for IPv4);
- The SIOCGIFNAME command can get the network interface name.

The following example shows how to get the local network interface information via the network `ioctl`. The program calls the `ioctl` function to use the `SIOCGIFCONF` command to get all local network interfaces and print the interface name and IP address.

Program List 15.18 Network interface information

```
#include <stdio.h>
#include <sys/socket.h>
#include <net/if.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <netdb.h>

void show_netif_msg (struct ifreq *ifreq)
{
    struct sockaddr_in *ip4;
    char ipbuf[20];

    fprintf(stdout, "-----\n");
    fprintf(stdout, "netif name: %s\n", ifreq->ifr_name);

    ip4 = (struct sockaddr_in *)&ifreq->ifr_addr;
    fprintf(stdout, "source addr: %s\n",
            inet_ntop(AF_INET, (void *)&(ip4->sin_addr), ipbuf, 20));
}

int main (int argc, char *argv[])
{
    int fd;
    struct ifconf iconf;
    char buf[512];
    int ret, i;
```

```
fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0) {
    fprintf(stderr, "socket error.\n");
    return (-1);
}

iconf.ifc_len = 512;
iconf.ifc_buf = (char *)buf;

ret = ioctl(fd, SIOCGIFCONF, (void *)&iconf);
if (ret < 0) {
    fprintf(stderr, "ioctl error.\n");
    return (-1);
}

if (iconf.ifc_len >= 512) {
    fprintf(stderr, "ifconf buffer overflow.\n");
    return (-1);
}

for (i = 0; i < iconf.ifc_len / sizeof(struct ifreq); i++) {
    show_netif_msg(&iconf.ifc_req[i]);
}

return (0);
}
```

Run the program under the SylixOS Shell, and the results are as follows:

```
# ./ifconf_test
-----
netif name: en1
source addr: 192.168.1.32
-----
netif name: lo0
source addr: 127.0.0.1
```

15.10 Brief introduction to wireless communications and ad-hoc network

The IEEE802.11 standard is adopted for majority of wireless networks. The basic wireless network includes multiple radio broadcast stations in the 2.4 GHz or 5 GHz band. (However, the specific frequency will change in the range of 2.3GHz to 4.9GHz with different regions or for better communication)

The 802.11 network has two organization modes: in the infrastructure mode, a communication station is taken as the master station, and other communication stations are associated with it; such network is called BSS, and the master station becomes the wireless access point (AP). In BSS, all communication is completed via AP; even if communication stations need to communicate with each other, the message must be sent to AP. In the second form of network, the master station does not exist, and communication stations communicate directly. The form of network is called IBSS, and ad-hoc network usually.

The 802.11 network was initially deployed in the 2.4 GHz band, and the protocols defined by the IEEE 802.11 and 802.11b standards are adopted. These standards define the operation frequency adopted and characteristics of the MAC layer including the subframe and transmission rate (different rates can be used during communication). The later 802.11a standard defines the use of the 5GHz band for operation, as well as different signaling mechanisms and higher transmission rates. The 802.11g standard defined later enabled how to use 802.11a signals and transport mechanisms at 2.4 GHz, so as to provide forward compatibility with the earlier 802.11b network.

Various underlying transport mechanisms adopted in the 802.11 network provide different types of security mechanisms. The original 802.11 standard defined a simple security protocol called WEP. This protocol adopts a fixed pre-release key, and uses RC4 encryption algorithm to encode the data transmitted on the network. All communication stations must use the same fixed key to communicate. This pattern has proved to be easily broken, and is therefore rarely used at present. Using this method only allows the users who access the network to quickly disconnect. The latest security practice is given by the IEEE 802.11i standard, which defines the new encryption algorithm and uses an additional protocol to allow the communication station to verify the identity to the wireless access point and exchange keys for data communication. Furthermore, the key used for encryption will be refreshed periodically, and there is a mechanism which can monitor the intrusion attempt (and prevent such attempt). Another common security protocol standard in wireless networks is WPA. This is a transitional standard defined by the industry organization prior to 802.11i. WPA defines a subset required and specified in 802.11i, and is designed to be implemented on legacy hardware. In particular, WPA requires that only the TKIP encryption algorithm derived from the algorithm adopted by original WEP be used. 802.11i not only allows use of TKIP, but also requires support for stronger encryption algorithm AES-CCM to encrypt data. (The AES encryption algorithm is not required in WPA because of high computational complexity required to implement this algorithm on legacy hardware).

In addition to the protocol standards described earlier, there is another standard to be introduced is 802.11e. It defines the protocol for running multimedia applications on 802.11 network, such as video streaming and voice transmitted with IP (VoIP). Similar to 802.11i, 802.11e also has a predecessor standard, commonly referred to as WME (later renamed as WMM), which is also the subset of the 802.11e defined by the industry organization, in order to use multimedia applications in legacy hardware. Another important difference between 802.11e and WME/WMM is that the former allows traffic to be prioritized through quality of service (QoS) protocol and enhanced media access protocol. For the correct implementation of these protocols, high-speed burst data and traffic classification can be achieved.

SylixOS supports the networks adopting 802.11a, 802.11b and 802.11g. Similarly, she also supports WPA and 802.11i security protocols (in conjunction with 11a, 11b, and 11g), while QoS and traffic grading required by WME/WMM provide support on some wireless devices.

SylixOS not only supports traditional wired network communication models such as Ethernet and a variety of industrial buses. SylixOS also supports the new-generation wireless communication networking technology: Mesh (wireless mesh network). The wireless mesh network is also called an "ad-hoc" network. It is a new type of wireless network technology which is completely different from BSS network. In the BSS network, each client accesses the network through a wireless link connected to AP. If the nodes want to communicate with each other, they must first access a fixed access point (AP), which is also called as hotspot. The network structure is called as single-hop network. All nodes in the network must be within the range which AP can communicate with. However, in the wireless mesh network, any wireless device node can be taken as both AP and router. Each node in the network can send and receive the signal. Each node can communicate directly with one or more peer nodes.

Compared to BSS network, it is only required to simply connect the power supply to add the new device. It can automatically configure itself, and determine the optimal multi-hop transmission path. When adding or moving the device, the network can automatically discover topology changes, and automatically adjust communication routes to get the most efficient transmission path.

Compared with the BSS network, the wireless Mesh network has several unparalleled advantages:

- Due to adoption of dynamic topology technology, Mesh network deployment and installation are very simple and efficient. Remove the device from the box, and connect it to the power supply. As installation is greatly simplified, users can easily add new nodes to expand the wireless network coverage and network capacity;
- Since Mesh uses an automatic multi-hop routing mechanism, it is easy to implement NLOS. The signal can automatically select the best path to jump from one node to another, and eventually reach the target node without direct line of sight;
- Because the Mesh network does not have a central node, the entire network is very robust and resistant to destruction. Because it does not depend on performance of a single node. In a single-hop network, if one node fails, the entire network will collapse. In the mesh network structure, each node has one or more paths for data transmission. If the nearest node fails or is disturbed, the data packet will be automatically routed to the standby path to continue transmission, and the entire network operation will not be affected;
- Because the Mesh network adopts the dynamic topology, the structure is very flexible. In the multi-hop network, the device can connect to the network through different nodes at the same time, so it will not lead to decrease in system performance.

SylixOS currently supports relatively complete wired and wireless communication framework, and can support multiple types of communication networks simultaneously. They can be uniformly abstracted into the common TCP/IP network interface for extremely easy to access the Internet. The above features make it easy for nodes using SylixOS as the underlying operating system to access the Mesh network.

Figure 15.37 is the SylixOS wireless communication network framework.

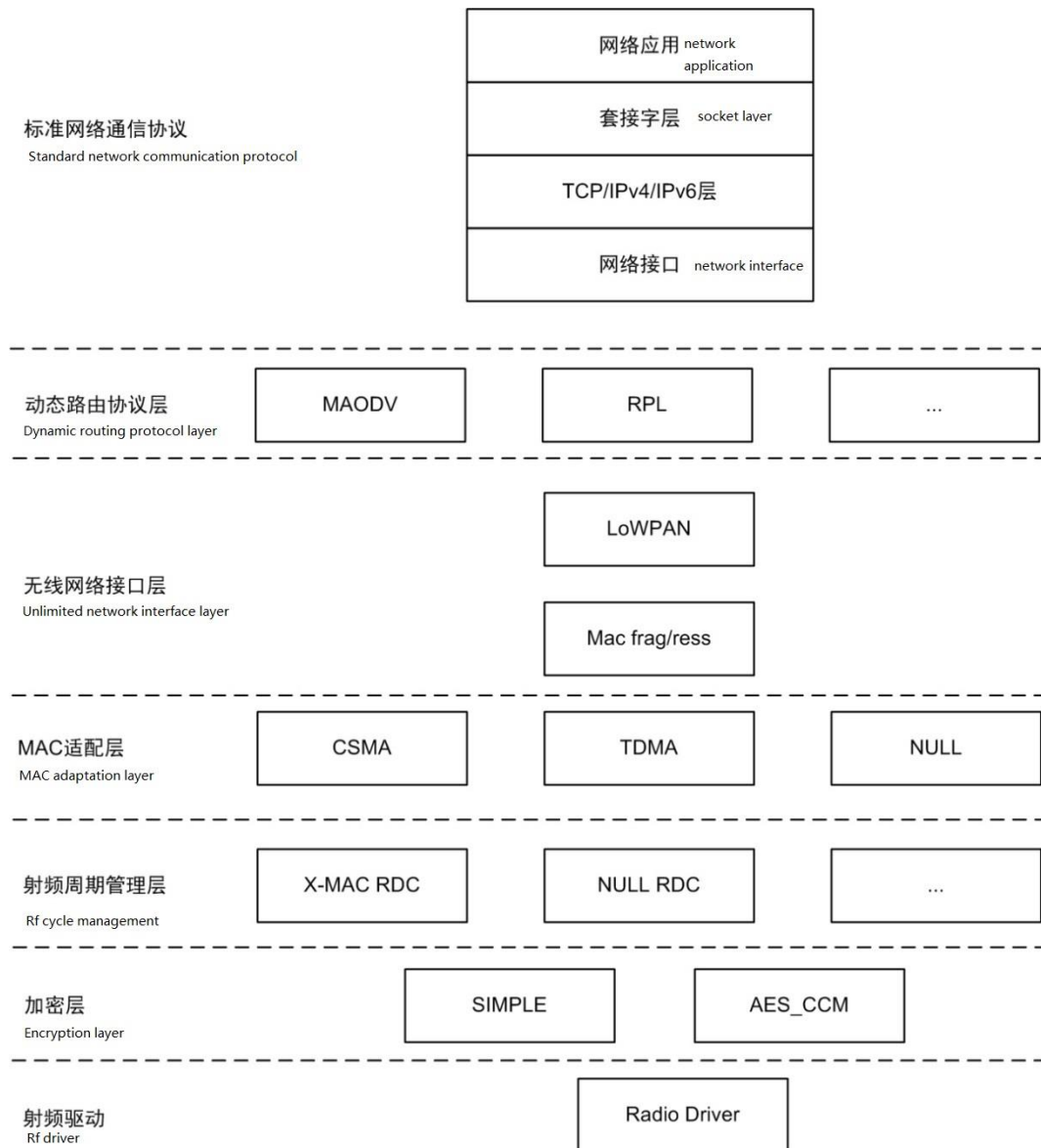


Figure 15.37 Wireless communication network framework

SylixOS's wireless network can be divided into 7 layers, including:

- **Standard communication protocol layer:** wired and wireless network systems do not have any difference here. Therefore, the applications above it can work seamlessly on the wired and wireless networks. There is no difference between them.

- **Dynamic routing protocol layer:** this layer implements the automatic networking function of the wireless network. The currently supported protocol is MAODV. This protocol is applicable to medium-sized network with rapid changes in contact points. The RPL protocol obtained the RFC standard number (RFC 6550) in 2012 and became a standard route exchange protocol. It is mainly used in IPv6 low-power wireless communication systems.
- **Wireless network interface layer:** SylixOS provides a standard IEEE 802.15.4 wireless network interface packet switching function. In order to reduce collision, the wireless network generally use shorter data packet, but IPv6 specifies that the MTU of the network interface shall not be less than 1280 bytes. Therefore, here is also responsible for the data packet split and merge functions based on MAC layer. RF interface which does not comply with the IEEE 802.15.4 standard can be virtualized into such interfaces in RF drives;
- **MAC adaptation layer:** according to the type of wireless network selected, SylixOS provides a variety of MAC access models: they are CSMA/CA (Carrier Sense Multiple Access / Conflict Avoidance), TDMA (Time Division Multiplexing), etc.;
- **Radio frequency cycle management layer:** according to the application model of the network, multiple management models for device work cycles are provided, such as low-power X-MAC model, NULL model for high throughput and real-time performance, and the like;
- **Encryption layer:** the encryption layer provides support for wireless network encryption. SylixOS includes SIMPLE and AES_CCM.
- **RF Drive:** SylixOS provides a set of standard specifications for RF interface (radio_driver) operation. All kinds of RF interface drivers can support Mesh network as long as they comply with this specification.



Chapter 16 File system

16.1 Introduction of the file system

SylixOS offers a variety of standard file systems for user convenience. These file systems are SylixOS built-in systems, if more file systems such as NTFS are needed, they can be added via kernel modules. The SylixOS file system is actually a set of virtual device drivers. It provides two sets of API interfaces that are compliant with the I/O system standards and require device drivers to comply with the block device standards. The structure of the file system in the I/O system is shown in figure 16.1.

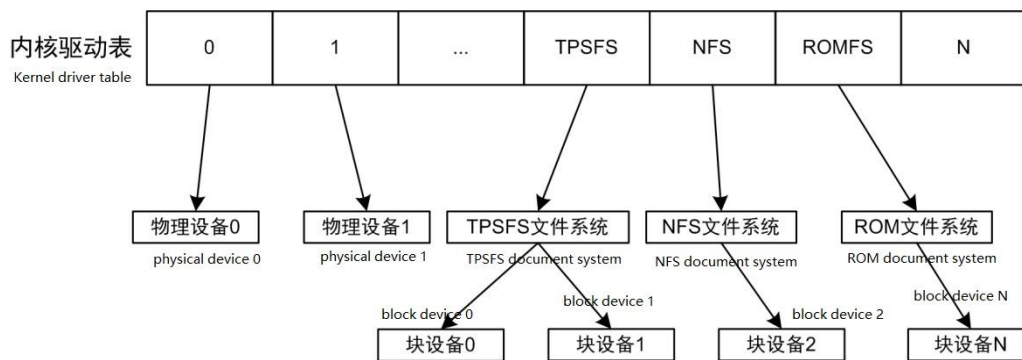


Figure 16.1 Brief structure of the file system

The SylixOS file systems are mounted using standard interfaces provided by the I/O system and access via standard I/O operation functions. In other words, there is no difference between operating a normal file and operating a device file.

SylixOS currently includes the following built-in file systems:

- TPSFS file system;
- ROOT file system;
- PROC file system;
- RAMFS file system;
- FAT file system;
- YAFFS file system;
- NFS file system;
- ROMFS file system.

The ROMFS file system is a read-only file system, and the key files of the system can be placed in this file system to ensure security. When mounting file systems via *mount*, the FAT, NFS, and YAFFS can also be mounted as read-only file systems.

PROC is a file system that holds operating system information and process information. The file entities corresponding to this file system are all in the operating system kernel and are the operating parameters and information fed back by the kernel. For example, the ID of each process has a corresponding directory, which contains the current running information of the process, such as the executable file corresponding to the process, the file descriptor table opened by the process, the memory information consumed by the process, and the internal dynamic link library information of the process etc. All SylixOS internal devices (including file systems) must be mounted on the root file system. The root file system has a very special device name — “/”. All device or file absolute paths start with the root symbol. That is, the operating system always starts with the root file system when querying a device.

SylixOS also provides some components for easy applications of the file system, including the disk partition checking tool, the disk buffer, the disk automounting tool, etc.

The disk partition checking tool can automatically check the partition status of a disk and generate logical devices for the corresponding partition, each of which can be mounted on the file system. The disk buffer is a special block device that between the file system and the disk. Since the disk is a low-speed device that reads and writes at much lower speed than the CPU, SylixOS provides a buffer for the corresponding block device to address this speed mismatch, and this buffer can also greatly reduce the disk I/O access rate while improving system performance. The principle of introducing a disk buffer is the same as that of a CPU, so it will result a short time inconsistency between the data in the memory and the disk, but this problem can be solved by calling the sync function, fsync function, or fdatasync function. The sync function will block the current thread, then write all the data that needs to be written back from the cache to the device and return it. The fsync function indicates that the cached data of the specified file is synchronized to the physical disk, and the fdatasync function indicates that the data portion of the specified file is synchronized to the physical disk.

The disk automounting tool is a collection of tools that encapsulate many disk tools. The device can transfer the physical disk block device to the disk automounting tool through a hot plug event. This tool will first create a disk buffer for this disk, then automatically perform disk partition check and generate a virtual block device for each partition. Finally, it will identify the file system type of each partition and load the corresponding file system. As a result, the user can see the corresponding mounted file system directory in the operating system directory. The SylixOS block device structure with the disk buffer and the partition processing tool is shown in figure 16.2.

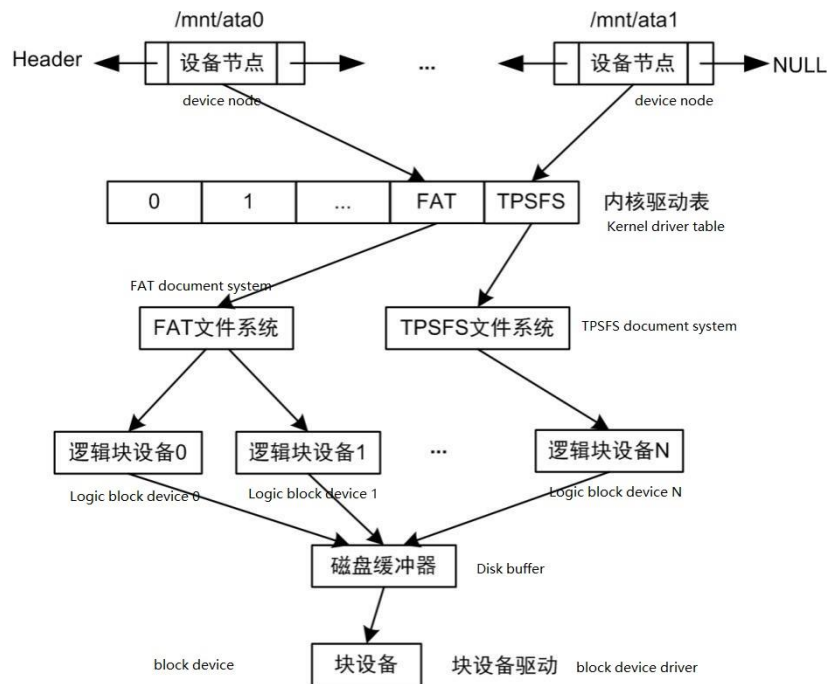


Figure 16.2 Schematic of the block device

16.2 TPSFS file system

TPSFS (True Power Safe FS) is a power-fail safe file system and is also a SylixOS built-in file system. Its structure is shown in Figure 16.3. TPSFS is a transaction-based B+ tree file system: use transaction commit mechanism to modify metadata to ensure the consistency of the file system; use B+ tree to manage the disk space and the file space, and record the starting block and the number of blocks in B+ tree with different key values; use the file starting block number to indicate the inode number, and use the space management B+ tree to implement the inode management. The above principles make the TPSFS file system has the following features:

- B+ tree file data storage improves the space management efficiency while accelerating reading, writing, and positioning speeds;
- Atomic operations on metadata ensures power-fail safety;
- 64-bit file system supports EP-level file length;
- Excellent large file processing performance;
- File record locks support large databases;
- Multi-block allocation mechanism improves system performance and allows the dispenser to have ample room for optimization;
- Subdirectory extensibility allows unlimited number of subdirectories to be created in one directory.

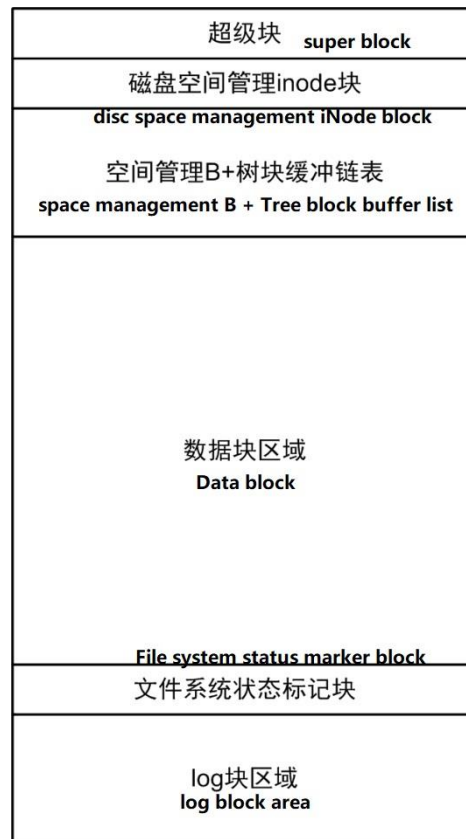


Figure 16.3 Structure of TPSFS file system

Superblock is the first block of TPSFS, which records the basic information of the file system, such as: block size, number of blocks, data block position, and log block position, etc. Each data block in TPSFS is recorded in a B+ tree rooted at the inode block. The structure is shown in figure 16.4.

After the superblock is the space management inode block, the B+ tree corresponding to the inode manages the free blocks of the entire disk, which can be understood as forming all the free block records into the space management inode to form a large file. Different from the ordinary file, the key value of the space management inode B+ tree node is the physical block number of the disk block interval, and the key value of the ordinary file inode B+ tree node is the offset of the block interval in the file.

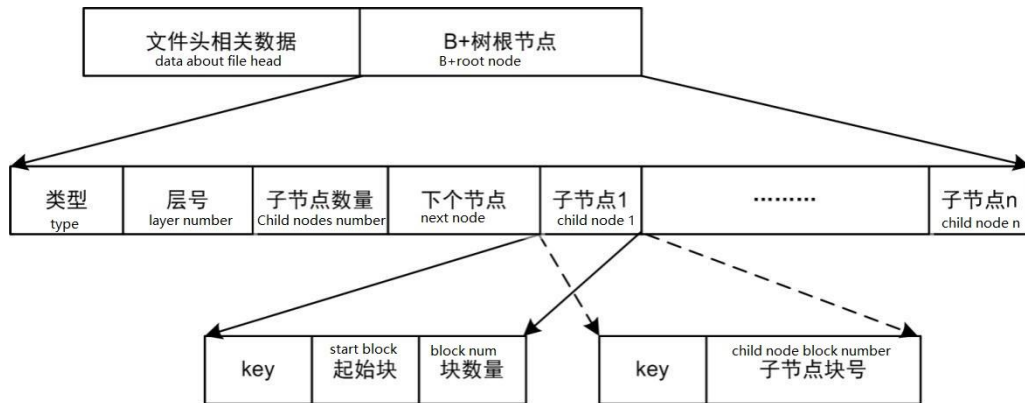


Figure 16.4 Structure of the B+tree nodes

As can be seen from figure 16.4, each B+ tree node contains several subitems whose values are determined by the node type. If it is a leaf node, it contains a key, a starting block, and a block number for recording a physical block interval. If it is a non-leaf node, the key and the corresponding node number of the child node are recorded.

16.3 FAT file system

FAT (File Allocation Table) is a Windows-compatible file system. The removable storage devices (USB flash disk and SD card, etc.) commonly used in SylixOS systems are often mounted in the FAT file system format.

FatFs is a universal FAT file system module designed for embedded systems. FatFs is written in compliance with ANSI C and is completely separate from the disk I/O layer. It is independent (not depend) on the hardware architecture.

The main features of the FatFs file system:

- A Windows-compatible FAT file system;
- Not independent on the platform and easy to transplant;
- The code and workspace occupy very little space;
- Multiple configuration options.

16.3.1 FAT command

In SylixOS, the **fatuid** command can be used to view and set the default uid and gid for a volume label.

[Command format]

```
fatuid [uid] [gid]
```

[Common option]

None

[Instructions for parameters]

```
uid : new user ID
```

```
gid : new group ID
```

See the example content of the default uid and gid for the specified volume label:

```
# fatugid
```

```
vfat current uid : 0 gid : 0
```

16.4 NFS file system

NFS, namely Network File System, is a distributed file system protocol released by Sun in 1984. It is also one of the file systems supported by FreeBSD, which allows network computers to share resources over the TCP/IP network. In NFS applications, a local NFS client application can transparently read and write files located on a remote NFS server just as if they were local files. NFS is a successful file sharing method, but the biggest problem is that it is not suitable for large distributed systems.

The main features of NFS:

- Local workstations use less disk space because the usual data can be stored on a single computer and accessed over the network;
- Users do not have to keep the same directory on every network computer. The same directory can be placed on the NFS server and is available everywhere on the network;
- Storage devices such as floppy disk drives and CDROMs can be used on the network by other computers, which can reduce the number of removable media devices on the entire network.

In the development and debugging phase of an embedded device, this technology can be used to create an NFS-based file system on the host and mount it on the embedded device to facilitate file sharing between the host and the embedded device.

16.4.1 Basic operations of NFS

In the SylixOS system, the file system types supported by the current system can be determined by viewing the `/proc/fs/fssup` file.

```
# cat /proc/fs/fssup
```

```
rootfs procfs ramfs romfs vfat nfs yaffs
```

Take FreeNFS as an example to illustrate the NFS mount operation of SylixOS. Run FreeNFS software and set the parameters, after right-clicking the software icon and selecting Settings, the interface shown in figure 16.5 will pop up.

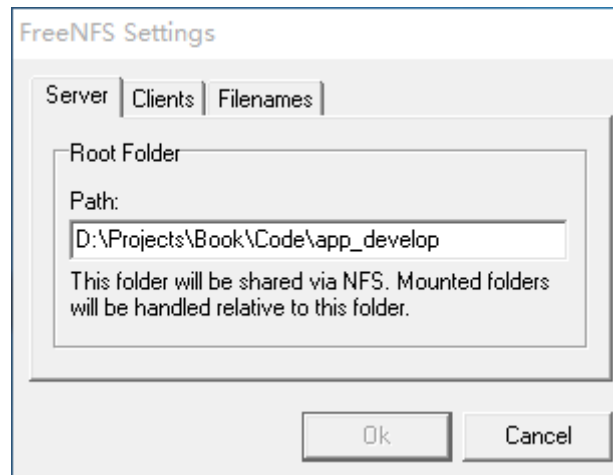


Figure 16.5 FreeNFS settings

Enter the path that the server (PC running FreeNFS) needs to share in the Path field in Server. The empty Allowed Host field in Clients indicates that all clients can connect to the server. Codename in FileNames indicates the encoding format of the current system file name.

Use **showmount** command to view the devices that are currently mounted on the system.

```
# showmount

AUTO-Mount point show >>

      VOLUME                BLK NAME
-----
/media/hdd0                /dev/blk/hdd0:0

Mount point show >>

      VOLUME                BLK NAME
-----

/tmp                        0
```

Use **mount** command to mount NFS file system. The nfs in /mnt/nfs is dynamically created and does not need to be manually created.

```
# mount -t nfs 192.168.1.15:/ /mnt/nfs
```

```

# showmount

AUTO-Mount point show >>

      VOLUME                BLK NAME
-----
/media/hdd0                /dev/blk/hdd0:0

Mount point show >>

      VOLUME                BLK NAME
-----
/mnt/nfs                   192.168.1.15:/
/tmp                       0

# ls /mnt/nfs/

.metadata      app_proc      base_armv4      bsp_micro2440  bsp_mini2440
RemoteSystemsTempFiles

```

Use **umount** command to unmount NFS file system.

```

# umount /mnt/nfs/

# showmount

AUTO-Mount point show >>

      VOLUME                BLK NAME
-----
/media/hdd0                /dev/blk/hdd0:0

Mount point show >>

      VOLUME                BLK NAME
-----

/tmp                       0

```


16.5 ROM file system

ROMFS is a relatively simple and space-saving read-only file system with small size, high reliability, and fast read speed. It supports directories, symbolic links, hard links (SylixOS does not currently support hard links) and device files.

ROMFS is a read-only file system that uses sequential storage. All data, including directories and links, are stored in the order of the directory tree. Therefore, ROMFS is very space-saving compared to other large file systems. It is usually used as the root file system in the embedded device, or used to save the bootloader to boot the system.

The sequential storage of ROMFS makes all the data stored in order, so the data in ROMFS cannot be modified once it is determined. This data storage method causes ROMFS to be unable to be written, which is why ROMFS can only be a read-only file system. Due to the sequential storage, the data of each file in the ROMFS can be stored continuously. In the reading process, only one addressing operation is needed to read the entire block of data, so the ROMFS data reading efficiency is high.

SylixOS provides support for ROMFS, which can mount ROM file system via Shell like other ones. SylixOS supports two ROMFS mounting methods: one is to mount a block device that conforms to the ROMFS format in the `/dev/blk` directory so that read operations can be performed at the application level via I/O functions (note that write operations cannot be performed); Another one is to mount a ROMFS-compliant image file, but this requires the use of other tools (like `genromfs`) to make a ROMFS image. The following is the process of mounting a ROMFS image file:

- Compile the `genromfs` tool (this tool comes from the network);
- Generate the image file by executing the following command with `genromfs`

```
# ./genromfs -f romfile.img
```

Mount the file system:

```
# mount -t romfs ./romfile.img /mnt/rom1
```

- Use ***showmount*** command to view the mounting progress of the file system.

```
# showmount
```

```
AUTO-Mount point show >>
```

```
VOLUME                BLK NAME
```

```
-----
```

```
/media/hdd0           /dev/blk/hdd0:0
```

```
Mount point show >>
```

```
VOLUME                BLK NAME
```

```
-----  
/mnt/rom1          ./romfile.img  
  
/tmp              0
```

Though the above steps, a ROMFS-compliant image file is mounted as a ROM file system.

16.6 RAM file system

RAMFS uses a portion of fixed-size memory as a partition. It is not an actual file system, but a mechanism to load the actual file system into memory. Putting some files that are frequently accessed but not changed into memory can significantly improve the performance of the system. Moreover, in the initial stage of device debugging, using an in-memory file system can facilitate device debugging when other file systems of the Flash type are still not working properly.

Use **mount** command to mount RAMFS file system. The ram in /mnt/ram is dynamically created and does not need to be manually created.

```
# showmount  
  
AUTO-Mount point show >>  
  
      VOLUME          BLK NAME  
-----  
  
/media/hdd0          /dev/blk/hdd0:0  
  
Mount point show >>  
  
      VOLUME          BLK NAME  
-----  
  
/tmp                0  
  
# mount -t ramfs 10000 /mnt/ram  
  
# showmount  
  
AUTO-Mount point show >>  
  
      VOLUME          BLK NAME  
-----  
  
/media/hdd0          /dev/blk/hdd0:0
```

```

Mount point show >>

  VOLUME          BLK NAME
-----
/mnt/ram          10000
/tmp              0

# touch /mnt/ram/hy

# ls /mnt/ram/

hy

# umount /mnt/ram/

# showmount

AUTO-Mount point show >>

  VOLUME          BLK NAME
-----
/media/hdd0       /dev/blk/hdd0:0

Mount point show >>

  VOLUME          BLK NAME
-----

/tmp              0

```

16.7 ROOT file system

ROOTFS is a special file system. The root file system is the first file system mounted at kernel startup, so the root file system includes the necessary directories and critical files for SylixOS startup, such as the etc directory necessary for kernel startup and the system command bin directory, etc. Any file system that includes the items necessary for the SylixOS system startup can become the root file system.

ROOTFS of the SylixOS operating system is a virtual-type root file system. Therefore, the file system does not exist on a physical disk but is dynamically created and stored in memory after the system is started.

The directory structure of ROOTFS is shown in table 16.1.

Table 16.1 Directory structure of root file system

Root file system	Typical symbolic link	Feature
/tmp	/yaffs2/n1/tmp	Store temporary files
/var	/yaffs2/n1/var	Store variable data
/root	/yaffs2/n1/root	Directory for root users
/home	/yaffs2/n1/home	Directory for ordinary users
/apps	/yaffs2/n1/apps	Store applications
/sbin	/yaffs2/n1/sbin	System-level executable program
/bin	/yaffs2/n1/bin	Ordinary executable program
/usr	/yaffs2/n1/usr	Store shared data
/lib	/yaffs2/n1/lib	Store shared libraries and kernel modules
/qt	/yaffs2/n1/qt	Store Qt related files
/ftk	/yaffs2/n1/ftk	Store FTK related files
/etc	/yaffs2/n0/etc	Store common configuration files
/boot	/yaffs2/n0/boot	Store the files required by the loader
/usb	N/A	Mount root node of USB
/yaffs2	N/A	YAFFS file system partition
/proc	N/A	PROC file system root node
/media	N/A	Mount root node of mobile device
/mnt	N/A	Mount root node of volume label
/dev	N/A	Mount root node of device

16.8 PROC file system

For easy access to kernel information, SylixOS provides a PROC virtual file system that exists in the /proc directory, contains a variety of files for displaying kernel information. This file system allows the processes to easily read via regular file I/O system calls and can sometimes modify this information. The /proc file system is called a virtual file system because the files and subdirectories it contains are not stored on disk, but are dynamically created by the kernel when the process accesses such information.

Sample directory content of the PROC file system provided by SylixOS:

```
# ls
1          ksymbol   posix     net       power     fs
version   kernel    cpuinfo   bspmem    self      yaffs
```

The description of the files and directories in the /proc directory is shown in table 16.2.

Table 16.2 Files and directories in the /proc directory

File/directory	Description (process property)
1	Process information directory with process ID 1
ksymbol	Kernel symbol table file
posix	Subsystem information directory of POSIX
net	Subsystem information directory of network
power	Subsystem information directory of power management
fs	Subsystem information directory of file system
version	Information file of the kernel version that the current system is running
kernel	Subsystem information directory of kernel
cpuinfo	Processor related information file
bspmem	Mapping file for each physical memory device (RAM or ROM) in system memory
self	Auxiliary information directory

yaffs Information file of YAFFS file system

The files in the /proc directory are usually accessed via scripts, and can also be accessed in the program via regular I/O system calls, but with the following restrictions:

- Some of the files in the /proc directory are read-only, meaning that these files can only display kernel information but cannot be modified. Most of the files in the /proc/pid directory are of this type.
- Some files in the /proc directory can only be read by the file owner (or the superuser process).
- Except for the files in the /proc/pid subdirectory, most of the other files in the /proc directory belong to the root user, and only the root user can modify those files that can be modified.

16.8.1 /proc/pid process related information

Enter the **ps** command in the terminal to view the current process information and the process with PID 0 is the system kernel process. The kernel provides the appropriate directory named /proc/pid for each process in the system, and pid is the process ID. Each file or subdirectory in this directory contains information about the process.

Sample content of the process information directory:

```
# ps
      NAME                FATHER      PID  GRP   MEMORY   UID  GID  USER
-----
kernel                <orphan>    0    0     0        0   0   root
app_proc              <orphan>    1    1   53248    0   0   root
total vprocess : 2

# cd /proc/

# ls
1      ksymbol  posix    net      power    fs
version kernel  cpuinfo  bspmem   self     yaffs

# cd 1

# ls
ioenv      filedesc  modules  mem      cmdline  exe
```

The description of the files in the /proc/pid directory is shown in table 16.3.

Table 16.3 Description of the files in the /proc/pid directory

File	Description (process property)
ioenv	Environment file of process I/O
filedesc	Information file of file descriptor
modules	Situation file of dynamic link library
mem	Memory information file
cmdline	Command line files separated by \0
exe	Symbolic link to executable file

Sample content of the file in the /proc/ pid directory:

```
# cd /proc/pid/1
# cat ioenv
umask:0
wd:/
# cat filedesc
FD NAME
0 /dev/pty/7.hst
1 /dev/pty/7.hst
2 /dev/pty/7.hst
# cat modules
NAME HANDLE TYPE GLB BASE SIZE SYMCNT
app_proc 30c5f170 USER YES c0008000 8428 2
libvmpdm.so 30c5f5a8 USER YES c0018000 d384 70
<VP Ver:1.5.1 dl-malloc>
# cat mem
static memory : 49152
```

```

heap memory      : 4096

total memory    : 53248

# cat cmdline

/apps/app_proc/app_proc

# ll

-r--r----- root    root    Mon Jul 27 14:37:11 2015    0 B, ioenv
-r--r----- root    root    Mon Jul 27 14:37:11 2015    0 B, filedesc
-r--r----- root    root    Mon Jul 27 14:37:11 2015    0 B, modules
-r--r----- root    root    Mon Jul 27 14:37:11 2015    0 B, mem
-r--r----- root    root    Mon Jul 27 14:37:11 2015    0 B, cmdline
lr--r----- root    root    Mon Jul 27 14:37:11 2015          exe/ ->
/apps/app_proc/app_proc

total items : 6

```

16.8.2 /proc/ksymbol kernel symbol table

Example content of the kernel symbol table file:

```

# cat /proc/ksymbol

      SYMBOL NAME                ADDR      TYPE
-----
viShellInit                      3000c888 RX
aodv_netif                       3149e6b4 RW
_cppRtUninit                     302afcec RX
_IosFileSet                      302935b4 RX
_epollFindEvent                  3028efb8 RX
__blockIoDevDelete              302631dc RX
__pxnameGet                     3023db10 RX
mq_timedreceive                  3023cc90 RX
API_INetNpfDetach               30221e90 RX
snmp_set_sysname                 301da42c RX
igmp_joingroup                   301c6d6c RX

```



```

vprocIoFileDescGet          30178130 RX
API_MonitorUploadCreate     3016e528 RX
API_MonitorNet6UploadCreate 3016d490 RX

```

16.8.3 /proc/posix POSIX subsystem information

The POSIX subsystem contains the naming information file pnamed. The POSIX naming information can be understood by viewing the pnamed content with the cat command. In the pnamed content, TYPE indicates the type (SEM indicates the signal type, MQ indicates the message queue), OPEN indicates the use count, and NAME indicates the object name.

Sample content of the /proc/posix directory:

```

# cd /proc/posix/

# ls

pnamed

# cat pnamed

TYPE  OPEN          NAME
-----
SEM   1 sem_named

```

16.8.4 /proc/net network subsystem

Sample content of the network subsystem directory:

```

# cd /proc/net/

# ls

netfilter  wireless      ppp           packet        arp           if_inet6
dev        unix          tcpip_stat    route         igmp6        igmp          raw6
raw       udplite6     udplite      udp6          udp          tcp6
tcp       mesh-adhoc

```

The description of each directory file in the network subsystem is shown in table 16.4.

Table 16.4 Description of the the files in the /proc/net directory

File	Description (process property)
netfilter	Network filter rules file

wireless	Wireless network configuration file
ppp	PPP dial file
packet	AF_PACKET information file
arp	ARP information file
if_inet6	PIV6 network interface file
dev	Information file of network interface device
unix	AF_UNIX information file
tcpip_stat	TCP/IP status information file
route	Routing table information file
igmp6	IPV6 IGMP information file
igmp	IGM information file
raw6	Information file of IPV6 raw data
raw	Raw data information file
udplite6	IPV6 UDP brief information file
udplite	UDP brief information file
udp6	IPV6 UDP information file
udp	UDP information file
tcp6	IPV6 TCP information file
tcp	TCP information file
mesh-adhoc	Mesh ad hoc network information directory

16.8.5 /proc/power power management subsystem

Sample content of the power management subsystem directory:

```
# cd /proc/power/

# ls

pminfo      devices    adapter
```

The description of the files in the /proc/power directory is shown in table 16.5.

Table 16.5 Description of the files in the /proc/power directory

File	Description (process property)
pminfo	Current system information file
devices	Device file of enabling power management
adapter	Adapter information file

Sample information of pminfo file:

```
# cat pminfo

NCPUS      : 1

ACTIVE     : 1

POWERLevel : Top

SYSStatus  : Running
```

- NCPUS: Current system CPU cores;
- ACTIVE: Current system-enabled CPU cores;
- POWERLevel: Power level. The levels from top to bottom are Top, Fast, Normal, and Slow;
- SYSStatus: Current system running status. The running statuses include the low-power status Power-Saving and the normal status Running.

devices file information:

```
# cat devices

PM-DEV      ADAPTER      CHANNLE    POWER

uart2       inner_pm      12         on

uart1       inner_pm      11         on

uart0       inner_pm      10         on
```

- PM-DEV: Power management device name;
- ADAPTER: Power management adapter name;
- CHANNLE: Channel number of the power management adapter;
- POWER: Current power statuses including on and off.

adapter file information:

```
# cat adapter
ADAPTER      MAX-CHANNLE
inner_pm     21
```

- ADAPTER: Power management adapter name;
- MAX-CHANNLE: Maximum channel number of the power management adapter.

16.8.6 Subsystem of the /proc/fs file system

Sample content of the file system directory:

```
# cd /proc/fs
# ls
fssup      procfs    rootfs
```

The description of the files in the /proc/fs directory is shown in table16.6.

Table 16.6 Description of the files in the /proc/fs directory

Directory/file	Description (process property)
fssup	File system support information file
procfs	Information directory of PROC file system
rootfs	Information directory of ROOT file system

fssup file information:

```
# cat fssup
rootfs procfs ramfs romfs vfat nfs yafts
```

procfs directory information:

```
# cd procfs/  
  
# ls  
  
stat  
  
# cat stat  
  
memory used : 0 bytes  
  
total files : 47
```

- memory used: file size;
- total files: number of files.

rootfs directory information:

```
# cd rootfs/  
  
# ls  
  
stat  
  
# cat stat  
  
memory used : 3031 bytes  
  
total files : 43
```

- memory used: file size;
- total files: number of files.

16.8.7 /proc/version kernel version information

Sample content of kernel version information file:

```
# cd /proc  
  
# cat version  
  
SylixOS kernel version: 1.2.1 NeZha(a) BSP version 5.1.2 for GEMINI  
  
(compile time : Jan 15 2016 11:44:33)  
  
GCC:4.9.3
```

The kernel version information contains SylixOS kernel version information, BSP version information, SylixOS kernel compile time and compiler version information.

16.8.8 /proc/kernel kernel information

Sample content of the kernel information directory:

```
# cd /proc/kernel/

# ls

affinity      objects      tick
```

The description of the files in the /proc/kernel directory is shown in table 16.7.

Table 16.7 Description of the files in the /proc/kernel directory

File	Description (process property)
affinity	Multi-core affinity information file
objects	Kernel object information file
tick	Systick information file

Sample content of the affinity file:

```
# cat affinity

      NAME          TID    PID  CPU
-----
t_idle            4010000    0   *
t_itimer          4010001    0   *
t_except          4010002    0   *
t_log             4010003    0   *
t_power           4010004    0   *
t_hotplug         4010005    0   *
t_reclaim         4010007    0   *
t_netjob          4010008    0   *
t_netproto        4010009    0   *
t_tftpd           401000a    0   *
t_ftpd            401000b    0   *
t_telnetd         401000c    0   *
t_tshell          401000e    0   *
```

- NAME: thread name;

- TID: thread ID;
- PID: process ID;
- CPU: Current thread affinity to the specified CPU.

Sample content of the objects file:

```
# cat objects
object      total    used    max-used
event       1500    99      101
eventset    100     0       0
heap        22      2       2
msgqueue    300     6       6
partition   30      7       7
rms         30      1       1
thread      100     13      14
threadvar   20      0       0
timer       100     2       2
dpma        2       0       0
threadpool  2       0       0
```

- object: Kernel object type;
- total: The total number of specified type objects;
- used: The number of specified type objects that have been used
- max-used: The maximum number of specified type objects that have been used

Sample content of the tick file:

```
# cat tick
tick rate   : 100 hz
tick        : 44304
```

- tick rate: System clock frequency;
- total: Total system clock count

16.8.9 /proc/cpuinfo processor information

Sample content of the cpuinfo file:

```
# cd /proc

# cat cpuinfo

CPU           : SAMSUNG S3C2440A (ARM920T 405/101MHz NonFPU)

CPU Family    : ARM(R) 32-Bits

CPU Endian    : Little-endian

CPU Cores     : 1

CPU Active    : 1

PWR Level     : Top level

CACHE         : 32KBytes L1-Cache (D-16K/I-16K)

PACKET        : Mini2440 Packet

BogoMIPS 0    : 426.600
```

- CPU: processor type and key parameters;
- CPU Family: processor architecture type and word length;
- CPU Endian: big endian and little endian type;
- CPU Cores: Processor cores;
- CPU Active: The number of currently active processors;
- PWR Level: current power level;
- CACHE: Cache information;
- PACKET: board support package type;
- BogoMIPS 0: a measure of the calculator speed in SylixOS (millions per second).

16.8.10 /proc/bspmem memory mapping information

Sample content of the bspmem file:

```
# cd /proc

# cat bspmem

ROM SIZE: 0x00200000 Bytes (0x00000000 - 0x001fffff)

RAM SIZE: 0x04000000 Bytes (0x30000000 - 0x33ffffff)

use "mems" "zones" "virtuals"... can print memory usage factor.
```

16.8.11 proc/self auxiliary information

Sample content of the self directory:

```
# cd /proc/self/  
  
# ls  
  
auxv
```

16.8.12 /proc/yaffs YAFFS partition information

Sample content of the yaffs file:

```
# cd /proc  
  
# cat yaffs  
  
Device : "/n1"  
  
startBlock..... 129  
  
endBlock..... 1023  
  
totalBytesPerChunk. 2048  
  
chunkGroupBits..... 0  
  
chunkGroupSize..... 1  
  
nErasedBlocks..... 871  
  
nReservedBlocks.... 16  
  
nCheckptResBlocks.. nil  
  
blocksInCheckpoint. 0  
  
nObjects..... 23  
  
nTnodes..... 96  
  
nFreeChunks..... 55975  
  
nPageWrites..... 0  
  
nPageReads..... 13  
  
nBlockErasures..... 0  
  
nErasureFailures... 0  
  
nGCCopies..... 0  
  
allGCs..... 0  
  
passiveGCs..... 0  
  
nRetriedWrites..... 0
```

```
nShortOpCaches..... 20
nRetiredBlocks..... 0
eccFixed..... 0
eccUnfixed..... 0
tagsEccFixed..... 0
tagsEccUnfixed..... 0
cacheHits..... 0
nDeletedFiles..... 0
nUnlinkedFiles..... 0
nBackgroudDeletions 0
useNANDECC..... 1
isYaffs2..... 1

Device : "/n0"
startBlock..... 1
endBlock..... 128
totalBytesPerChunk. 2048
chunkGroupBits..... 0
chunkGroupSize..... 1
nErasedBlocks..... 126
nReservedBlocks.... 10
nCheckptResBlocks.. nil
blocksInCheckpoint. 0
nObjects..... 9
nTnodes..... 3
nFreeChunks..... 8183
nPageWrites..... 0
nPageReads..... 6
nBlockErasures..... 0
nErasureFailures... 0
```

```
nGCCopies..... 0
allGCs..... 0
passiveGCs..... 0
nRetriedWrites..... 0
nShortOpCaches..... 10
nRetiredBlocks..... 0
eccFixed..... 0
eccUnfixed..... 0
tagsEccFixed..... 0
tagsEccUnfixed..... 0
cacheHits..... 0
nDeletedFiles..... 0
nUnlinkedFiles..... 0
nBackgroundDeletions 0
useNANDECC..... 1
isYaffs2..... 1
```

16.9 YAFFS file system

YAFFS (Yet Another Flash File System) is an embedded log file system designed specifically for NAND Flash memory and is suitable for large-capacity storage devices. As it is released under the GPL, the source code is available for free on its website.

YAFFS is a log-based file system that provides robustness to wear-leveling and power-down recovery. It also made adjustments for large-capacity Flash chips, optimized boot time and RAM usage, making it suitable for large-capacity storage devices.

16.9.1 The difference between NAND Flash and NOR Flash

NOR Flash features XIP (Execute In Place) so that applications can run directly within the NOR Flash without writing code into system RAM. Its high transmission efficiency makes it highly cost-effective at small capacities of 1 to 4 MB, but extremely low write and erase speed greatly affect its performance.

NAND Flash architecture was released by Toshiba in 1989. Its internal non-linear macro cell mode provides a cheap and effective solution for the development of solid-state large-capacity memory. The advantages of large capacity, fast write and erase speed make

NAND Flash suitable for storing large amounts of data, so it has been widely used in the industry, such as embedded products including digital cameras, memory cards for MP3 players, compact USB flash disks, etc.

Both types of FLASH have the same memory cells and operating principle. In order to shorten the access time, operations are not performed individually for each cell, but are performed collectively for a certain number of cells. The NAND FLASH memory cells are connected in series, while the NOR FLASH memory cells are connected in parallel. Therefore, all memory cells must be uniformly addressed for effective management. All memory cells of NAND FLASH are divided into several blocks, and each block is divided into several pages. Each page is 512 bytes or 2048 bytes, which is 512 or 2048 8 digits. This means that each page has 512 or 2048 bit lines and each bit line contains 8 memory cells, so the data stored on each page is exactly the same as the data stored on one sector of the hard disk. This is specially designed to facilitate the data exchange with the disk. A NAND FLASH block is similar to a cluster of the hard disk. When the capacity is different, the number of blocks is different, and the number of pages that make up the block is also different. When the word line and the bit line are locked to a certain transistor while reading data, the control electrode of this transistor is not biased, and the other seven are biased to conduct. If there is a charge in the floating gate of this transistor, it will conduct to make the bit line low. At this time, the readout number is 0, otherwise it is 1. Each memory cell of the NOR FLASH is connected in parallel to the bit lines to facilitate random access to each bit, and a dedicated address line enables one-time direct addressing, reducing the time for NOR FLASH to execute processor instructions.

In NAND FLASH, the maximum number of EW per block is one million, while in NOR FLASH it is one hundred thousand. Compared with NOR FLASH, NAND FLASH has a 10 to 1 advantage over the block erase cycle. Typical NAND FLASH block size is 8 times smaller than NOR FLASH, each NAND FLASH block erases less in a given time, and NAND FLASH controller interface is simpler.

16.9.2 YAFFS proper nouns

Page: The page unit is the addressing unit of the basic operation (read or write operation or bad block flag) including a general area and an extension area. The general area is mainly used for storing data, and the extended area is mainly used for storing flag information;

Block: The block unit is the address unit of the erase operation;

OOB: Exist in extension area, including ECC, bad block flag, YAFFS flag, etc. Typically, each 512 bytes corresponds to a 16-byte OOB area, and each 2048 bytes corresponds to a 64-byte OOB area.

YAFFS flags in YAFFS2 mode:

- 4-byte 32-bit data block ID
- 4-byte 32-bit object ID;

- The number of data bytes in this 2-byte data block;
- The serial number of this 4-byte block;
- 3-byte flag area ECC
- 12-byte data area ECC (3-byte ECC per 256-byte data).

Chunk: Addressing unit of YAFFS, usually consistent with the size of Page;

Object: YAFFS object, usually including files, paths, links, devices, etc.

Taking the NAND Flash of K9F4G08 from SAMSUNG as an example to illustrate the structure of the common NAND Flash physical memory cell array. The schematic for the structure of the K9F4G08 physical memory cell array is shown in figure 16.6.

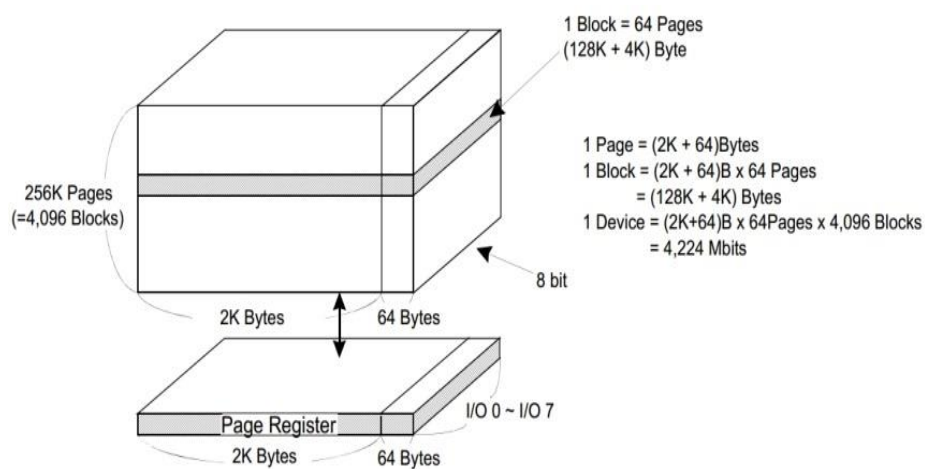


Figure 16.6 K9F4G08 storage array schematic

16.9.3 Memory Technology Device (MTD)

MTD (Memory Technology Device) provides an abstract layer for memory operations in order to make the drive of newly added storage devices easier. This subsystem provides a common interface for drivers and upper-level systems to ensure that all memory (NAND, OneNAND, NOR, AG-AND, NOR with ECC, etc.) operations use the same API. Hardware drivers shall ignore the basic data storage formats and only provide basic operations such as read operations, write operations, and erase operations.

MTD devices are neither character devices nor block devices. The comparison between MTD devices and block devices is shown in figure 16.7.

Figure 16.7 The comparison between MTD devices and block devices

Block device	MTD device
Consists of sectors	Consists of erasable blocks

Smaller sector size (512 or 1024 Byte)	Larger erasable sector size (128 KB)
Read and write sectors	Read block, write block and erase block
Bad sectors are hidden or remapped by hardware	Bad blocks cannot be hidden and require software processing
No limit on the reads and writes of sectors	Limited block erasures

The MTD subsystem source code in SylixOS is located at "libsylxos/SylixOS/fs/mtd", which mainly uses the MTD original device layer of the MTD subsystem. The Flash hardware driver layer includes the NOR Flash driver and the NAND Flash driver and the NAND Flash driver source code is located in "SylixOS/driver/mtd/nand" in the BSP package.

16.9.4 YAFFS partition

The YAFFS file system in SylixOS usually has two partitions as /yaffs2/n0 and /yaffs2/n1. N0 is the boot area, which mainly stores the device firmware and some common configuration files. N1 is the comm area which mainly stores the common files.

Sample directory structure of YAFFS file system:

```
# ls
tmp          var          root         home         apps
sbin        bin          usr          lib          qt
ftk         etc          boot         usb          yaffs2
proc        media        mnt          dev

# cd yaffs2/

# ls
n1          n0

# ll n0
drwxr-xr-- root    root    Mon Jul 27 14:12:39 2015    boot/
drwxr-xr-- root    root    Mon Jul 27 14:14:58 2015    etc/
drw-rw-rw- root    root    Tue Aug 04 10:55:20 2015    lost+found/

total items : 3

# ll n1
```

```

drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    ftk/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    qt/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    lib/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    usr/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    bin/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    tmp/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    sbin/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    apps/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    home/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    root/
drwxr-xr-- root    root    Thu Jul 30 10:03:41 2015    var/
drw-rw-rw- root    root    Tue Aug 04 10:55:20 2015    lost+found/

total items : 12

```

Description for the sample directory structure of n0 partition:

- boot: boot loader required file directory;
- etc: configuration files (passwd, group, shadow, startup.sh, etc.) directory.

Description for the sample directory structure of n1 partition:

- ftk: FTK (funny tool kit) embedded GUI system directory
- qt: Qt cross-platform C++ GUI directory
- lib: dynamic library and kernel module directory;
- usr: user directory for storing user-level files;
- bin: executable files and program directories required at system startup;
- tmp: temporary file directory. The temporary file generated after system startup is stored in /var/tmp;
- sbin: executable program directory;
- apps: application directory;
- home: Ordinary user's personal file directory
- root: system core file directory;
- var: frequently changed file directory during system execution.

Common symbolic links:

```
/yaffs2/n0/boot --> /boot
/yaffs2/n0/etc --> /etc
/yaffs2/n1/ftk --> /ftk
/yaffs2/n1/qt --> /qt
/yaffs2/n1/lib --> /lib
/yaffs2/n1/usr --> /usr
/yaffs2/n1/bin --> /bin
/yaffs2/n1/sbin --> /sbin
/yaffs2/n1/apps --> /apps
/yaffs2/n1/home --> /home
/yaffs2/n1/root --> /root
/yaffs2/n1/var --> /var
/yaffs2/n1/tmp --> /var/tmp
/yaffs2/n1/tmp --> /tmp
```

16.9.5 YAFFS command

Operate the YAFFS file system with the ***yaffscmd*** command in SylixOS.

[Command format]

```
yaffscmd volname [{bad | info | markbad | erase}]
```

[Common option]

None

[Instructions for parameters]

```
volname : Volume label name, currently using n0 or n1 in the SylixOS system
bad : Bad block information
info : Parameter information
markbad : Mark bad blocks
erase : Memory erase
```

The following is the output of querying parameter information with the ***yaffscmd*** command:

```
# yaffscmd n1 info
Device : "/n1"
```



```
startBlock..... 129
endBlock..... 1023
totalBytesPerChunk. 2048
chunkGroupBits..... 0
chunkGroupSize..... 1
nErasedBlocks..... 894
nReservedBlocks.... 16
nCheckptResBlocks.. nil
blocksInCheckpoint. 0
nObjects..... 18
nTnodes..... 0
nFreeChunks..... 57265
nPageWrites..... 28
nPageReads..... 15
nBlockErasures..... 0
nErasureFailures... 0
nGCCopies..... 0
allGCs..... 0
passiveGCs..... 0
nRetriedWrites..... 0
nShortOpCaches..... 20
nRetiredBlocks..... 0
eccFixed..... 0
eccUnfixed..... 0
tagsEccFixed..... 0
tagsEccUnfixed..... 0
cacheHits..... 0
nDeletedFiles..... 0
nUnlinkedFiles..... 0
nBackgroudDeletions 0
```

```
useNANDECC..... 1
```

```
isYaffs2..... 1
```

The following is the output of marking bad blocks with the **yaffscmd** command:

```
# yaffscmd n1 markbad aa

yaffs: marking block 170 bad

mark the block 0xaa is a bad ok.
```

The following is the output of viewing bad block information with the **yaffscmd** command:

```
[root@sylixos_station:/]# yaffscmd n1 bad

block 0xaa is bad block.
```

The following is the output of erasing memory with the **yaffscmd** command:

```
# yaffscmd n1 erase

yaffs volume erase ok.
```

16.10 File system Shell command

Switch the current directory with the **cd** command in SylixOS.

[Command format]

```
cd path
```

[Common option]

None

[Instructions for parameters]

```
path: path name
```

The following is the output of switching the current directory to the /etc directory with the **cd** command:

```
# cd /etc/

# ls

pointercal      passwd          group           shadow          fs_init.sh
qtenv.sh        qtln_4.8.6.sh  startup.sh      profile
```

Switch the directory with the **ch** command in SylixOS.

[Command format]

```
ch dir
```

[Common option]

None

[Instructions for parameters]

dir: path name

The following is the output of switching the current directory to the /etc directory with the **ch** command:

```
# ch /etc/

# ls

pointercal      passwd          group           shadow          fs_init.sh
qtenv.sh        qtln_4.8.6.sh  startup.sh      profile
```

View the current directory with the **pwd** command in SylixOS.

[Command format]

pwd

[Common option]

None

[Instructions for parameters]

None

The following is the output of viewing the current directory with the **pwd** command:

```
# cd /etc

# pwd

/etc
```

View the file system information for the specified directory with the **df** command in SylixOS and the default is the current directory.

[Command format]

df volume name

[Common option]

None

[Instructions for parameters]

volume name: path name

The following is the output of viewing the n1 file system information in the yaff2 partition with the **df** command:

```
# df /yaffs2/n1
```

VOLUME	TOTAL	FREE	USED	RO	FS TYPE
/yaffs2/n1	109.88MB	42.05MB	61%	n	YAFFS FileSystem

Get a temporary file name that can be created with the **tmpname** command in SylixOS.

[Command format]

```
tmpname
```

[Common option]

None

[Instructions for parameters]

None

The following is the output of the **tmpname** command:

```
# tmpname

can mktmp as name : /tmp/tmp.0.6DQbEP

# tmpname

can mktmp as name : /tmp/tmp.1.HcRXHN
```

Create a directory with the **mkdir** command in SylixOS.

[Command format]

```
mkdir directory
```

[Common option]

None

[Instructions for parameters]

```
directory: directory name
```

The following is the output of creating a new directory **sylixos** in the **/tmp** directory with the **mkdir** command:

```
# cd /tmp

# ls

.qt_soundserver-0          qtembedded-0

# mkdir sylixos

# ls
```

sylixos

.qt_soundserver-0

qtembedded-0

Create a named pipe with the **mkfifo** command in SylixOS.

Note: can only be created in the root file system device.

[Command format]

```
mkfifo [fifo name]
```

[Common option]

None

[Instructions for parameters]

```
[fifo name]: fifo name
```

The following is the output of creating a new named pipe sy in the /dev directory with the **mkfifo** command:

```
# cd /dev

# ls

log          socket      netevent    fb0          ttyS2
ttyS1        ttyS0       urandom     random       shm          rtc
hotplug      epollfd    gpiofd      signalfd     hstimerfd
timerfd      eventfd    zero        null         input
pipe         pty

# mkfifo /dev/sy

# ls

sy          log          socket      netevent    fb0
ttyS2       ttyS1        ttyS0       urandom     random       shm
rtc         hotplug     epollfd    gpiofd      signalfd
hstimerfd  timerfd     eventfd    zero        null
input      pipe        pty

# devs

device show (minor device) >>

drv open name

19  0 /dev/sy

14  1 /dev/pty/9.dev

15  1 /dev/pty/9.hst

30  0 /dev/input/xmse

30  0 /dev/input/xkbd
```

```
29 0 /dev/socket
28 0 /dev/netevent
26 1 /dev/input/touch0
27 0 /dev/fb0
24 0 /yaffs2
16 0 /dev/ttyS2
16 0 /dev/ttyS1
16 1 /dev/ttyS0
13 0 /dev/urandom
13 0 /dev/random
12 0 /dev/shm
11 0 /proc
10 0 /dev/rtc
9 1 /dev/hotplug
8 0 /dev/epollfd
7 0 /dev/gpiofd
6 0 /dev/signalfd
5 0 /dev/hstimerfd
4 0 /dev/timerfd
3 0 /dev/eventfd
1 0 /dev/zero
0 0 /dev/null
2 0 /
```

Delete a directory with the ***rmdir*** command in SylixOS.

[Command format]

```
rmdir directory
```

[Common option]

None

[Instructions for parameters]

```
directory: directory name
```

The following is the output of deleting the `sylixos` directory in the `/tmp` directory with the `rmdir` command:

```
# cd /tmp

# ls

qtembedded-0  .qt_soundserver-0      sylixos

# rmdir sylixos/

# ls

qtembedded-0  .qt_soundserver-0
```

Delete a file with the `rm` command in SylixOS.

[Command format]

```
rm file name
```

[Common option]

None

[Instructions for parameters]

```
file name
```

The following is the output of deleting the `sy` file in the `/tmp` directory with the `rm` command:

```
# cd /tmp

# ls

qtembedded-0  .qt_soundserver-0

# touch sy

# ls

sy            qtembedded-0  .qt_soundserver-0

# rm sy

# ls

qtembedded-0  .qt_soundserver-0
```

Move or rename a file with the `mv` command in SylixOS.

[Command format]

```
mv SRC file name, DST file name
```

[Common option]

None

[Instructions for parameters]

SRC file name : source file name

DST file name : Destination file name

The following is the output of renaming the `sy` file in the `/tmp` directory with the `mv` command:

```
# cd /tmp

# ls

qtembedded-0    .qt_soundserver-0

# touch sy

# ls

sy              qtembedded-0    .qt_soundserver-0

# mv sy sy0

# ls

sy0            qtembedded-0    .qt_soundserver-0
```

View the content of a file with the `cat` command in SylixOS.

[Command format]`cat` file name**[Common option]**

None

[Instructions for parameters]

file name: file name

The following is the output of viewing the content of the `sy` file in the `/tmp` directory with the `cat` command:

```
# cd /tmp

# ls

qtembedded-0    .qt_soundserver-0

# touch sy

# ls

sy              qtembedded-0    .qt_soundserver-0
```



```
# vi sy
sylixos and soft
# ls
sy          qtembedded-0    .qt_soundserver-0
# cat sy
sylixos and soft
```

Copy a file with the **cp** command in SylixOS.

[Command format]

```
cp scr file name dst file name
```

[Common option]

None

[Instructions for parameters]

scr file name: source file name

dst file name: destination file name

The following is the output of copying the sy file in the /tmp directory to the sy0 file with the **cp** command.

```
# cd /tmp
# ls
sy          qtembedded-0    .qt_soundserver-0
# cat sy
sylixos and soft
# cp sy sy0
copy complete. size:20(Bytes) time:0(s) speed:20(Bps)
# ls
sy0         sy          qtembedded-0    .qt_soundserver-0
# cat sy0
sylixos and soft
```

Compare two files with the **cmp** command in SylixOS.

[Command format]

```
cmp [file one] [file two]
```

[Common option]

None

[Instructions for parameters]

[file one]: file 1's name

[file two]: file 2's name

The following is the output of comparing the `sy` file and the `sy0` file in the `/tmp` directory with the **`cmp`** command:

```
# cd /tmp
# ls
sy0          sy          qtembedded-0  .qt_soundserver-0
# cat sy0
sylixos and soft
# cat sy
sylixos and soft
# cmp sy0 sy
file same!
```

Create a file with the **`touch`** command in SylixOS.

[Command format]**`touch`** [-amc] file name**[Common option]**

```
-a: Change only access time
-m: Only change the modification time
-c: Do not create a file
```

[Instructions for parameters]

file name: file name

The following is the output of creating a `sylixos` file in the `/tmp` directory and changing the contents of the file with the **`touch`** command:

```
# cd /tmp
# ls
qtembedded-0  .qt_soundserver-0
```

```
# touch sylixos

# ls

sylixos          qtembedded-0    .qt_soundserver-0

# vi sylixos

sylixos and soft

# ls

sylixos          qtembedded-0    .qt_soundserver-0

# cat sylixos

sylixos and soft
```

List the files in the specified directory with the **ls** command in SylixOS and the default is the current directory.

[Command format]

```
ls [path name]
```

[Common option]

None

[Instructions for parameters]

```
path name: directory name
```

The following is the output of listing the files in the /tmp directory with the **ls** command:

```
# ls /tmp/

sylixos          qtembedded-0    .qt_soundserver-0

# cd /tmp/

# ls

sylixos          qtembedded-0    .qt_soundserver-0
```

List the file details in the specified directory with the **ll** command in SylixOS and the default is the current directory.

[Command format]

```
ll [path name]
```

[Common option]

None

[Instructions for parameters]

```
path name: directory name
```

The following is the output of listing the file details in the /tmp directory with the **ll** command:

```
# ll /tmp/

-rw-r--r-- root    root    Thu Jun 18 20:19:19 2015    20 B, syl
drwx----- root    root    Thu Jun 18 15:38:06 2015           qtembedded-0/
-rw----- root    root    Thu Jun 18 15:38:06 2015    0 B, .qt_soundserver-0

    total items : 3

# cd /tmp/

# ll

-rw-r--r-- root    root    Thu Jun 18 20:19:19 2015    20 B, syl
drwx----- root    root    Thu Jun 18 15:38:06 2015           qtembedded-0/
-rw----- root    root    Thu Jun 18 15:38:06 2015    0 B, .qt_soundserver-0

    total items : 3
```

Calculate the size of all the files contained in a given directory with the **dsiz**e command in SylixOS.

[Command format]

```
dsize [path name]
```

[Common option]

None

[Instructions for parameters]

```
path name: directory name
```

The following is the output of calculating all file sizes in the /tmp directory with the **dsiz**e command:

```
# dsize /tmp/

scanning...

total file 4 size 172
```

Set the permission bits of a directory or file with the **chm**od command in SylixOS.

[Command format]

```
chmod newmode filename
```

[Common option]

None

[Instructions for parameters]

newmode: New permission bits, using absolute mode

filename: File or directory name

The following is the output of setting the permission bits of the `sylixos` file in the `/tmp` directory with the **`chmod`** command:

```
# cd /tmp

# ll

-rw-r--r-- root    root    Thu Jun 18 20:19:19 2015    20 B, sylixos
drwx----- root    root    Thu Jun 18 15:38:06 2015           qtembedded-0/
-rw----- root    root    Thu Jun 18 15:38:06 2015    0 B, .qt_soundserver-0

    total items : 3

# chmod 755 sylixos

# ll

-rwxr-xr-x root    root    Thu Jun 18 20:19:19 2015    20 B, sylixos
drwx----- root    root    Thu Jun 18 15:38:06 2015           qtembedded-0/
-rw----- root    root    Thu Jun 18 15:38:06 2015    0 B, .qt_soundserver-0
```

Format the specified disk with the **`mkfs`** command in SylixOS.

[Command format]

`mkfs` media name

[Common option]

None

[Instructions for parameters]

media name: disk name

The following is the output of formatting `sdcard0` with the **`mkfs`** command:

```
# mkfs /media/sdcard0/

now format media, please wait...

disk format ok.
```

Run the specified Shell script file with the **`shfile`** command in SylixOS.

[Command format]

```
shfile shell file
```

[Common option]

None

[Instructions for parameters]

```
shell file: shell file
```

The following is the output of running script file `factory.sh` in the `/etc` directory with the **shfile** command:

```
# cd /etc

# ls

passwd          group           shadow          fs_init.sh     qtenv.sh
qtl_n_4.8.6.sh  startup.sh      profile         pointercal

# touch factory.sh

# vi factory.sh

echo factory shell file

# cat factory.sh

echo factory shell file

# shfile factory.sh

factory shell file
```

Mount a volume with the **mount** command in SylixOS.

[Command format]

```
mount [-t fstype] [-o option] [blk dev] [mount path]
```

[Common option]

```
-t: File system types such as ramfs, romfs, nfs, etc.
```

```
-o: File system type, ro is read-only, rw is read-write type
```

[Instructions for parameters]

```
[blk dev]: block device
```

```
[mount path]: mount path
```

The following is the output of using the **mount** command:

```
# showmount

AUTO-Mount point show >>
```

```

          VOLUME          BLK NAME
-----
/media/hdd0          /dev/blk/hdd0:0

Mount point show >>

          VOLUME          BLK NAME
-----
/tmp                 0
# mount -t ramfs 100 /mnt/ram
# showmount

AUTO-Mount point show >>

          VOLUME          BLK NAME
-----
/media/hdd0          /dev/blk/hdd0:0

Mount point show >>

          VOLUME          BLK NAME
-----
/mnt/ram             100
/tmp                 0

```

Unmount a volume with the **umount** command in SylixOS.

[Command format]

```
umount [mount path]
```

[Common option]

None

[Instructions for parameters]

```
[mount path]: device path
```

The following is the output of using the **umount** command:

```
# showmount

AUTO-Mount point show >>
```

```

          VOLUME          BLK NAME
-----
/media/hdd0          /dev/blk/hdd0:0

Mount point show >>

          VOLUME          BLK NAME
-----
/mnt/ram             100
/tmp                 0

# umount /mnt/ram/

# showmount

AUTO-Mount point show >>

          VOLUME          BLK NAME
-----
/media/hdd0          /dev/blk/hdd0:0

Mount point show >>

          VOLUME          BLK NAME
-----

/tmp                 0

```

View all the mounted volumes in the system with the **showmount** command in Sylix OS.

[Command format]

showmount

[Common option]

None

[Instructions for parameters]

None

The following is the output of viewing all the mounted volumes in the system with the **showmount** command:

```
# showmount
```



```

AUTO-Mount point show >>

VOLUME                BLK NAME
-----
/media/hdd0           /dev/blk/hdd0:0

Mount point show >>

VOLUME                BLK NAME
-----
/tmp                  0

```

Create a symbolic link file with the **ln** command in SylixOS.

[Command format]

```
ln [-s | -f] [actualpath] [sympath]
```

[Common option]

-s: Soft links (symbolic links)

-f: Enforce

[Instructions for parameters]

[actualpath]: Actual path

[sympath]: Symbolic link path

The following is the output of linking /tmp/sylixos to /sylixos with the **ln** command:

```

# ls

tmp          var          root         home         apps         sbin
bin          usr          lib          qt           ftk          etc
boot        usb          yaffs2       proc         media        mnt
dev

# mkdir /tmp/sylixos

# ls /tmp/

sylixos      ram0         syl          qtembedded-0 .qt_soundserver-0

# ln -s /tmp/sylixos /sylixos

# ls

sylixos      tmp          var          root         home         apps
sbin         bin          usr          lib          qt           ftk

```

	etc	boot	usb	yaffs2	proc
media	mnt	dev			

View or set the fat file system volume label with the ***dosfslabel*** command in SylixOS.

[Command format]

```
dosfslabel [[vol newlabel] [vol]]
```

[Common option]

None

[Instructions for parameters]

[vol newlabel]: Volume name

[vol]: Label

The following is the output of viewing the volume label of /media/sdcard0 and setting the new label as sylixos with the ***dosfslabel*** command:

```
# dosfslabel /media/sdcard0/  
  
sdcard  
  
# dosfslabel /media/sdcard0/ sylixos  
  
# dosfslabel /media/sdcard0/  
  
sylixos
```

Chapter 17 Logging System

17.1 SylixOS logging system

SylixOS has added a log management capability to enable real-time recording of various events that occur in the system. By analyzing the log files, users can find and solve the runtime problems in a timely manner.

In actual use, logs are divided into different levels according to specific conditions. SylixOS log level is compatible with Linux log level and SylixOS provides the following macros to indicate different log levels:

- **KERN_EMERG**: may cause the host system to be unavailable;
- **KERN_ALERT**: must be solved immediately;
- **KERN_CRIT**: a worse situation;
- **KERN_ERR**: runtime error;
- **KERN_WARNING**: may affect the system functionality;
- **KERN_NOTICE**: not affecting the system but is worth noting;
- **KERN_INFO**: general information;
- **KERN_DEBUG**: Program or system debugging information, etc.

The Log level goes down from top to bottom. In general, if a KERN_EMERG level log is found, it means that the system cannot run due to a serious problem. The KERN_DEBUG level logs are usually used to print some debugging information. In the development of the SylixOS driver, the KERN_ERR level logs are often used to print some error messages, and the KERN_INFO level logs are used to print some general information.

Call the following function to print log message.

```
#include <SylixOS.h>

INT  logPrintk(CPCHAR  pcFormat, ...);
```

The prototype analysis of the function logPrintk:

- This function returns the print length on success, returns -1 and sets the error number on failure;
- The parameter *pcFormat* is the log print format string;
- The parameter ... is the variable parameter that can transfer more parameters.

This function is typically used to drive log printing in development and is functionally equivalent to the `printk` function, so the `printk` function is often used in place of this function to improve program compatibility.

Note: In fact, `printk` in SylixOS is a macro definition of the `logPrintk` function.

The `logPrintk` function can be called to print the log message to the terminal. In the case of a small amount of log message, this is undoubtedly an effective viewing method, but when the log message is greatly increased, this method will be inefficient or even undesirable. To solve this problem, log message is usually printed to a specified file for subsequent analysis.

The following functions are used to set the file descriptor of the log file.

```
#include <SylixOS.h>

INT    logFdSet(INT    iWidth, fd_set  *pfsetLog);

INT    logFdGet(INT    *piWidth, fd_set  *pfsetLog);
```

The prototype analysis of the function `logFdSet`:

- This function returns 0 on success, returns -1 and sets the error number on failure;
- The parameter *iWidth* is the width of the file descriptor;
- The parameter *pfsetLog* is the file descriptor set of interest.

The function `logFdSet` sets the new file descriptor set to that of the logging system and the function `logFdGet` gets the previous file descriptor information of the logging system. It should be noted that these two functions are usually used as follows to ensure that the current file descriptor set of the logging system will not be destroyed:

```
logFdGet(&iWidth, &fdset);

FD_SET(iNewFd, &fdset);

logFdSet(iNewWidth, &fdset);
```

When the file descriptor is set, the following functions are called to print the log.

```
INT    logMsg(CPCHAR    pcFormat, PVOID    pvArg0,
             PVOID    pvArg1,  PVOID    pvArg2,
             PVOID    pvArg3,  PVOID    pvArg4,
             PVOID    pvArg5,  PVOID    pvArg6,
             PVOID    pvArg7,  PVOID    pvArg8,
             PVOID    pvArg9,  BOOL     bIsNeedHeader);
```

The prototype analysis of the function `logMsg`:

- This function returns 0 on success, returns -1 and sets the error number on failure;

- The parameter ***pcFormat*** is the character print format;
- The parameter ***pvArg0~pvArg9*** are print parameters;
- The parameter ***blsNeedHeader*** indicates whether to print the log header information.

Although SylixOS provides these print log functions to the application layer, they are often used in kernel space (such as the function `printk` for driver development). In fact, POSIX already provides a method for printing logs at the application layer, and SylixOS also provides support for the POSIX logging system.

17.2 POSIX logging system

When there is no control terminal, we can neither write the error message to the standard error nor write it to the specified file. Therefore, there is a need to have a centralized method of recording the error message. Syslog can not only write error messages to the terminal or the specified file, but also send them to the specified host.

Syslog is an industry-standard protocol that can be used to record device logs. In network devices such as routers and switches, system logs can record events that occur at any time in the system. Users can keep track of system statuses by viewing system records. The operating system records system-related events and application runtime events via system daemons or system threads. Users can also implement the machine-to-machine communication in the syslog protocol with proper configuration, and track the relevant status of devices and networks by analyzing the network behavior logs.

The syslog protocol provides a delivery method that allows a device to transfer event information over the network to event information receivers (also known as log servers). Since each process, application, and operating system is more or less independently completed, there are some inconsistencies in the syslog message, so there is no specification for the format or content of the information in the protocol. This protocol is simply designed to transfer event information. In fact, syslog message can be transferred without configuring the receiver or even without the receiver. Conversely, the receiver can also receive information without being clearly configured or defined.

Almost all network devices can send log message via syslog protocol to the remote server via UDP protocol. The remote receiving log server must monitor the UDP port (514) through the syslog daemon and process the log message of the local and receiving access system according to the configuration in `syslog.conf`, and write the specified event to a specific file for the backend database to manage and respond. That is, all events can be logged to one or more servers so that the backend database can analyze the events offline.

The device must be configured with some rules to display or transfer event information. To send log message to the syslog receiver generally requires the following steps:

- Decide which message to send;
- Determine the level to send;

- Define a remote receiver.

The format of the transferred syslog message mainly consists of three parts: PRI, HEADER, and MSG. The length of the data packet is less than 1024 bytes. The PRI section must have 3 to 5 characters, beginning with a “<” followed by a number and ending with a “>”. The number in the parenthesis is called Priority and consists of two values: facility and severity.

The facility name and its description are shown in table 17.1.

Table 17.1 facility parameters

facility	Description
LOG_AUTO	Certification related logs
LOG_KERN	Kernel related logs
LOG_MAIL	Mail related logs
LOG_DAEMON	Daemon related logs
LOG_USER	User related logs
LOG_SYSLOG	syslog related logs
LOG_LPR	Print related logs
LOG_NEWS	News related logs
LOG_UUCP	Unix to Unix cp related logs
LOG_CRON	Task schedule related logs
LOG_AUTHPRIV	Permission and authorization related logs
LOG_FTP	FTP related logs
LOG_LOCAL0~LOG_LOCAL7	User defined

Each message Priority contains a decimal Severity parameter that describes the message of Severity levels.

Table 17.2 Severity message

Level	Description
LOG_EMERG	cause the system to be unavailable
LOG_ALERT	must be solved immediately
LOG_CRIT	a worse situation
LOG_ERR	error message
LOG_WARNING	may affect the system functionality
LOG_NOTICE	not affecting the system functionality but is worth noting
LOG_INFO	general information
LOG_DEBUG	program or system debugging information

Note: Priority = facility | Severity value.

The HEADER part consists of two fields called **TIMESTAMP** and **HOSTNAME**. Immediately after the ">" at the end of the PRI is a **TIMESTAMP**, and any **TIMESTAMP** or **HOSTNAME** field must be followed by a space character. **HOSTNAME** contains the host name. If there is no host name or the host name is not recognized, the IP address is displayed. If a host holds multiple IP addresses, the one used to transfer information is usually displayed. **TIMESTAMP** is the local time. It uses the format "Mmm dd hh:mm:ss" to represent the month, day, hour, minute and second.

MSG is the rest part of the syslog packet. It usually contains additional information that generates the information process, as well as the textual part of the information. The **MSG** includes two domains, **TAG** and **CONTENT**. The value of **TAG** is the name of the program or process that generated the information, and **CONTENT** contains the details of this information. Traditionally, the format of this domain is more liberal and gives some time-specific information. **TAG** is an alphanumeric string of not more than 32 characters. Any non-alphanumeric character will terminate the **TAG** domain and be assumed to be the start of the **CONTENT** domain.

SylixOS checks the environment variable **SYSLOGD_HOST**. If it is a valid syslog server such as **SYSLOGD_HOST="192.168.0.1:514"**, the message will be sent to the syslog server. If the server needs to be re-determined, it is necessary to set the environment variable first, and then call **closelog**, so that the system will re-determine the server when sending the next message.

Call the following function to connect to a log server.

```
#include <syslog.h>

void    openlog(const char *ident, int logopt, int facility);

void    syslog(int priority, const char *message, ...);
```

The prototype analysis of the function **openlog**:

- The parameter ***ident*** is the prefix of each message;
- The parameter ***logopt*** is the option flag, as shown in table 17.3;
- The parameter ***facility*** is the capability parameter, as shown in table 17.1.

The prototype analysis of the function **syslog**:

- The parameter **priority** is the log priority, as shown in table 17.2;
- The parameter ***message*** is the log message;
- The parameter **...** is the variable parameter that can transfer more parameters.

Table 17.3 Option flag

Option flag	Description
LOG_PID	Each message contains a process ID
LOG_CONS	The terminal displays the sending message.

LOG_ODELAY	Delay when opening a connection
LOG_NDELAY	No delay when opening a connection
LOG_NOWAIT	Do not wait for child processes
LOG_PERROR	Simultaneous printing to standard error

Call the `closelog` function to disconnect when no messages need to be sent to the log server.

```
#include <syslog.h>

void    closelog(void);
```

Each sending log process has a log priority mask that determines which logs can be sent by syslog and which cannot. Call the `setlogmask` function to change the priority mask and return the previous one.

```
#include <syslog.h>

int    setlogmask(int maskpri);
```

The prototype analysis of the function `setlogmask`:

- This function returns the previous mask value.
- The parameter *maskpri* is the new mask value.

Since all of the above function calls need to operate on global variables, they are not thread-safe in a multithreaded environment. Syslog adds `syslog_data` structure to solve this problem. The structure is described as follows:

```
struct syslog_data {

    int            log_file;

    int            connected;

    int            opened;

    int            log_stat;

    const char     *log_tag;

    int            log_fac;

    int            log_mask;

};
```

- `log_file`: socket;
- `connected`: established a connection;
- `opened`: open flag;

- log_stat: option flag;
- log_tag: TAG domain value;
- log_fac: facility value;
- log_mask: priority mask.

The following is the syslog reentrant version function:

```
#include <syslog.h>

int  setlogmask_r(int maskpri, struct syslog_data *data);

void  syslog_r(int priority, struct syslog_data *data,
              const char *message, ...);
```

The prototype analysis of the function setlogmask_r:

- This function returns the previous mask on success, returns -1 and sets the error number on failure;
- The parameter *maskpri* is the new mask;
- The output parameter *data* is the syslog_data structure pointer.

The prototype analysis of the function syslog_r:

- The parameter *priority* is the Severity priority value, as shown in table 17.2;
- The parameter *data* is the syslog_data structure pointer that requires the application to fill in the appropriate value;
- The parameter *message* is the log message sent.

The following programs show the use of syslog. The syslog receiver program communicates by creating UNIX domain sockets. The default file name of UNIX domain socket for syslog is "/dev/log". In the syslog log sender program, after setting the option flag LOG_CONS of the openlog function for debugging purposes, the message sent by syslog will be displayed on the terminal.

Program List 17.1 syslog receiver program

```
#include <stdio.h>

#include <socket.h>

#include <sys/un.h>

#include <syslog.h>

int main (int argc, char *argv[])

{
```

```
int sockfd;

struct sockaddr_un unixaddr, unixfrom;

socklen_t fromlen = sizeof(unixaddr);

socklen_t len = fromlen;

int ret;

char buf[LOG_DEFAULT_SIZE] = {0};

sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);

if (sockfd < 0) {
    perror("socket");
    return (-1);
}

unixaddr.sun_family = AF_UNIX;

strcpy(unixaddr.sun_path, "/dev/log");

unixaddr.sun_len = (uint8_t)(SUN_LEN(&unixaddr));

ret = bind(sockfd, (struct sockaddr *)&unixaddr, len);

if (ret < 0) {
    perror("bind");
    return (-1);
}

while (1) {
    ssize_t len;

    len = recvfrom(sockfd, buf, sizeof(buf), 0,
                  (struct sockaddr *)&unixfrom, &fromlen);

    fprintf(stdout, "MSG len: %ld\n", len);

    fprintf(stdout, "[R-MSG]: %s", buf);
}
```

```
    }  
  
    return (0);  
}
```

Program List 17.2 syslog log sender program

```
#include <stdio.h>  
  
#include <syslog.h>  
  
int main (int argc, char *argv[])  
{  
    int i;  
  
    openlog("SylixOS-SYSLOG", LOG_CONS | LOG_PID, LOG_USER);  
  
    for (i = 0; i < 10; i++) {  
        syslog(LOG_INFO, "[%d]syslog running...\n", i);  
        sleep(1);  
    }  
  
    closelog();  
  
    return (0);  
}
```

Run the program in the SylixOS Shell. First start the receiver program, and then start the sender program, the results are shown below:

Sender display:

```
<14>Oct 29 01:39:11 SylixOS-SYSLOG[4010043]:[0]syslog running...  
<14>Oct 29 01:39:12 SylixOS-SYSLOG[4010043]:[1]syslog running...  
<14>Oct 29 01:39:13 SylixOS-SYSLOG[4010043]:[2]syslog running...  
.....
```

Receiver display:

```
MSG len: 65
```

```
[R-MSG]: <14>Oct 29 01:49:35 SylixOS-SYSLOG[4010046]:[0]syslog running...
```

```
MSG len: 65
```

```
[R-MSG]: <14>Oct 29 01:49:36 SylixOS-SYSLOG[4010046]:[1]syslog running...
```

```
MSG len: 65
```

```
[R-MSG]: <14>Oct 29 01:49:37 SylixOS-SYSLOG[4010046]:[2]syslog running...
```

```
.....
```

Chapter 18 Multi-user Management

18.1 Introduction of POSIX User Management

Users and user groups are an important part of SylixOS system management and also the basis of system security. In SylixOS, all files and programs belong to a specific user. Each file and program has certain access rights for limiting different users' access behaviors. The SylixOS system divides users into different user groups based on certain principles. SylixOS supports multi-user management and conforms to the UNIX multi-user management standard.

18.1.1 Users

Users are usually denoted by a UID (User Identifier), which represents a combination of authorities. Managing users and user groups is an important task of a system administrator. The UID uniquely identifies a specific user within the SylixOS system. For the users, however, remembering the UID is not easy because the UID is actually an integer value. In order to solve this problem, there is also a concept in the SylixOS system that is very close to the user, namely the login name. The login name is a string which is specified for a user by the system administrator when the user is created. Generally, the login name has a clear meaning. For the sake of safety, each login usually has its own password, and the user can log in to the SylixOS system with its login name and password. If the login name or password is incorrect, the SylixOS system will reject the login of the user.

For example, the user "root" has its user name "root" and a password. The "root" can be connected to the SylixOS system via Telnet, enter the username "root" and the password "root" to log in to SylixOS.

In SylixOS, common users' operations are subject to certain restrictions. For example, you cannot access other users' home directories, nor access the unauthorized files or directories. When switching from a standard user to another user's execution environment, we must know the user's password.

In SylixOS, we can use the user command to view all the user information.

[Command format]

```
user [genpass]
```

[Common Options]

```
None
```

[Instructions for parameters]

```
[genpass] : Whether to generate encrypted password information
```

The following is the output of the **user** command.

```
# user
login: root
password:
  USER      ENABLE  UID   GID
-----
root         yes     0     0
sylixos     yes     1     1
apps        yes     2     2
hanhui      yes     2000  2
tty         no      3     3
anonymous   no      4     4
```

18.1.2 User Groups

A user group is a collection of users with the same or similar functions. Creating a separate user group for certain users can facilitate the management of these users. For example, the users in a group are uniformly authorized through the user group. In addition, the users can be identified through the user group.

Similar to users, within the SylixOS system, each user group is assigned with a group identifier, abbreviated as GID. GID is an unsigned integer value that uniquely identifies a user group within the system. In addition, a user group also has a group name and a list of users. Same as the user's login name, the group name is also a string with a clear meaning. The list of users is a list of all the users belonging to the group.

In SylixOS, you can use the **group** command to view all the group information.

[Command format]

```
group
```

[Common Options]

```
None
```

[Instructions for parameters]

```
None
```

The following is the output of the **group** command:

```
# group
login: root
password:
  GROUP      GID      USERS
-----
root         0      root,
sylixos     1      sylixos,
```

```

apps          2  apps, hanhui, sylixos,
tty           3  tty,
anonymous     4  anonymous,

```

In SylixOS, using the **who** command can view the users who are currently logged in.

[Command format]

who

[Common Options]

None

[Instructions for parameters]

None

The following is the output of the **who** command:

```

# who
user:root terminal:/dev/ttyS0 uid:0 gid:0 euid:0 egid:0

```

18.2 Management of POSIX Authority

When a file is created, the system adds owner and group information to the user who created the file and the private group file where the user is located, and sets its default access authorities. The owner is often referred to as the file owner in the Linux system. The file owner can perform all operations on the file, including reading, modifying and deleting.

Note: In SylixOS, the root user has the highest authority, so it can read, modify, and delete any file in the system.

The **//** command can be used to view file details in the SylixOS system.

[Command format]

// [path name]

[Common Options]

None

[Instructions for parameters]

[path name]: path name

The following is the output of the **//** command:

```

# // /dev/
srwxrwxrwx root    root    Tue Jun 16 17:22:38 2015    0 B,  log
srw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B,  socket

```

```

cr--r--r-- root    root    Tue Jun 16 17:22:38 2015  4096 B, netevent
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015  750KB, fb0
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, ttyS2
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, ttyS1
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, ttyS0
cr--r--r-- root    root    Tue Jun 16 17:22:38 2015    0 B, urandom
cr--r--r-- root    root    Tue Jun 16 17:22:38 2015    0 B, random
drw-rw-rw- root    root    Tue Jun 16 17:22:38 2015      shm/
crw-r--r-- root    root    Tue Jun 16 17:22:38 2015    0 B, rtc
cr--r--r-- root    root    Tue Jun 16 17:22:38 2015  4096 B, hotplug
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, epollfd
drw-rw-rw- root    root    Tue Jun 16 17:22:38 2015      gpiofd/
cr--r--r-- root    root    Tue Jun 16 17:22:38 2015    0 B, signalfd
cr--r--r-- root    root    Tue Jun 16 17:22:38 2015    0 B, hstimerfd
cr--r--r-- root    root    Tue Jun 16 17:22:38 2015    0 B, timerfd
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, eventfd
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, zero
crw-rw-rw- root    root    Tue Jun 16 17:22:38 2015    0 B, null
drwxr-xr-- root    root    Tue Jun 16 17:22:38 2015      input/
drwxr-xr-- root    root    Tue Jun 16 17:22:38 2015      pipe/
drwxr-xr-- root    root    Tue Jun 16 17:22:38 2015      pty/

total items : 23

```

It can be seen that both the owner and the group of these files are root (the first root is the owner, and the second root is the group).

By default, the system will give the members of the user's group some authorities (in the form of group's authorities). If the user is a member of multiple groups, the system will use the group saved in the user file `/etc/passwd` (i.e. private group).

18.2.1 File Authority and Expression Method

Although the traditional file authorities in the SylixOS system only include three types, i.e. reading, writing, and executing, the three have different meanings for files and directories.

For files:

- Read: Allows reading of file contents, viewing, copying, etc.
- Write: Allows writing contents, editing, appending, deleting of files, etc.
- Execution: If the file is an executable script, binary code file, or program, this authority controls whether the user can execute the file.

For directories:

- Read: Allows the user to view the list of files in the directory, such as `ls`;

- **Write:** Allows the user to create and delete files in the directory. When deleting a file in a directory, we should also have the corresponding write authority of the file.
- **Execution:** Allows the user to access the directory using the **cd** command.

Although the user may not be able to access the directory or view the list of files in the directory, if the file authority in the directory allows, the user can still manipulate the file by entering the full path.

The authority expression methods in symbolic mode: read: r, write: w, execute: x.

The authority expression methods in absolute mode: read: 4, write: 2, execute: 1.

18.2.2 File Authority Management Command **chmod**

The file authority management is performed by the owner of the file or the root user, and the **chmod** command can change the file's authority information.

[Command format]

```
chmod① newmode filename
```

[Common Options]

None

[Instructions for parameters]

```
newmode: Permission expression, permission expression using absolute mode
filename: file name
```

The following is the result of the execution of the **chmod** command:

```
# cd tmp/
# ls
.qt_soundserver-0          qtembedded-0
# mkdir user
# ls
user          .qt_soundserver-0          qtembedded-0
# ll
drwxr-xr-- root      root      Tue Jun 16 22:19:38 2015          user/
-rw----- root      root      Tue Jun 16 20:22:18 2015          0 B, .qt_soundserver-0
drwx----- root      root      Tue Jun 16 20:23:26 2015          qtembedded-0/
      total items : 3
# chmod 070 user/
# ll
```

① Chmod does not currently support setting authorities in symbolic mode.

```

dr--rwx--- root    root    Tue Jun 16 22:19:38 2015      user/
-rw----- root    root    Tue Jun 16 20:22:18 2015      0 B, .qt_soundserver-0
drwx----- root    root    Tue Jun 16 20:23:26 2015      qtembedded-0/
total items : 3

```

18.3 User Management-related Files in the /etc Directory

With the **ls** command, we can view the file information in the directory.

[Command format]

```
ls [path name]
```

[Common Options]

None

[Instructions for parameters]

```
[path name]: path name
```

The following is the execution result of viewing the /etc directory with the **ls** command:

```

# cd /etc/
# ls
passwd          group          shadow         fs_init.sh    qtenv.sh
qtl_n_4.8.6.sh  startup.sh     profile        pointercal

```

The user account information in the SylixOS system is maintained jointly by the /etc/passwd and /etc/shadow files, and the user group information is maintained by the /etc/group file. In these two files, each user has a corresponding record. After the user name and password are entered as prompted using a terminal registration such as a console, the system will check the /etc/passwd file based on the user name provided by the user, and then make a comparison with the password field in the /etc/shadow file after encrypting the file using the same encryption algorithm according to the password provided by the user and also check other fields like the password expiration. If passing the verification code, the user can access its own home directory according to the home directory and command interpreter specified in the /etc/passwd file.

In addition to the user name and password, the user ID, user group ID, home directory, and command interpreter are stored in the /etc/passwd and /etc/shadow files, respectively.

18.3.1 /etc/passwd File

The /etc/passwd file contains the main user information except the password in the SylixOS system. Each user's information occupies one line and each line consists of

seven fields with a colon ":" as a delimiter. / The format of the etc/passwd file is defined as follows:

```
username:password:uid:gid:comment: login info:home_dir:login_shell
```

- Username: the user name, consisting of more than 2 characters, and unique in SylixOS;
- Password: This field was originally a user password, but the password has now been moved into the /etc/shadow file. Therefore, if the user has a password, this field will contain a lowercase "x". The encrypted password is stored in the /etc/shadow file. If this field is "!", it indicates that the corresponding user cannot register to the system normally without a password. If the content of this field is neither of the above two cases, it indicates that the corresponding user is in the prohibited state;
- Uid: User ID (user's identification). The user ID is the unique numeral ID assigned to each user by the system. It is the primary means by which the system identifies each user. When the system needs to understand the user information (such as the contents of the account field), it usually uses the user ID as an index to retrieve the /etc/passwd file. The user ID is a 32-bit unsigned integer, where 0 to 99 are reserved for the system user and the custom common user ID should be within the range of 100 to 60000. Taking into account the compatibility with other systems, SylixOS recommends using 65535, the maximum of 16-bit unsigned integers, as the upper limit for the user ID;
- Gid: User group ID (user group identification). Each user in the SylixOS system should belong to a user group. Each user group has a corresponding user group ID in addition to the group name. Similarly, ID numbers 0 to 99 are reserved for system users;
- Comment: comment information, usually including the user information like user's full name and user's role;
- Login info: login information;
- Home_dir: User's home directory (full path name). A user's home directory is a sub-directory assigned to the user, and used for the user to store personal files and is the initial working directory after the user is registered. Usually the SylixOS system adopts the user home directory structure in the form of /home/username, where username is the registered user name;
- Login_shell: Specifies the shell (command interpreter) invoked by the user after registration.

These fields are contained in the passwd structure defined in <pwd.h>, the correspondence is shown in Table18.1.

Table18.1 User Password Parameters

Functional description	Struct passwd member	/etc/passwd field
User name	pw_name	username
Encrypted password	pw_passwd	password
User ID	pw_uid	uid
User group ID	pw_gid	gid
Comments	pw_comment	comment
Login information	pw_gecos	login info
Initial working directory	pw_dir	home_dir
Initial Shell	pw_shell	login_shell

The initial contents of the `/etc/passwd` file in the SylixOS system are as follows:

```
# cat passwd
root:x:0:0:root::/root:/bin/sh
sylixos:x:1:1:developer::/home/sylixos:/bin/sh
apps:x:2:2:application::/home/apps:/bin/sh
tty:!:3:3:tty owner::/home/tty:/bin/false
anonymous:!:4:4:anonymous user::/home/anonymous:/bin/false
```

18.3.2 /etc/shadow File

`/etc/shadow` is a system file that restricts access by common users. It contains encrypted passwords and other related information. Corresponding to the `/etc/passwd` file, the password information of each user in the `/etc/shadow` file occupies one line, and each line consists of 9 fields, with a colon ":" as a delimiter. Its file format is defined as follows:

```
username:password:lastchanged:mindays:maxdays:warn:inactive:expire:reserve
```

- Username: user name, see the corresponding part of the `/etc/passwd` file;
- Password: The encrypted password (generated by the `crypt_safe` function). If this field is "!", it means that the corresponding user has not set a password;

- **Lastchanged:** the number of days from January 1st, 1970 to the date on which the password was changed for the last time;
- **mindays:** The minimum number of days for which the password remains stable. Only when this limit is exceeded can the password be changed. This field must be greater than or equal to 0 to enable the password expiration examination;
- **maxdays:** The maximum number of days to keep the password valid. If this limit is exceeded, the system will force the user to change the password;
- **warn:** Specifies how many days in advance a warning should be made to the user before the expiration of the password;
- **inactive:** Specifies the maximum number of days for which the account cannot be accessed after the expiration date of the password expires, but the account information is still valid. When this limit is exceeded, this user's account will be disabled.
- **expire:** Specifies the user account's expiration date. Upon the expiration, the account will automatically expire and the user can no longer register with the system;
- **reserve:** Reserved field.

These fields are contained in the `spwd` structure defined in `<shadow.h>`, the correspondence is shown in Table 18.2.

Table18.2 User Shadow Password Parameter

Functional description	Struct <code>spwd</code> member	<code>/etc/shadow</code> field
User name	<code>sp_namp</code>	<code>username</code>
Encrypted password	<code>sp_pwdp</code>	<code>password</code>
Time elapsed since the password was last changed	<code>sp_lstchg</code>	<code>lastchanged</code>
How soon it is allowed to change	<code>sp_min</code>	<code>mindays</code>
The number of days remaining for request for change	<code>sp_max</code>	<code>maxdays</code>
Number of days for due warning	<code>sp_warn</code>	<code>warn</code>
Number of days left before account inactivity	<code>sp_inact</code>	<code>inactive</code>

Number of days to account expiration	sp_expire	expire
Reserved	sp_flag	reserve

The initial contents of the `/etc/shadow` file in the SylixOS system are as follows:

```
# cat shadow
root:$1$qY9g/6K4$/FKP3w1BsziKGCP3uLDnG.:0:0:99999:7:::
sylixos:$1$qY9g/6K4$WFEEx17sXu/3aL3wE.u8NZ1:0:0:99999:7:::
apps:$1$qY9g/6K4$buV57yqE0kMbApOVI/jKM1:0:0:99999:7:::
anonymous:!!:0:0:99999:7:::
```

18.3.3 /etc/group File

The `/etc/group` file contains the user group information. Each user group's information occupies one line and each line consists of three fields with a colon ":" as a delimiter. / The format of the `etc/passwd` file is defined as follows:

```
username:password:gid:members
```

- Username: the user name, consisting of more than 2 characters, and unique in SylixOS;
- Password: This field was originally a user password, but the password has now been moved into the `/etc/shadow` file. Therefore, if the user has a password, this field will contain a lowercase "x". The encrypted password is stored in the `/etc/shadow` file. If this field is "!!", it indicates that the corresponding user cannot register to the system normally without a password. If the content of this field is not either of the above two cases, it indicates that the corresponding user is in the prohibited state;
- Gid: User group ID (user group identification). Each user in the SylixOS system should belong to a user group. Each user group has a corresponding user group ID in addition to the group name. Similarly, ID numbers 0 to 99 are reserved for system users;
- members: User members of this user group.

These fields are contained in the group structure defined in `< grp.h>`, and the correspondence is as shown in Table 18.3.

Table18.3 User group parameters

Functional description	Struct group members	/etc/group field
User name	gr_name	username
Encrypted password	gr_passwd	password
User group ID	gr_gid	gid
User group members	gr_mem	members

The initial contents of the /etc/group file in the SylixOS system are as follows:

```
# cat group
root:x:0:root
sylixos:x:1:sylixos
apps:x:2:apps,hanhui,sylixos
tty:x:3:tty
anonymous:x:4:anonymous
```

18.4 POSIX User Operations

18.4.1 User Password Operation

The SylixOS system defines two functions for obtaining password file entries. After the user gives the user name or user ID, the following two functions can be used to query the user information.

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
               size_t bufsize, struct passwd **result);
```

Prototype analysis of the function getpwuid_r:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter *uid* is the user ID;
- The output parameter *pwd* returns user information;
- The parameter *buffer* is a buffer;
- The parameter *bufsize* is the size of the buffer;

- The output parameter **result** returns a user information pointer.

The function `getpwuid_r` will search the user database file `/etc/passwd` to obtain the user information matching the parameter **uid**. The user information will be stored in the buffer pointed to by **buffer** and update the **pwd** structure. If a matching user is found, the **pwd** pointer will be returned after being stored in **result**, otherwise **result** will return NULL.

Note that the function `getpwuid` is not reentrant, so it is non-thread safe.

```
struct passwd *getpwnam(const char *name);
int getpwnam_r(const char *name, struct passwd *pwd,
               char *buffer, size_t bufsize, struct passwd **result);
```

Prototype analysis of the function `getpwnam_r`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **name** is the user name;
- The output parameter **pwd** returns user information;
- The parameter **buffer** is a buffer;
- The parameter **bufsize** is the size of the buffer;
- The output parameter **result** returns a user information pointer.

The function of the function `getpwnam_r` is similar to that of the function `getpwuid_r`, except that the parameter **name** is the user name that needs to be matched instead of the user ID (`getpwuid_r` the matched user ID). The function `getpwnam` is not reentrant.

If you only want to view the user name and user ID, then the above two functions can meet the requirements, but some programs need to view the entire password file, then you need to call the following three functions to achieve this goal.

```
#include <pwd.h>
struct passwd *getpwent(void);
void setpwent(void);
void endpwent(void);
```

When the function `getpwent` is called, it will return the next entry of record in the password file. Like the two functions described above, it returns a `passwd` structure pointer that is filled in by it. The first time the function is called, each file it uses is opened and the `passwd` structure is overwritten each time this function is called. Note that there is no requirement for the order of the record entries of the password file when this function is used.

The function `setpwent` can point the read and write addresses of the `getpwent` function back to the beginning of the password file (usually called `rewind`), and `endpwent` closes the files. After viewing the password file using `getpwent`, be sure to call `endpwent`

to close the files. Getpwent only knows when it should open the file it uses (on the first call), but it doesn't know when it should close the file.

Calling setpwent at the beginning of the program is a self-protective measure to prevent the caller from having previously called the getpwent function to open the file and rewind the relevant file to locate it at the beginning of the file.

18.4.2 User Shadow Password Operation

Similar to a set of functions that accesses the password file, there is another set of functions for accessing the shadow password file.

```
#include <shadow.h>
void  setspent(void);
void  endspent(void);
struct spwd  *getspent(void);
struct spwd  *getspnam(const char *name);
int  getspent_r(struct spwd *result_buf, char *buffer,
               size_t buflen, struct spwd **result);
int  getspnam_r(const char *name, struct spwd *result_buf,
               char *buffer, size_t buflen, struct spwd **result);
```

Prototype analysis of the function `getspent_r`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The output parameter **result_buf** returns the content of the spwd structure;
- The parameter **buffer** is a buffer;
- The parameter **buflen** is the size of the buffer;
- The output parameter **result** returns the result.

Prototype analysis of the function `getspnam_r`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **name** is the user name;
- The parameter **result_buf** returns the content of the spwd structure;
- The parameter **buffer** is a buffer;
- The parameter **buflen** is the size of the buffer;
- The output parameter **result** returns the result.

Here is the implementation of the spwd structure in SylixOS:

```

struct spwd {
    char    *sp_namp;          /* user login name          */
    char    *sp_pwdp;         /* encrypted password       */
    long    sp_lstchg;        /* last password change     */

    int     sp_min;           /* days until change allowed. */
    int     sp_max;           /* days before change required */
    int     sp_warn;         /* days warning for expiration */
    int     sp_inact;         /* days before account inactive */
    int     sp_expire;        /* date when account expires */
    int     sp_flag;          /* reserved for future use   */
};

```

Note: The functions `getspent_r` and `getspnam_r` are reentrant.

18.4.3 User Group Operation

Similar to a set of functions that accesses password files, there is another set of functions for accessing user group files.

```

#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
int getgrnam_r(const char *name, struct group *grp,
               char *buffer, size_t bufsize, struct group **result);
int getgrgid_r(gid_t gid, struct group *grp,
               char *buffer, size_t bufsize, struct group **result);

```

Prototype analysis of the function `getgrnam_r`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter *name* is the group name;
- The output parameter *grp* returns the content of the group structure;
- The parameter *buffer* is a buffer;
- The parameter *bufsize* is the size of the buffer;
- The output parameter *result* returns the result.

Prototype analysis of the function `getgrgid_r`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **gid** is the group ID;
- The output parameter **grp** returns the content of the group structure;
- The parameter **buffer** is a buffer;
- The parameter **bufsize** is the size of the buffer;
- The output parameter **result** returns the result.

Here is the implementation of the group structure in SylixOS:

```
struct group {
    char      *gr_name;           /* group name           */
    char      *gr_passwd;        /* group password       */
    gid_t     gr_gid;           /* group id             */
    char      **gr_mem;          /* group members        */
};
```

Note: The `getgrnam_r` and `getgrgid_r` functions are reentrant.

18.4.4 User's Additional Group Operation

The group members may not only belong to the group corresponding to the group ID in the record entry of the password file, but also belong to NGROUPS_MAX (usually 16) additional groups. The check of file access authorities not only checks the effective group ID of the process but also checks the additional group ID. The advantage of using an additional group ID is that there is no need to explicitly change the group. A user may participate in multiple projects, so it is also necessary to belong to multiple groups at the same time. To obtain and set the additional group ID, SylixOS provides the following three functions:

```
#include <grp.h>
int  setgroups(int  groupsun, const gid_t grlist);
int  getgroups(int  groupsize, gid_t grlist);
int  initgroups(const char *name, gid_t basegid);
```

Prototype analysis of the function `setgroups`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **groupsun** is the number of groups;
- The parameter **grlist[]** is an array pointer that holds the user groups.

Prototype analysis of the function `getgroups`:

- This function returns the number of gids actually stored in the `grlist` array;

- The parameter **groupsize** is the size of the array **grlist[]**;
- The output parameter **grlist[]** stores the user group.

Prototype analysis of the function `initgroups`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter `name` is the group member name;
- The parameter **basegid** is the group ID.

The function `setgroups` fills **groupsun** additional group IDs into the array `grlist`. The array contains up to `NGROUPS_MAX` elements. The function `getgroups` obtains the group ID of the additional group to which the current process user ID belongs. When **groupsize** is 0 or **grlist** is NULL, the function will return the number of additional groups; the function `initgroups` reads the file `/etc/group` and adds the group ID of the existing user **name** to the process' additional group.

18.5 Multi-user Management Database

18.5.1 User Operation

Calling the function `user_db_uadd` in SylixOS creates a new account. To create an account, this function needs to access the `/etc/passwd` file and the `/etc/shadow` file. Therefore, adding a new account requires the existence of these two files in the `/etc` directory.

```
#include <userdb.h>
int user_db_uadd(const char *user, const char *passwd,
                int enable, uid_t uid, gid_t gid,
                const char *comment, const char *home);
```

Prototype analysis of the function `user_db_uadd`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **user** is the user name;
- The parameter **passwd** is the user password;
- The parameter **enable** indicates whether the user is enabled.
- The parameter **uid** is the user ID;
- The parameter **gid** is the user group ID;
- The parameter **comment** is the comment information;
- The parameter **home** is the user's home directory.

Note that calling the function `user_db_uadd` can only create a new account in an existing user group. If the specified group ID does not exist, the creation of a new account will fail.

Calling the function `user_db_umod` can modify the enable status, comments, and user home directory of an existing account,

```
#include <userdb.h>
int user_db_umod(const char *user, int enable,
                const char *comment, const char *home);
```

Prototype analysis of the function `user_db_umod`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **user** is the user name;
- The parameter **enable** indicates whether the user is enabled.

- The parameter **comment** is the comment information;
- The parameter **home** is the user's home directory.

Calling the function `user_db_uget` can obtain the specified account information, as described below:

```
#include <userdb.h>
int user_db_uget(const char *user, int *enable,
                uid_t *uid, gid_t *gid,
                char *comment, size_t sz_com,
                char *home, size_t sz_home);
```

Prototype analysis of the function `user_db_uget`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **user** is the user name;
- The output parameter **enable** indicates whether the user is enabled.
- The output parameter **uid** is the user ID;
- The output parameter **gid** is the user group ID;
- The output parameter **comment** returns the comment information;
- The parameter **sz_com** is the size of the comment buffer;
- The output parameter **home** returns to the user's home directory;
- The parameter **sz_home** is the size of the user's home directory buffer.

Calling the function `user_db_udel` can delete an existing account.

```
#include <userdb.h>
int user_db_udel(const char *user);
```

Prototype analysis of the function `user_db_udel`:

- This function returns 0 if it succeeds, and returns -1 and set the error number if it fails;
- The parameter **user** is the user name.

18.5.2 Group Operation

The following function provides the operation method of the user group in SylixOS. The function `user_db_gadd` adds a new user group to SylixOS. The function `user_db_gget` can obtain the group ID of the specified user group **group**. The function `user_db_gdel` will delete a specified group group.

```
#include <userdb.h>
int user_db_gadd(const char *group, gid_t gid);
int user_db_gget(const char *group, gid_t *gid);
int user_db_gdel(const char *group);
```

Prototype analysis of the function :

- The above function returns 0 when it succeeds, and returns -1 and sets the error number if it fails.
- The parameter **group** is the group name;
- The parameter **gid** is the group ID, and the **gid** parameter of the user_db_gget function is used to store the returned group ID.

18.5.3 Password Operation

```
#include <userdb.h>
int user_db_pmod(const char *user,
                const char *passwd_old,
                const char *passwd_new);
```

Prototype analysis of the function user_db_pmod:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter **user** is the user name;
- The parameter **passwd_old** is the previous password;
- The parameter **passwd_new** is the new password.

Calling the function user_db_pmod can modify the password of the specified user. In particular, if the parameter passwd_new is NULL, the user password is cleared.

18.5.4 User Shell Commands

In a UNIX system, the root account has the "Supreme" authority, that means, an account with root authority can do anything and even destroy the system. Similarly, SylixOS also has a similar authority management mechanism. An account with SylixOS root authority can do anything in SylixOS (such as delete other accounts or the underlying work). Therefore, the user management requires that the common user's operation authority is lower than that of the root user. The following is the Shell command to add, modify, and delete user in SylixOS.

uadd^① command can add new users to SylixOS:

[Command format]

```
uadd name password enable[0 / 1] uid gid comment homedir
```

[Common Options]

None

[Instructions for parameters]

```

name : Created account name
password : New account password
enable : Whether to enable new account (0 is not enabled, 1 is enabled)
uid : New account ID
gid : New account group ID
comment : Comment information
homedir : New account home directory (eg /home/flags)

```

The following command can add new account flags to SylixOS:

```
# uadd flags 123456 1 10 2 a_new_user /home/flags
```

The above command displays: new account flags, the account password is 123456, enabling this account (1), the account ID is 10, the group ID is 2, and the home directory of the account is /home/flags. This is a new account as noted in the comment.

The **umod** command can modify an existing account:

[Command format]

```
umod name enable[0 / 1] comment homedir
```

[Common Options]

None

[Instructions for parameters]

```

name : account name
enable : Whether to enable the account (0 is not enabled, 1 is enabled)
comment : Comment information
homedir : Account home directory

```

The following is about how to modify an account's information. The command displays that the status of the account flags is set to be disabled. The account set to be disabled will have no authority to operate SylixOS.

```
# umod flags 0 a_mod_user /home/flags
```

The **udel** command can delete an existing account:

[Command format]

```
udel name
```

[Common Options]

None

[Instructions for parameters]

```
name : account name
```

The **gadd** command adds a new user group to SylixOS;

[Command Format]

```
gadd group_name gid
```

[Common Options]

```
None
```

[Instructions for parameters]

```
group_name : User group name  
gid : User group ID
```

The following command adds a new group grp(ID is 11) to SylixOS.

```
# gadd grp 11
```

The **gdel** command deletes an existing group. Note that the deleted group must be a group without user (that is, an empty group):

[Command Format]

```
gdel group_name
```

[Common Options]

```
None
```

[Instructions for parameters]

```
group_name : User group name
```

The following command is to delete the group grp:

```
# gdel grp
```

The **pmod** command can modify the password of a specified user:

[Command Format]

```
pmod name old_password new_password
```

[Common Options]

```
None
```

[Instructions for parameters]

```
name : user name  
old_password : old password  
new_password : new password
```

The following command changes the password of the user flags from 123456 to abcd:

```
# pmod flags 123456 abcd
```

Chapter 19 Dynamic Loading

19.1 Principle of Dynamic Link Library

19.1.1 Format of ELF File

The ELF (Executable and Linking Format) file are generated by the compiler and linker and used to save binary programs and data to facilitate the processor to load the executed file format. Originally it was developed and published by the UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). The ELF File formats include three types:

- Relocatable File, including code and data that can be linked with other object files to create executable files or shared object files;
- Executable File, containing a program that can be executed. This file specifies how the function exec creates a process image of a program.
- Shared Object File, containing the code and data that can be linked in two types of contexts. Firstly, the linker can process it with other relocatable files to generate another object file. Secondly, the Dynamic Linker can combine it with an executable file and other shared objects to create a process image.

19.1.2 ELF Files in SylixOS

The ELF files in SylixOS are the following:

- **Kernel Module File** (ending with .ko): Generated by linking the object file compiled from the source file, belonging to "relocatable file";
- **Executable File**: Generated by the link of the object file obtained by compiling, it is a position-independent "shared object file", and the application file must specify the program entry (usually the function "main");
- **Dynamic Link Library File** (ending with .so): Generated by the link of the object file obtained by compiling, it is a position-independent "shared object file", but has no program entry;
- **Statically Linked Library File** (ending with .a): Generated by using the archive command (ar) based on the object file obtained by compiling, used for program linking.

After the SylixOS application source code is written, we must first use gcc to compile the source file into the intermediate object file, and then link it into the kernel module, application, or library file according to the actual situation. The flow is shown in Figure 19.1:

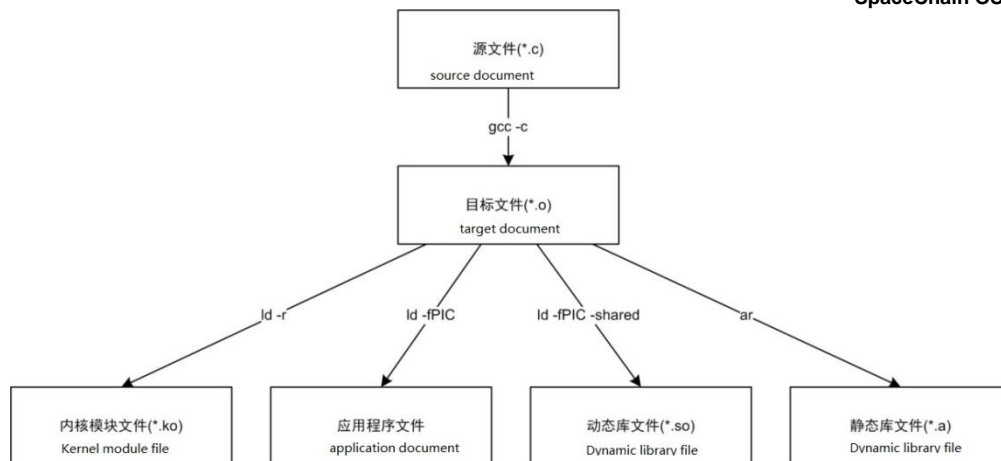


Figure 19.1 SylixOS ELF File Generation Process

19.1.3 SylixOS Dynamic Loader Features

The SylixOS dynamic loader has the following features:

- Supports loading of kernel modules, location-independent executables, and dynamic libraries;
- Supports loading of the library file on which the application depends when the application is being loaded, and automatically resolves the dependent relationship;
- Supports the manual loading through the operating system interface during program operation;
- Supports the automatic construction and destruction operations of C + + global object, and supports C + + exception handling.

19.2 Autoloading of Dynamic Library

19.2.1 Linking of Dynamic Library

Autoloading of dynamic library is automatic loading of the depended libraries before the program is run, and as for which library files will be depended, it is determined during linking. For example, running the following link command can link libvpmpdm.so, libsubfun.so, libm.a, libgcc.a to the app executable. The linker loads dynamic or static libraries based on specific conditions. In this example, libm.a and libgcc.a are self-contained static libraries of the compiler.

```
arm-sylixos-eabi-g++ -mcpu=cortex-a8 -nostdlib -fPIC -shared -o app app.o
-lvpmpdm -lsubfun -lm -lgcc
```

After the link is complete, we can use the `arm-sylixos-eabi-readelf` command to view the dynamic libraries that the application depends on.

```
windows>① arm-sylixos-eabi-readelf -d app
Dynamic section at offset 0x2cc contains 12 entries:
  Tag          Type              Name/Value
0x00000001 (NEEDED)         Shared library: [libvpmpdm.so]
0x00000001 (NEEDED)         Shared library: [libsubfun.so]
0x00000004 (HASH)           0x94
0x00000005 (STRTAB)        0x1d4
0x00000006 (SYMTAB)        0xe4
0x0000000a (STRSZ)         133 (bytes)
0x0000000b (SYMENT)        16 (bytes)
0x00000003 (PLTGOT)        0x8354
0x00000002 (PLTRELSZ)      8 (bytes)
0x00000014 (PLTREL)        REL
0x00000017 (JMPREL)        0x25c
0x00000000 (NULL)         0x0
```

19.2.2 Downloading of Dynamic Library

Through the use of RealEvo-IDE, a dynamic library can be downloaded to the SylixOS system. Before downloading, we need to determine the path of the dynamic library file in the SylixOS system. The search paths of the application dynamic libraries in SylixOS are as follows in sequence:

- Shell current directory;
- The search path contained in the `LD_LIBRARY_PATH` environment variable;
- The search path included in the `PATH` environment.

The paths in the above environment variables are separated by `:"`. We can use the `env` command to view the SylixOS environment variables as follows:

```
# env
variable show >>
-----
VARIABLE          REF              VALUE
-----
TERMCAP           /etc/termcap
TERM              vt100
PATH_LOCALE       /usr/share/locale
LC_ALL
```

^① In this book, `windows>` is used to indicate the command that is operated in the Windows environment.

LANG	C
LD_LIBRARY_PATH	/usr/lib:/lib:/usr/local/lib
PATH	/usr/bin:/bin:/usr/pkg/sbin:/usr/local/bin
NFS_CLIENT_PROTO	udp
NFS_CLIENT_AUTH	AUTH_UNIX
SYSLOGD_HOST	0.0.0.0:514
FIO_FLOAT	1
SO_MEM_PAGES	8192
TSLIB_CALIBFILE	/etc/pointercal
TSLIB_TSDEVICE	/dev/input/touch0
MOUSE	/dev/input/mouse0:/dev/input/touch0
KEYBOARD	/dev/input/keyboard0
TZ	CST-8:00:00
TMPDIR	/tmp/
LICENSE	SylixOS license: BSD/GPL.
VERSION	1.2.1
SYSTEM	SylixOS kernel version: 1.2.1 NeZha(a)

19.2.3 Loading of Kernel Module

The kernel module does not attach to any application and cannot be loaded automatically when the application is started. To autoload kernel modules, we can use the load command in the SylixOS startup script. The SylixOS kernel module loader cannot solve the dependencies between kernel modules. We need to decide the loading sequence of the modules by our own. SylixOS does not automatically search for a path, and the path must be specified in the load command. Normally SylixOS kernel modules are stored in the /lib/modules directory or its subdirectories. The loading method of a kernel module is shown in 19.4.2 Loading Kernel Modules.

19.3 POSIX Dynamic Link Library API

19.3.1 Common API of Dynamic Library

1. Loading of Dynamic Library

```
#include <dlfcn.h>
void *dlopen(const char *pcFile, int iMode);
```

Prototype analysis of the function dlopen:

- The module handle is returned if this function succeeds, and NULL is returned and the error number is set if this function fails.
- The parameter *pcFile* is the name of the dynamic library file;

- The parameter ***iMode*** express the library's load property with a mask.

Calling the function `dlopen` will open a dynamic library with the specified ***iMode***. The SylixOS loader detects whether ***pcFile*** is a path, and if so, it will load the file corresponding to the path. Otherwise, it will search for a file named ***pcFile*** in the search path of the dynamic library file, and the search path of the dynamic library file is shown in Section 19.2.2.

SylixOS opens the dynamic library in modes including: `RTLD_GLOBAL` and `RTLD_LOCAL`, where `RTLD_GLOBAL` indicates that the module is a global module. Note that only the global module can export symbols to the kernel symbol table; `RTLD_LOCAL` indicates a local module.

2. Search of Symbols

```
#include <dlfcn.h>
void *dlsym(void *pvHandle, const char *pcName);
```

Prototype analysis of the function `dlsym`:

- This function returns a symbol address or NULL if it succeeds, and returns NULL and sets an error number if it fails;
- The parameter ***pvHandle*** is a module handle returned by the function `dlopen`;
- The parameter ***pcName*** is the symbol name being searched.

Calling the function `dlsym` will return the function address represented by ***pcName*** from ***pvHandle*** dynamic library file. If ***pcName*** does not exist, it will return NULL, so it is not correct to judge whether the function `dlsym` is successful according to the return value, and we can get error message by calling the function `dlerror`.

3. Unloading of Dynamic Library

```
#include <dlfcn.h>
int dlclose(void *pvHandle);
```

Prototype analysis of the function `dlclose`:

- This function returns 0 if it succeeds, and returns -1 and sets the error number if it fails;
- The parameter ***pvHandle*** is a module handle returned by the function `dlopen`;

Calling the function `dlclose` will reduce the reference count of the ***pvHandle*** dynamic library. If the reference count is reduced to zero and no symbol is referenced, the dynamic library will be unloaded.

4. Getting Error Message

```
#include <dlfcn.h>
char *dlerror(void);
```

Prototype analysis of the function dlerror:

- Returns a string with an error message.

Calling the function dlerror will return an error message on calling the function dlopen, the function dlsym, and the function dlclose. If there is no error, NULL will be returned.

The following program shows how to load a dynamic library.

Program List 19.1 App program source code

```
#include <stdio.h>
#include <dlfcn.h>

int main (int argc, char *argv[])
{
    void      *so_handler;
    void      (*sub_fun) ();

    fprintf(stdout, "Hello World!\n");
    so_handler = dlopen("libsubfun.so", RTLD_GLOBAL);
    if (!so_handler) {
        fprintf(stderr, "%s \n", dlerror());
        return (-1);
    }

    sub_fun = dlsym(so_handler, "lib_func_test");
    if (!sub_fun) {
        fprintf(stderr, "%s \n", dlerror());
        return (-2);
    }

    sub_fun();
    dlclose(so_handler);
    return (0);
}
```

Program List 19.2 Dynamic link library source code

```
#include <stdio.h>

void lib_func_test (void)
{
    fprintf(stdout, "hello library lib_func_test() run!\n");
}
```

Run the program under the SylixOS Shell and the results are as follows:


```
# ./app
Hello World!
hello library lib_func_test() run!
```

19.3.2 Other APIs

The following function can get the symbol information that is smaller than the specified address and is closest to the specified address.

```
#include <dlfcn.h>
int dladdr(void *pvAddr, Dl_info *pdlinfo);
```

Prototype analysis of the function dladdr:

- This function returns the value greater than 0 if it succeeds, and returns 0 and sets the error number if it fails;
- The parameter **pvAddr** is a symbol address;
- The output parameter **pdlinfo** returns symbol information, the following is its member information.

Calling the dladdr function returns information for the **pvAddr** address, which is returned by the DL_info structure type, which is as follows:

```
typedef struct {
const char    *dli_fname;
void         *dli_fbase;
const char    *dli_sname;
void         *dli_saddr;
} Dl_info;
```

- Dli_fname: indicates the module file path;
- Dli_fbase: indicates the load address of the module;
- Dli_sname: indicates the symbol name;
- Dli_saddr: indicates the symbol address.

The dladdr function is generally used to print program stack information when an error is located.

The following program shows how to use the dladdr function to perform stack traceback.

Program List 19.3 Stack Traceback Example

```
#include <dlfcn.h>
#include <execinfo.h>
#include <stdio.h>
```

```
#include <string.h>

#define BT_SIZE 100
void print_backtrace()
{
    Dl_info    info;
    int        nptrs;
    int        i;
    int        ret;
    void        *ptr_buffer[BT_SIZE];

    nptrs = backtrace(ptr_buffer, BT_SIZE);    /* Get back stack
information */
    for (i = 1; i < nptrs; i++) {            /* Remove this function
frame print */
        ret = dladdr(ptr_buffer[i], &info);    /* Get the frame symbol
information */
        if (ret == 0) {
            break;
        }

        fprintf(stdout, "module:%s, function:%s, address:%p\n",
            info.dli_fname, info.dli_sname, info.dli_saddr);

        if (strcmp(info.dli_sname, "main") == 0) {
            break;
        }
    }
}

void func_test2 (void)
{
    print_backtrace();    /* Print stack information
*/
}

void func_test1 (void)
{
    func_test2();
}

int main (int argc, char *argv[])
{
    func_test1();
}
```

```

    return (0);
}

```

Run the program under the SylixOS Shell and the results are as follows:

```

# ./dladdr_test
module:/apps/dladdr_test/dladdr_test, function:func_test2, address:0xc00084a4
module:/apps/dladdr_test/dladdr_test, function:func_test1, address:0xc00084b4
module:/apps/dladdr_test/dladdr_test, function:main, address:0xc00084c4

```

19.4 Dynamic Link Library Shell Command

19.4.1 Viewing Dynamic Link Libraries

Running the **modules** command can view all module information that the SylixOS system has loaded, including kernel modules, executable programs, and dynamic libraries.

[Command Format]

```
modules
```

[Common Options]

```
None
```

[Instructions for parameters]

```
None
```

The following is the output of the **modules** command.

```

# modules
      NAME                               HANDLE   TYPE  GLB  BASE      SIZE  SYMCNT
-----
VPROCESS: kernel                        pid:   0 TOTAL MEMORY: 32768
+ xsiipc.ko                             30c5dfa8 KERNEL YES  c00e9000  633c   14
VPROCESS: app                            pid:   3 TOTAL MEMORY: 65536 <vp ver:1.3.4>
+ app                                    30c639c8 USER  YES  c0008000  83d8   2
+ libvmpdm.so                            30c63f20 USER  YES  c0018000  d39c   70
+ libsubfun.so                            30c63e58 USER  YES  c0028000  8344   2

total modules : 4

```

If only to view the kernel module information loaded by the SylixOS system, we can use the **lsmmod** command.

[Command Format]

lsmod

[Common Options]

None

[Instructions for parameters]

None

The following is the output of the **lsmod** command.

```
# lsmod

```

NAME	HANDLE	TYPE	GLB	BASE	SIZE	SYMCNT

VPROCESS: kernel	pid: 0	TOTAL MEMORY: 32768				
+ xsiipc.ko	30c5dfa8	KERNEL	YES	c00e9000	633c	14
total modules : 1						

19.4.2 Loading Kernel Modules

Through the use of the **modulereg** command, the kernel modules can be loaded.

[Command Format]

```
modulereg [kernel module file *.ko]
```

[Common Options]

None

[Instructions for parameters]

```
kernel module file *.ko: kernel module
```

The following is an example of using the **modulereg** command to register the xinput.ko module.

```
# modulereg /lib/modules/xinput.ko
module /lib/modules/xinput.ko register ok, handle : 0x30c64ae8
```

19.4.3 Unloading Kernel Modules

Using the **moduleunreg** command can unload the kernel module. Note that the parameter of the **moduleunreg** command is a module handle, so you need to use the **modules** or **lsmod** command to get the module handle before unloading.

[Command Format]

```
moduleunreg [kernel module handle]
```

[Common Options]

None

[Instructions for parameters]

kernel module handle: Kernel module handler

The following is the process of unloading a module.

```
# lsmod

```

NAME	HANDLE	TYPE	GLB	BASE	SIZE	SYMCNT

VPROCESS: kernel	pid: 0	TOTAL MEMORY: 49152				
+ xsiipc.ko	30c5dfa8	KERNEL	YES	c00e9000	633c	14
+ xinput.ko	30c64ae8	KERNEL	YES	c210e000	21d0	1

```
total modules : 2
# moduleunreg 30c64ae8
module /lib/modules/xinput.ko unregister ok.
```



Chapter 20 Power Management

20.1 SylixOS Power Management

The SylixOS power management is divided into two major parts: CPU power-consumption management and peripheral power-consumption management.

The CPU power management includes three modes:

- **Normal running mode:** CPU executes commands normally;
- **PowerSaving mode:** All the devices with power management function enter the power-saving mode, while the CPU dominant frequency is reduced, and the multi-core CPU only keeps one CPU running;
- **System suspend mode:** System suspend makes all devices with power management function enter the Suspend state. If the system needs to wake up through a specified event, it will recover from the reset vector. In this case, the bootloader or BIOS program is required to cooperate.

In SMP multi-core, the number of running CPU cores can be dynamically adjusted.

Peripheral power management includes into four states:

- **Normal operating state:** The device is turned on and enables the power and clock of the corresponding device to begin work;
- **Devices Shutdown state:** The device driver is turned off, to request the power management adapter to disconnect the device from power and the clock stops working.
- **Power-saving state:** The system enters power-saving mode to request peripherals to enter the power-saving mode;
- **Device idle state:** The device power-consumption management unit has a watchdog function. Once the idle time exceeds the set value, the system will make the device idle.

Figure 20.1 shows the basic structure diagram of power management in SylixOS.

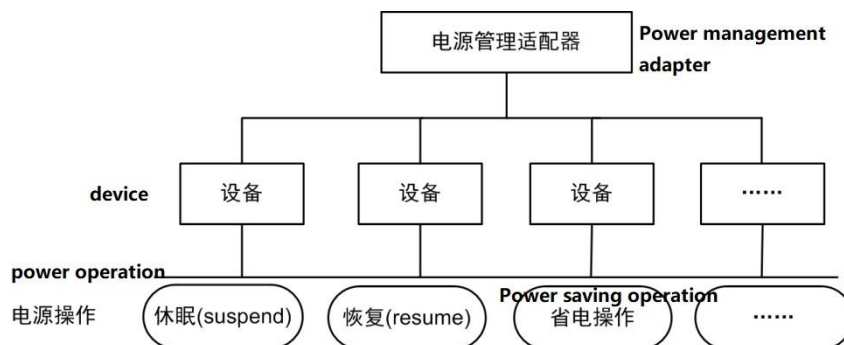


Figure 20.1 Power Management Structure Diagram

20.2 Power Management API

1. System Suspend^①

The following function controls all peripherals that support the suspend function to enter the suspend state.

```
#include <SylixOS.h>
VOID Lw_PowerM_Suspend(VOID);
```

2. System Wake-up

The following function controls all peripherals that support the suspend function to resume to the normal state from the suspend state.

```
#include <SylixOS.h>
VOID Lw_PowerM_Resume(VOID);
```

3. Set CPU Energy-Saving Parameters

The following function sets the number of CPU cores to run in a multi-core system and the power consumption level of them. The system turns off or turns on the CPU core according to the parameters. At the same time, it sets the CPU power consumption level. The power consumption levels are different. The CPU runs at different dominant frequencies and reduces the dominant frequency while entering the power saving mode, otherwise raises the dominant frequency. This function also notifies all peripherals that support power management of the change in CPU parameters.

```
#include <SylixOS.h>
VOID Lw_PowerM_CpuSet(ULONG ulNCpus, UINT uiPowerLevel);
```

Prototype analysis of the function `Lw_PowerM_CpuSet`:

- The parameter *ulNCpus* is the number of CPU cores in the running state;
- The parameter *uiPowerLevel* is the CPU power-consumption level.

4. Obtain CPU Energy-Saving Parameters

The following function obtains the number of the currently-running CPUs and the CPU power consumption level.

```
#include <SylixOS.h>
VOID Lw_PowerM_CpuGet(ULONG *pulNCpus, UINT *puiPowerLevel);
```

Prototype analysis of the function `Lw_PowerM_CpuGet`:

- The output parameter ***pulNCpus*** returns the number of running CPU cores.
- The output parameter ***puiPowerLevel*** returns the CPU power-consumption level.

Note that if ***pulNCpus*** and ***puiPowerLevel*** are NULL, this function does nothing.

5. System Enters the Power-Saving Mode

Through calling of the following function, we can make the system enter the power saving mode. It controls all the devices that support power management to enter the power-saving mode, and sets the number of CPU cores to run and the the power-consumption level.

```
#include <SylixOS.h>
VOID Lw_PowerM_SavingEnter(ULONG ulNCpus, UINT uiPowerLevel);
```

Prototype analysis of the function `Lw_PowerM_SavingEnter`:

- The parameter ***ulNCpus*** is the number of CPU cores in the running state;
- The parameter ***uiPowerLevel*** is the CPU power-consumption level.

6. System Exits the Power-Saving Mode

The following function controls the system to exit the power saving mode. It controls all devices that support power management to exit the power-saving mode, and sets the number of running CPU cores and power consumption levels.

```
#include <SylixOS.h>
VOID Lw_PowerM_SavingExit(ULONG ulNCpus, UINT uiPowerLevel);
```

Prototype analysis of the function `Lw_PowerM_SavingExit`:

- The parameter ***ulNCpus*** is the number of CPU cores in the running state;
- The parameter ***uiPowerLevel*** is the CPU power-consumption level.



Chapter 21 Introduction to Standard Third-Party Software

The open source feature of SylixOS facilitates the porting of open source software to SylixOS, which allows it to build high-end embedded systems using feature-rich open source software. This chapter mainly introduces the process of porting and using multiple types of open source software in SylixOS. According to the features of different types of open source software, different methods are used.

Note: If you have installed ACOINFO's Integrated Development Kit (RealEvo Kit), the Qt Graphical Interface Library will be automatically included without any configuration.

21.1 Qt Graphical Interface Software

Qt is a type of C++ graphical interface software widely-used at present. Qt itself has some features that are also very suitable for embedded applications, for example, it has rich API, contains hundreds of C++ class libraries, templates, etc., and can provide suitable functions for a wide variety of embedded applications. Qt adopts object-oriented design concept, with high degree of modularity, good reusability, and easy tailoring and customization.

In addition to the advantages of Qt design itself, Qt provides developers with a development tool QtCreator and a wealth of development documentation. Using QtCreator can not only develop graphical interface programs, but also develop non-graphical interface programs. QtCreator can complete almost all program development work and has improved the efficiency of program development to a certain extent.

As a cross-platform graphical interface library, Qt supports a wide range of operating systems, such as Linux, QNX, VxWorks, etc. Qt also supports the SylixOS operating system.

21.1.1 Qt Porting in SylixOS

The porting of Qt needs to be done under Linux. Therefore, it is necessary to establish the compiling environment of SylixOS under Linux first. It can be done through reference of "Guides for Linux Environment Development" of SylixOS.

Download the Qt source package first.

```
<linux> $① cd ~/sylixos_workspace/  
<linux> $ git clone https://github.com/SylixOS/qt.git
```

① This expression represents a shell prompt under Linux.

```
<linux>$ cd qt
```

Modify the configuration file of `mkspecs/qws/sylixos-arm-g++/qmake.conf` and set `SYLIXOS_BASE_PATH`:

```
<linux>$ vim mkspecs/qws/sylixos-arm-g++/qmake.conf
SYLIXOS_BASE_PATH = /home/user/sylixos_workspace/sylixos-base
```

Note: The value of `SYLIXOS_BASE_PATH` is the absolute path of the SylixOS Base project.

Configure Qt and the configuration command is as follows:

```
<linux>$ ./configure \
-prefix /opt/arm-sylixos-qt-4.8.7 \
-importdir "/opt/arm-sylixos-qt-4.8.7/qml" \
-release \
-opensource \
-confirm-license \
-embedded arm \
-xplatform qws/sylixos-arm-g++ \
-depths 8,16,24,32 \
-little-endian -host-little-endian \
-shared \
-fast \
-largefile \
-exceptions \
-stl \
-qt-sql-sqlite \
-qt3support \
-xmlpatterns \
-multimedia \
-audio-backend \
-svg \
-webkit \
-javascript-jit \
-script \
-scripttools \
-declarative \
-qt-zlib \
-qt-libtiff \
-qt-libpng \
-qt-libmng \
-qt-libjpeg \
-qt-freetype \
-verbose \
-optimized-qmake \
-no-pch \
```

```

-qt-kbd-sylixosinput \
-qt-gfx-sylixosfb \
-qt-mouse-sylixosinput \
-qt-gfx-transformed \
-qt-gfx-vnc \
-no-gfx-linuxfb \
-no-mouse-pc \
-no-mouse-linuxtp \
-no-kbd-tty \
-make examples -make tools -make docs -make demos

```

Compile Qt:

```
<linux>$ make
```

Install Qt under the root user, enter su to switch to the root user:

```
<linux>$ su
```

Modify the environment variable PATH, otherwise it will be very large after make install because arm-none-eabi-strip cannot be found:

```
<linux>$ export PATH=/usr/lib/gcc-arm-none-eabi/bin:$PATH
```

Enter the command to install Qt and Qt will be installed in the directory /opt/arm-sylixos-qt-4.8.7 specified during configuration:

```
<linux>$ make install
```

21.1.2 Verification of Qt Graphical Interface Library

1. Setting of Environment Variables

Enter the following environment variables under SylixOS Shell. Note that it is necessary to execute one by one to avoid errors due to the limited buffer size of the serial port driver.

```

DISPLAY=/dev/fb0
KEYBOARD=/dev/input/kbd0
MOUSE=/dev/input/touch0:/dev/input/mse0
XINPUT_PRIO=199
QTDIR=/qt
QPEDIR=/qt
QWS_DISPLAY=sylixosfb:$DISPLAY
QWS_MOUSE_PROTO=sylixosinput
QWS_KEYBOARD=sylixosinput
POINTERCAL_FILE=/etc/pointercal
QT_PLUGIN_PATH=$QTDIR/plugins
QT_QWS_FONTDIR=$QTDIR/lib/fonts

```

```
QML_IMPORT_PATH=$QTDIR/qml
QML2_IMPORT_PATH=$QTDIR/qml
LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

The following environment variables will be configured with function of using VNC:

```
QWS_DISPLAY=VNC:sylixosfb:$DISPLAY
```

Note: The VNC client-end under Windows can use vnc-4_1_2-x86_win32_viewer.exe.

The following environment variables will be configured with the 90° enable rotary screen:

```
QWS_DISPLAY=Transformed:Rot90
```

The following environment variables will be configured with the 90° enable rotary screen while using VNC;

```
QWS_DISPLAY=Transformed:Rot90:VNC:sylixosfb:$DISPLAY
```

Save the environment variable:

```
# varsave
```

Transfer the following files to the Sylix OS target board via the ftp tool^①:

- Qt plugin: From the /opt/arm-sylixos-qt-4.8.7/plugins directory;
- List of target board files in SylixOS: /qt/plugins;
- QML plugin: From the /opt/arm-sylixos-qt-4.8.7/qml directory;
- List of target board files in SylixOS: /qt/qml;
- Qt dynamic library: From the /opt/arm-sylixos-qt-4.8.7/lib directory.

2. Create Symbolic Links

After the download is complete, create a symbolic link under the SylixOS Shell as follows:

```
# ln -s /qt/lib/libQtCLucene.so.4.8.7 /qt/lib/libQtCLucene.so.4
# ln -s /qt/lib/libQtDesignerComponents.so.4.8.7
/qt/lib/libQtDesignerComponents.so.4
# ln -s /qt/lib/libQtHelp.so.4.8.7 /qt/lib/libQtHelp.so.4
# ln -s /qt/lib/libQtScriptTools.so.4.8.7 /qt/lib/libQtScriptTools.so.4
# ln -s /qt/lib/libQtTest.so.4.8.7 /qt/lib/libQtTest.so.4
# ln -s /qt/lib/libQtCore.so.4.8.7 /qt/lib/libQtCore.so.4
# ln -s /qt/lib/libQtDesigner.so.4.8.7 /qt/lib/libQtDesigner.so.4
# ln -s /qt/lib/libQtNetwork.so.4.8.7 /qt/lib/libQtNetwork.so.4
# ln -s /qt/lib/libQtSql.so.4.8.7 /qt/lib/libQtSql.so.4
# ln -s /qt/lib/libQtWebKit.so.4.9.4 /qt/lib/libQtWebKit.so.4
```

```
# ln -s /qt/lib/libQtDeclarative.so.4.8.7 /qt/lib/libQtDeclarative.so.4
# ln -s /qt/lib/libQtGui.so.4.8.7 /qt/lib/libQtGui.so.4
# ln -s /qt/lib/libQtScript.so.4.8.7 /qt/lib/libQtScript.so.4
# ln -s /qt/lib/libQtSvg.so.4.8.7 /qt/lib/libQtSvg.so.4
# ln -s /qt/lib/libQtXml.so.4.8.7 /qt/lib/libQtXml.so.4
# ln -s /qt/lib/libQtXmlPatterns.so.4.8.7 /qt/lib/libQtXmlPatterns.so.4
# ln -s /qt/lib/libQtMultimedia.so.4.8.7 /qt/lib/libQtMultimedia.so.4
# ln -s /qt/lib/libQt3Support.so.4.8.7 /qt/lib/libQt3Support.so.4
```

3. Screen Calibration

Set the size of the Shell stack:

```
# shstack 120000
```

Note: The Qt program stack has a large consumption, so the recommended setting is greater than 120K.

Register IPC module:

```
# modulereg /lib/modules/xsiipc.ko
```

Register input module:

```
# modulereg /lib/modules/xinput.ko
```

Perform a screen calibration procedure. According to the screen prompts, click on the "+" on the screen to complete the screen calibration.

```
# /apps/mousecalibration -qws &
```

21.1.3 QtCreator Installation and Configuration

The porting, compilation and development environment configuration of Qt takes a long time. To facilitate Qt programming, Beijing ACOINFO Co., Ltd. provided a compiled Qt library and optimized QtCreator (Windows version of RealEvo-QtSylixOS, the tool makes the above configuration process automated). Below is a brief introduction of the configuration process of RealEvo-QtSylixOS .

After turning on RealEvo-QtSylixOS, click on the menu "Tools → Options..." so that the Options dialog box pops up as shown in Figure 21.1.

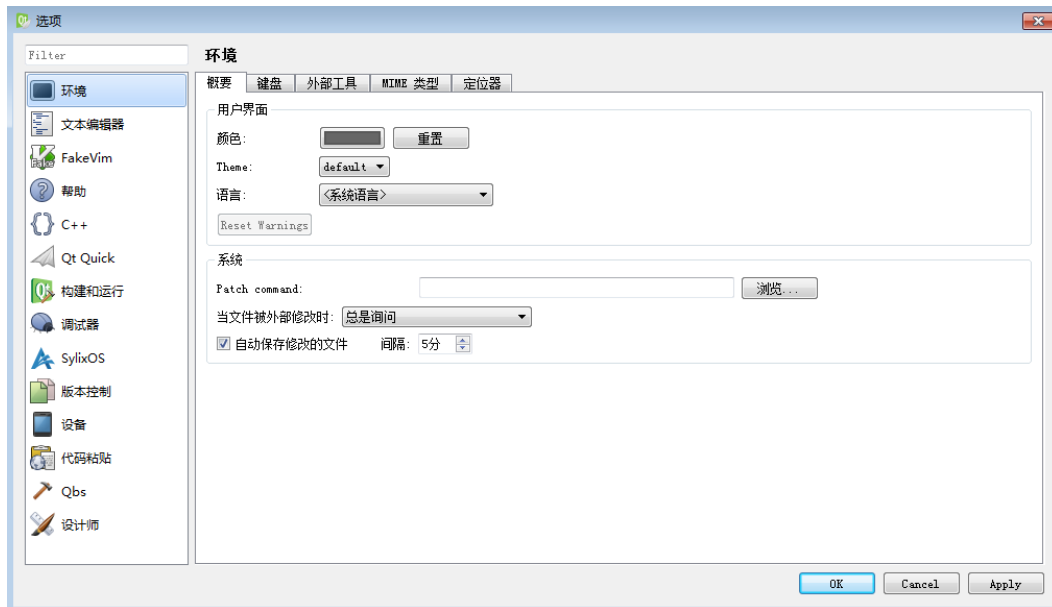


Figure 21.1 Options Dialog Box

Click "Device" in the left column to switch to the "Device" subpage, and click the "Add..." button and the device setup wizard selection dialog box will pop up as shown in Figure 21.2.

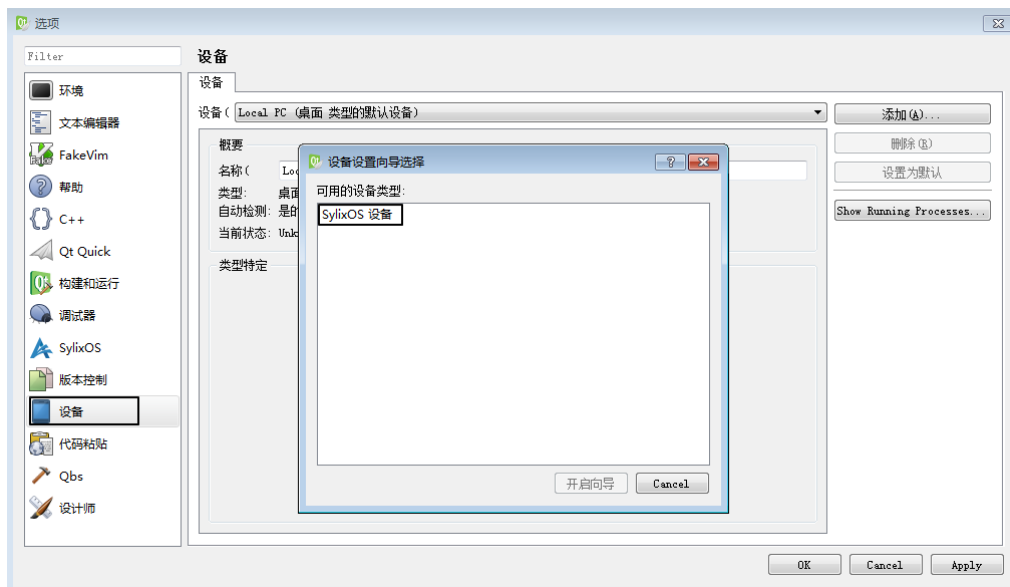


Figure 21.2 Device Settings Wizard Selection Dialog Box

Select "SylixOS Device" and then click "Start Wizard" button to start the SylixOS Device Configuration Wizard as shown in Figure 21.3.



Figure 21.3 SylixOS Device Configuration Wizard

Enter the name of the SylixOS device in the "Name" input box, such as "SylixOS Device"; enter the IP address of the SylixOS device in the "Host Name" input box, such as "192.168.7.30". In the "GDB port" input box, enter the port number, such as "1234" for monitoring during debugging of the Qt application of the device is started; enter the user name "root" in the "User name" input box; and enter the password "root" in the "Password" input box.

Click the "Next" button to enter the device's summary dialog box, and click the "Finish" button in the such dialog box to enter the device test's dialog box (ensure that the SylixOS device can be connected to the PC through the network). As shown in Figure 21.4, after the device test is completed successfully, click " the "Close" button for setting.

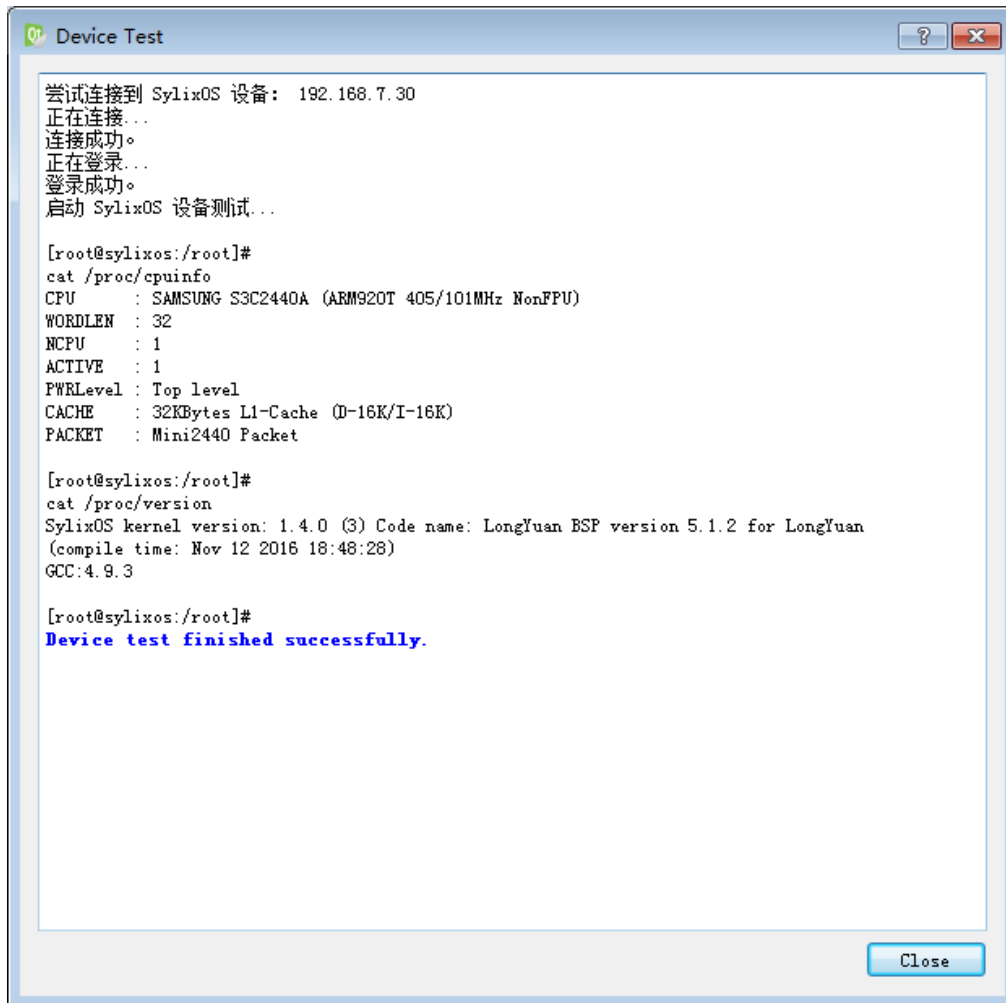


Figure 21.4 Device Test Dialog Box

After the above configuration is completed, we need to configure the SylixOS base project process. Return to the Options dialog box again, click "Build and Run" in the left column of the Options dialog box and switch to the "Kit" tab as shown in Figure 21.5.



Figure 21.5 Kit tab

Select "arm-sylixos-qt-4.8.7" (Kit) under "Auto Detect". The device shall be selected as "SylixOS XXX device (default device for SylixOS type)"; as for "SylixOS base project", click the "Browse" button on the right to select the compiled SylixOS base project category of "arm920t" for "CPU type", as shown in Figure 21.6.



Figure 21.6 Settings of SylixOS base project

After completing the basic tool configuration, we need to deploy the library for operation of Qt to the SylixOS device as follows:

Click on the menu "Tools → Options..." to bring up the Options dialog box. Click on "Device" in the left column to switch to the "Device" subpage, as shown in Figure 21.7.

Select "SylixOS XXX device (default device for SylixOS type)" in the device drop-down box, and then click the "Deploy Qt shared library ..." button, the "Deploy Qt to SylixOS device" dialog box will pop up as shown in Figure 21.8.

Select the Qt shared library most preferentially supported by the SylixOS device in the Qt shared library version drop-down box in Figure 21.8. If it is required to deploy the Debug version of the shared library and kernel module in the SylixOS Base project, we need to check the "Debug" check box. Otherwise, the "Release" version of shared libraries and kernel modules will be deployed.

Note: If the SylixOS device uses a file system (such as FAT32) that does not support symbolic links as the storage medium, we need to check the "Not support for symbolic links" check box.

Finally click on the "Deploy" button to start the deployment of the Qt shared library, as shown in Figure 21.7.

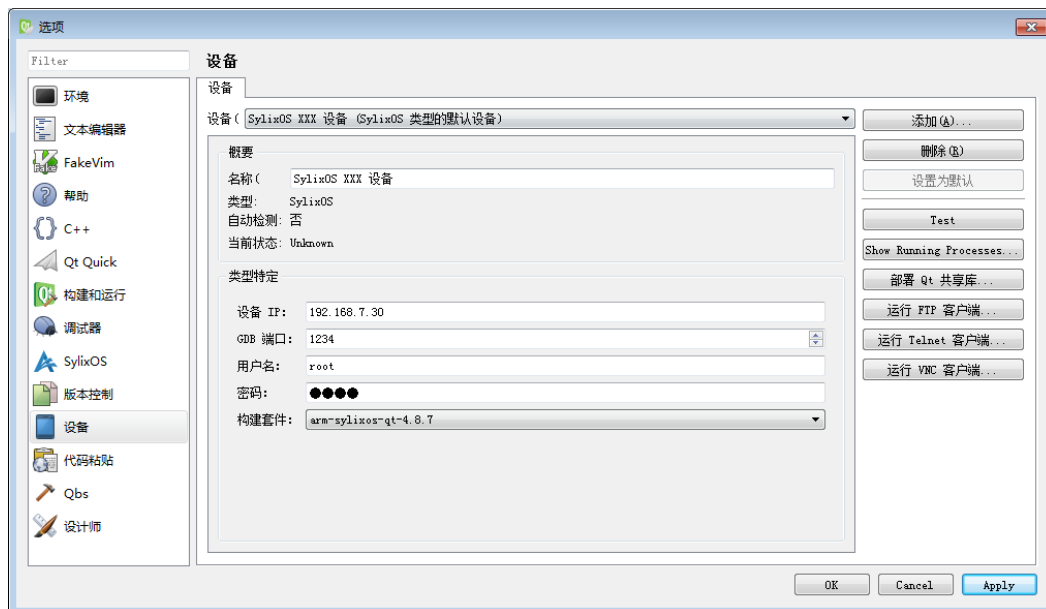


Figure 21.7 "Device" Selection Dialog Box

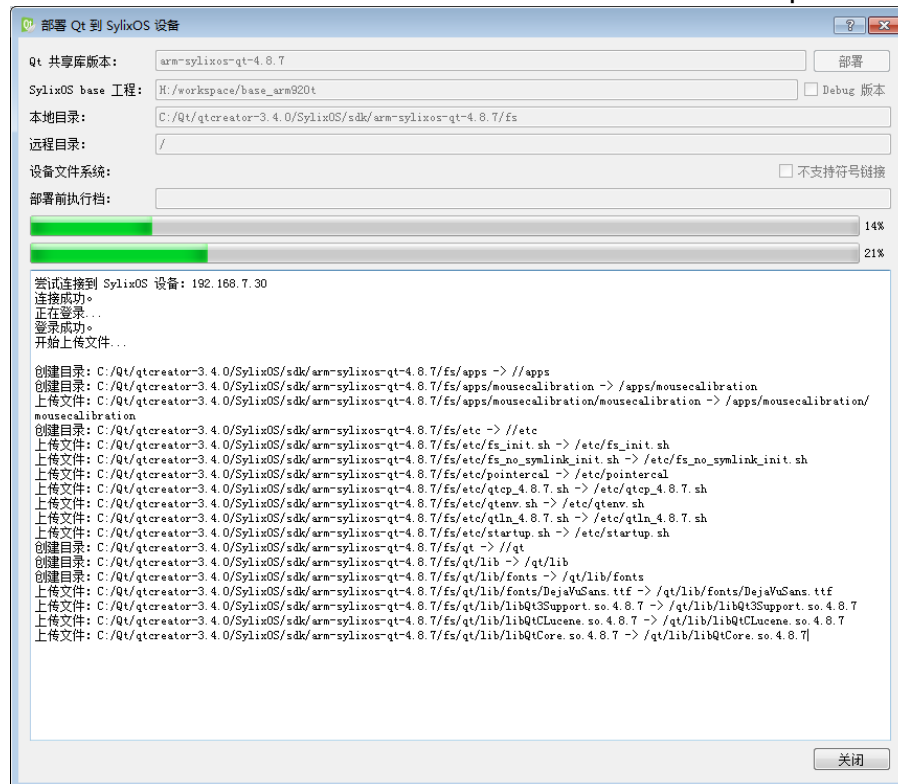


Figure 21.8 "Deploy Qt to SylixOS Device" dialog box

Through the above process, the configuration of RealEvo-QtSylixOS development environment and library files is completed. The development process of Qt program will not be introduced in this manual. Refer to the book "C++ GUI Qt 4 Programming" (2nd edition) for details.

21.2 Zlib File Compression Library

Zlib is a widely used file compression library at present. The projects like Linux, OpenSSL, libpng and ffmpeg all use zlib. In addition, thousands types of software also rely on the compression service provided by zlib.

21.2.1 Porting of Zlib Library

This section describes how to compile the zlib library under RealEvo-IDE. Download the zlib source code from the website <http://www.zlib.net/>, the file name is zlib-1.2.8.tar.xz, and use the decompression software to extract it to the current file folder. The file name of the extracted file is zlib-1.2.8.

Access the zlib-1.2.8 folder and open the /watcom/watcom_l.mak file. The contents of the file are as follows:

```
C_SOURCE =  adler32.c  compress.c  crc32.c    deflate.c    &
            gzclose.c  gzlib.c    gzread.c    gzwrite.c   &
            inffback.c inffast.c  inflate.c   inftrees.c  &
```

trees.c uncompr.c zutil.c

Where, *.c file is the program source file that implements zlib library compression. Copy the *.c files with the same name in the zlib-1.2.8 folder to a separate folder zlib, and copy the related *.h files to the zlib directory.

Open RealEvo-IDE, create "SylixOS Shared Lib" project, the project name is zlib_dll, copy the *.c and *.h files in the zlib folder to the directory of the project zlib_dll, and delete the zlib_dll.c file automatically generated when the project is created.

Modify the Makefile of the zlib_dll project and modify the SRCS as follows:

```

#*****
# src(s) file
#*****
SRCS =      \
    adler32.c \
    compress.c \
    crc32.c   \
    deflate.c \
    gzclose.c \
    gzlib.c   \
    gzread.c  \
    gzwrite.c \
    infback.c \
    inffast.c \
    inflate.c \
    inftrees.c \
    trees.c   \
    uncompr.c \
    zutil.c

```

SRCS is the same as C_SOURCE under the watcom_l.mak file. Modify the target output file name as follows:

```

#*****
# target
#*****
LIB = $(OUTPATH)/libzlib.a
DLL = $(OUTPATH)/libzlib.so

```

Note: The modification of the target part is not a must. The modification here is just to make the file name consistent with the usage habits.

After successful compilation, the two files defined in the target are generated in the Debug directory. At this point, the zlib library has been compiled and ported under SylixOS.

Next, we need to verify whether zlib can be used normally under SylixOS. Create a new "SylixOS App" project and modify the generated project file as follows:

Program List 21.1 Use of zlib Library

```
#include <stdio.h>
#include <zlib.h>

int main (int argc, char *argv[])
{
    unsigned char data_in[]    = "hello world! aaaaa bbbbbb ccccc
                                dddd 1234567890 notrecongen yes";

    unsigned char buf[1024]    = {0};
    unsigned char data_out[1024] = {0};
    unsigned long srcLen       = sizeof(data_in);
    unsigned long bufLen       = sizeof(buf);
    unsigned long dstLen       = sizeof(data_out);

    fprintf(stdout, "src string:%s\nlength:%ld\n", data_in, srcLen);

    compress(buf, &bufLen, data_in, srcLen);          /* compression
*/
    fprintf(stdout, "after compressed length:%ld\n", bufLen);

    uncompress(data_out, &dstLen, buf, bufLen);       /* unzip          */
    fprintf(stdout, "after uncompressed length:%ld\n", dstLen);

    fprintf(stdout, "uncompressed string:%s\n", data_out);
    return (0);
}
```

Modify the Makefile, add the included path of header file under the include path, and modify the following:

```
INCDIR = -I"${SYLIXOS_BASE_PATH}/libsylixos/SylixOS"
INCDIR += -I"${SYLIXOS_BASE_PATH}/libsylixos/SylixOS/include"
INCDIR += -I"${SYLIXOS_BASE_PATH}/libsylixos/SylixOS/include/inet"
INCDIR += -I"./zlib_dll"
```

Add zlib library file name and directory

```
*****
# depend dynamic library
*****
DEPEND_DLL = -lvmpdm
DEPEND_DLL += -lzlib
*****
```

```
# depend dynamic library search path
#*****
DEPEND_DLL_PATH = -L"${SYLIXOS_BASE_PATH}/libsylixos/${OUTDIR}"
DEPEND_DLL_PATH += -L"../zlib_dll/${OUTDIR}"
```

Note: When filing in the dynamic library name, the file generated in the zlib_dll project is libzlib.so file, and both the beginning and the end should be removed, -l should be added.

21.2.2 Verification of Zlib Library

Upload the libzlib.so file to the /lib directory of the SylixOS device and upload zlib_demo to the /apps/zlib_demo directory. The running results are as follows:

```
# ./zlib_demo
Src string:hello world! aaaaa bbbbb ccccc ddddd 1234567890 notrecongen yes
Length:64
After Compressed Length:62
After UnCompressed Length:64
UnCompressed String:hello world! aaaaa bbbbb ccccc ddddd 1234567890 notrecongen
yes
```

From the running results of the program, it can be seen that extracting function is achieved successfully through the zlib library.

21.3 SQLite3 Database

SQLite is a file type database, that means, a database is a file. Even if many tables, indexes, triggers, etc. are created in the database, it is physically just a file. This facilitates the backup of the database. The use of SQLite neither requires any database engine nor needs the installation of other database servers. So it is very suitable for the embedded devices. Actually, it has also been widely used in the embedded field, such as in iphone.

SQLite merges many files of earlier versions into one sqlite3.c file. This facilitates its porting but increases the size of the combined file.

21.3.1 Porting of SQLite3 in SylixOS

Currently, the SQLite3 library has been ported to SylixOS and can be compiled using RealEvo-IDE. The project file URL is <http://git.sylixos.com/cgit/cgit.cgi/SQLite3.git/> and the project is cloned into the local directory using the git management tool. After compilation, the files libsqlite3.so and sqlite3 are generated in the Debug directory.

Note: SQLite3 may be ported with reference to the porting method of zlib library.

21.3.2 Verification of SQLite3 Library

Upload the libsqlite3.so file to the /lib directory of the SylixOS device and upload the sqlite3 file to the /apps/sqlite3/ directory of the SylixOS device. The execution effects of the program sqlite3 are as follows:

```
# ls
```



```

sqlite3
# ./sqlite3
SQLite version 3.8.5 2014-04-28 17:56:19
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>

```

Create a database file using SQL command and write the data. SQL commands all end with ";".

```

# ls
sqlite3
# ./sqlite3 test.db /* Create a database
file */
SQLite version 3.8.5 2014-04-28 17:56:19
Enter ".help" for usage hints.
sqlite> create table mytable(name varchar(40), /* Create a table
*/
age smallint);
sqlite> insert into mytable values('sylixos tianhe',10);/* Insert a record into
the table */
sqlite> select *from mytable; /* View the contents of
the table */
sylixos tianhe|10 /* Table output from
the console */
sqlite> .quit
# ls /* View current
directory */
test.db sqlite3 /* One more database
file */

```

21.4 OpenSSL Encryption Library

21.4.1 Introduction to OpenSSL

OpenSSL is the open source implementation of SSL which is widely used, and is also a widely-used encryption function library as it implements various encryption algorithms used by SSL.

The SSL (Secure Socket Layer) security protocol was first proposed by Netscape and was originally used to secure HTTP communication between the Navigator browser and the web server. Later, the SSL protocol became the de facto standard for transmission-level secure communication and was adopted and improved by the IETF as a Transport Layer Security (TLS) protocol.

The SSL/TLS protocol is located between the TCP protocol and the application layer protocol and provides security services such as authentication, encryption, and integrity protection for both parties involved in the transmission. With SSL as the protocol framework, both parties in the communication can use the appropriate symmetric algorithms, public key algorithms, and MAC algorithms to enjoy security services.

OpenSSL consists of three parts: SSL protocol, cryptographic algorithm library, and application library. The SSL protocol part completely implements and encapsulates the three versions of the SSL protocol and the TLS protocol. The SSL protocol library is realized on the basis of a cryptographic algorithm library, and an SSL server and client can be established using the library. The cryptographic algorithm library is a powerful library, and it is the basis of OpenSSL, having implemented most of the current mainstream cryptographic algorithms and standards. The library mainly includes public key algorithm, symmetric encryption algorithm, hash function algorithm, X509 digital certificate standard, PKCS12 and PKCS7. We can also integrate external encryption algorithms into OpenSSL, such as using encryption cards.

The application part is the most lively part of OpenSSL, and is also the part of getting started with OpenSSL. A number of practical and exemplary applications are implemented based on the above cryptographic algorithm library and SSL protocol library, covering massive cryptographic applications. It mainly includes the encryption program for various cryptographic algorithms, the generation program of various keys, the SSL connection test program, and other standard applications.

The OpenSSL application includes the following two types:

- Applications based on OpenSSL directives, such as creating a CA certificate;
- Applications based on OpenSSL encryption library and protocol library.

The workload of an application based on OpenSSL library is much larger than that of an application based on OpenSSL command. However, the application of OpenSSL is based on the OpenSSL library. The application based on OpenSSL library can be flexibly selected according to the requirements without being restricted by the OpenSSL command.

21.4.2 Porting of OpenSSL Library

The porting of OpenSSL needs to be done under Linux. Before start of this work, it is necessary to establish the compiling environment of SylixOS under Linux first. It can be done with the reference to "Guides for Linux Environment Development" of SylixOS wiki .

Download the OpenSSL source file from the OpenSSL official website <http://www.openssl.org/>, and copy the contents of the file to the SylixOS workspace directory.

```
<linux>$ cd /home/user/sylixos_workspace /* Enter the sylixos workspace
*/
```

```

<linux>$ tar zxvf openssl-1.0.2a.tar.gz /* Unzip OpenSSL
*/
<linux>$ mv openssl-1.0.2a openssl /* Modify the OpenSSL directory name
*/
<linux>$ cd openssl /* Enter the OpenSSL directory
*/

```

Configure OpenSSL-version Open Source ARM Toolchain:

```

<linux>$ ./configure linux-armv4 -D__ARM_MAX_ARCH__=8
--cross-compile-prefix=arm-none-eabi- no-asm shared
--prefix=/opt/arm-sylixos-openssl

```

Add the following content to Makefile and modify the path of the SYLIXOS_BASE_PATH project according to the actual situation:

```

SYLIXOS_BASE_PATH=/home/user/sylixos_workspace/sylixos-base

INCDIR = $(SYLIXOS_BASE_PATH)/libsylixos/SylixOS
INCDIR += $(SYLIXOS_BASE_PATH)/libsylixos/SylixOS/include
INCDIR += $(SYLIXOS_BASE_PATH)/libsylixos/SylixOS/include/inet
SHLDIR = $(SYLIXOS_BASE_PATH)/libsylixos/Debug
SHLDIR += $(SYLIXOS_BASE_PATH)/libcextern/Debug

SYLIXOS_CFLAGS=-DSYLIXOS -DSYLIXOS_LIB $(addprefix -I,$(INCDIR)) -mcpu=arm920t
-fmessage-length=0 -fno-short-enums
SYLIXOS_EXLIBS=-nostdlib $(addprefix -L,$(SHLDIR)) -lcextern -lvmpm -lm -lgcc

```

Modify CFLAG:

```
CFLAG = $(SYLIXOS_CFLAGS) ...
```

Modify PEX_LIBS and EX_LIBS:

```

PEX_LIBS = -shared
EX_LIBS = $(SYLIXOS_EXLIBS)

```

Compile OpenSSL:

```
<linux>$ make
```

The installation of OpenSSL requires superuser authority, enter su to switch to the root user:

```
<linux># su
```

Modify the environment variable PATH:

```
<linux># export PATH=/usr/lib/gcc-arm-none-eabi/bin:$PATH
```

Enter the command to install OpenSSL to the specified directory /opt/arm-sylixos-openssl:

```
<linux># make install
```

Deploy OpenSSL to the openssl directory in the /lib directory of the SylixOS target board and upload the result of make install (located in the /opt/arm-sylixos-openssl directory) to the /lib/openssl directory of the SylixOS target board.

Run the following command to complete the relevant configuration of OpenSSL:

```
# cd /lib/openssl/lib /* Create a symbolic link
*/
# ln -s libssl.so.1.0.0 libssl.so
# ln -s libcrypto.so.1.0.0 libcrypto.so
# PATH=$PATH:/lib/openssl/bin /* Modify environment variables
*/
# LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/lib/openssl/lib
# varsave
```

21.4.3 Verification of OpenSSL Library

On the console of the Sylix OS device, enter openssl x (x is any character) and the console will output the command supported by the current OpenSSL.

```
# openssl x
openssl:Error: 'x' is an invalid command.

Standard commands
asn1parse      ca              ciphers         cms
crl             crl2pkcs7      dgst            dh
dhparam        dsa             dsaparam        ec
ecparam        enc             engine          errstr
gendh          gensa           genpkey         genrsa
nseq           ocsf            passwd          pkcs12
pkcs7          pkcs8          pkey            pkeyparam
pkeyutl        prime          rand            req
rsa            rsautl         s_client        s_server
s_time         sess_id        smime           speed
spkac          srp            ts              verify
version        x509

Message Digest commands (see the `dgst' command for more details)
md4             md5             mdc2            rmd160
sha             sha1

Cipher commands (see the `enc' command for more details)
aes-128-cbc     aes-128-ecb    aes-192-cbc     aes-192-ecb
aes-256-cbc     aes-256-ecb    base64          bf
```

	bf-cbc	bf-cfb	bf-ecb	bf-ofb
camellia-128-cbc	camellia-128-ecb	camellia-128-ecb	camellia-192-cbc	camellia-192-ecb
camellia-256-cbc	camellia-256-ecb	cast	cast-cbc	
cast5-cbc	cast5-cfb	cast5-ecb	cast5-ofb	
des	des-cbc	des-cfb	des-ecb	
des-ede	des-ede-cbc	des-ede-cfb	des-ede-ofb	
des-ede3	des-ede3-cbc	des-ede3-cfb	des-ede3-ofb	
des-ofb	des3	desx	idea	
idea-cbc	idea-cfb	idea-ecb	idea-ofb	
rc2	rc2-40-cbc	rc2-64-cbc	rc2-cbc	
rc2-cfb	rc2-ecb	rc2-ofb	rc4	
rc4-40	seed	seed-cbc	seed-cfb	
seed-ecb	seed-ofb			

Enter openssl version to output the current version number of OpenSSL:

```
# openssl version
OpenSSL 1.0.2a 19 Mar 2015
```

21.5 GoAhead Web Server

GoAhead is widely used in the embedded web servers, such as Siemens, Honeywell and Hewlett-Packard. Extensive use illustrates the power, security, and stability of GoAhead. The main reason why GoAhead is widely used is that it has the following advantages:

- Easy to be integrated with device: The web application can be ported onto a real-time operating system easily, and through the web application, we can access hardware features easily as well;
- Supporting storage of web pages in BOM: GoAhead supports compiling of web pages and linking them to the final executable file. Very meaningful for some embedded applications without a file system;
- Encryption and user management: GoAhead Server supports the use of SSL for data encryption and authentication. It also supports digest authentication (a more secure authentication mechanism using encrypted passwords). The user management feature allows different users to have different levels of access authorities;
- In addition to the above features, GoAhead also provides a variety of ways to write dynamic pages, which can quickly and easily generate dynamic pages.

21.5.1 Porting of GoAhead

Currently, the porting on SylixOS has been completed, and the GoAhead project can be compiled under RealEvo-IDE. The project file is located at <http://git.sylixos.com/cgit/cgit.cgi/GoAhead-WebServer.git/>, and the project is cloned to the current host and the compiling can be implemented by importing it into RealEvo-IDE. The compiled file is in the Debug directory of the project, and 4 files will be generated at the same time:

```
goahead:          /* The main program to achieve webserver
*/
gopass:           /* Account management for the web
*/
webcomp:          /* Used to generate rom pages
*/
libgoahead.so:    /* Goahead dynamic library, need to be copied to the /lib
directory of the SylixOS target system */
```

In the above four files, basic web functions can be realized by using the libgoahead.so dynamic library and the goahead program.

21.5.2 GoAhead Verification

Upload the libgoahead.so file to the /lib directory of the SylixOS target system. Upload the goahead program file to the /apps/goahead directory. We also need to upload the auth.txt, route.txt, and web folders under ./embedthis-goahead/src to the /apps/goahead directory of the SylixOS target system. Run the goahead program^① and use the **ifconfig** command to view the IP address of the SylixOS target system, such as the displayed IP address 192.168.1.13.

Start the browser on the PC host and enter <http://192.168.1.13/> in the address bar and then press Enter, the web page information will be displayed. The default information is "Congratulations! The server is up and running.", modify the index.html file under the ./web directory. Run goahead again, start the browser, the modified content will be displayed

21.6 C-Language Interpreter

An Interpreter is a program that can directly translate and run high-level programming languages line by line. The interpreter does not translate the entire program at a time. Rather, it acts like an "intermediary". The program must be converted into another language before it is run. Therefore, the interpreter program runs slowly. It runs a line of program immediately after the line is translated. And then, it will translate and run the next line, and so on.

Picoc is an open source code project that was originally intended to realize the execution and interpretation of C-language on an embedded device. Its core code is only about 4,000 lines, readable and it is able to execute the basic C runtime library, very suitable for use as a script in automation areas such as robots and drones.

21.6.1 Picoc Porting

The Tiny-c-interpreter project is the name of the Picoc interpreter on github, which has now been ported to SylixOS and can be compiled using RealEvo-IDE. The compilation and execution of the Picoc interpreter relies on the readline library, so the readline project should be compiled before compiling of the Picoc interpreter.

Its project file is available at <https://github.com/jiaojinxing/tiny-c-interpreter>. After the project file is cloned into a local directory, open the folder and you will find there are no source files. Here, the TortoiseGit tool should be installed. And then, right click /TortoiseGit/SubMoudle Add in the blank space to add the assembly. You also need to clone the project readline in the same workspace. Compile the readline project first and then compile the tiny-c-interpreter project.

21.6.2 Picoc Usage Verification

Upload the compiled project file to the target system of SylixOS. The files that need to be uploaded include libreadline.so and picoc files. libreadline.so is uploaded to the /lib directory of the target system of SylixOS, and picoc is uploaded to the /apps/picoc directory. In the project directory tiny-c-interpreter/test is picoc test project routines, copy 00_assignment.c file to /apps/picoc directory. The execution effect of picoc is as follows:

```
# ./picoc 00_assignment.c
42
64
12, 34
```

Chapter 22 Platform Porting

22.1 From Linux to SylixOS

Linux is a UNIX-like operating system that is compatible with POSIX standards and supports multi-user, multi-thread, and multi-CPU. The basic idea of Linux is that everything is a file. SylixOS follows this philosophy, and all that in SylixOS (including hardware, software devices, etc.) can be seen as files. As a POSIX-compliant operating system, SylixOS can easily run the applications under the POSIX standard (such as the third-party software described in Chapter 21). Therefore, the same POSIX-compliant application under Linux can be run perfectly under SylixOS without any modification, as shown in Program List 22.1.

Program List 22.1 Thread Creation

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

void *thread1 (void *arg)
{
    int i;

    for (i = 0; i < 10; i++) {
        fprintf(stdout, "thread1 running...\n");
        sleep(1);
    }

    return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t    tid;
    int          ret;

    ret = pthread_create(&tid, NULL, thread1, NULL);
    if (ret < 0) {
        perror("pthread_create");
        return (-1);
    }

    pthread_join(tid, NULL);
}
```



```
return (0);  
}
```

Note: The method to port the Linux application SylixOS is described in Section 21.2 zlib File Compression Library.

22.2 From VxWorks to SylixOS

VxWorks is also a POSIX-compliant real-time operating system (this is exactly the same as SylixOS), so there is not much difference between the two operating systems for POSIX application, and the two are basically fully compatible (SylixOS is more compatible with the POSIX standard than VxWorks). Besides, SylixOS provides a VxWorks compatible interface to enable VxWorks developers to adapt to SylixOS's program development faster. The following is an example of the API in the SylixOS logging system.

```
#include <SylixOS.h>  
INT logFdSet(INT iWidth, fd_set *pfdssetLog);  
INT logFdGet(INT *piWidth, fd_set *pfdssetLog);
```

22.2.1 Development of VxWorks Applications in RealEvo-IDE

RealEvo-IDE provides a compatible library for the development of VxWorks program, which makes SylixOS fully compatible with VxWorks in terms of code (Figure 22.1 is a compatible library file supported by SylixOS for VxWorks). The following describes how to use the RealEvo-IDE to develop a VxWorks program:

1. Creation of SylixOS Base Project

Open RealEvo-IDE and create a new SylixOS base project (Refer to Section 4.1.1 Creating SylixOS Base project for the creation process). Note that, the default library selection dialog box should be selected as the libVxWorks-compliant library (the VxWorks application depends on this library) and finally click "Finish" to finish the creation of the project, as shown in Figure 22.1.

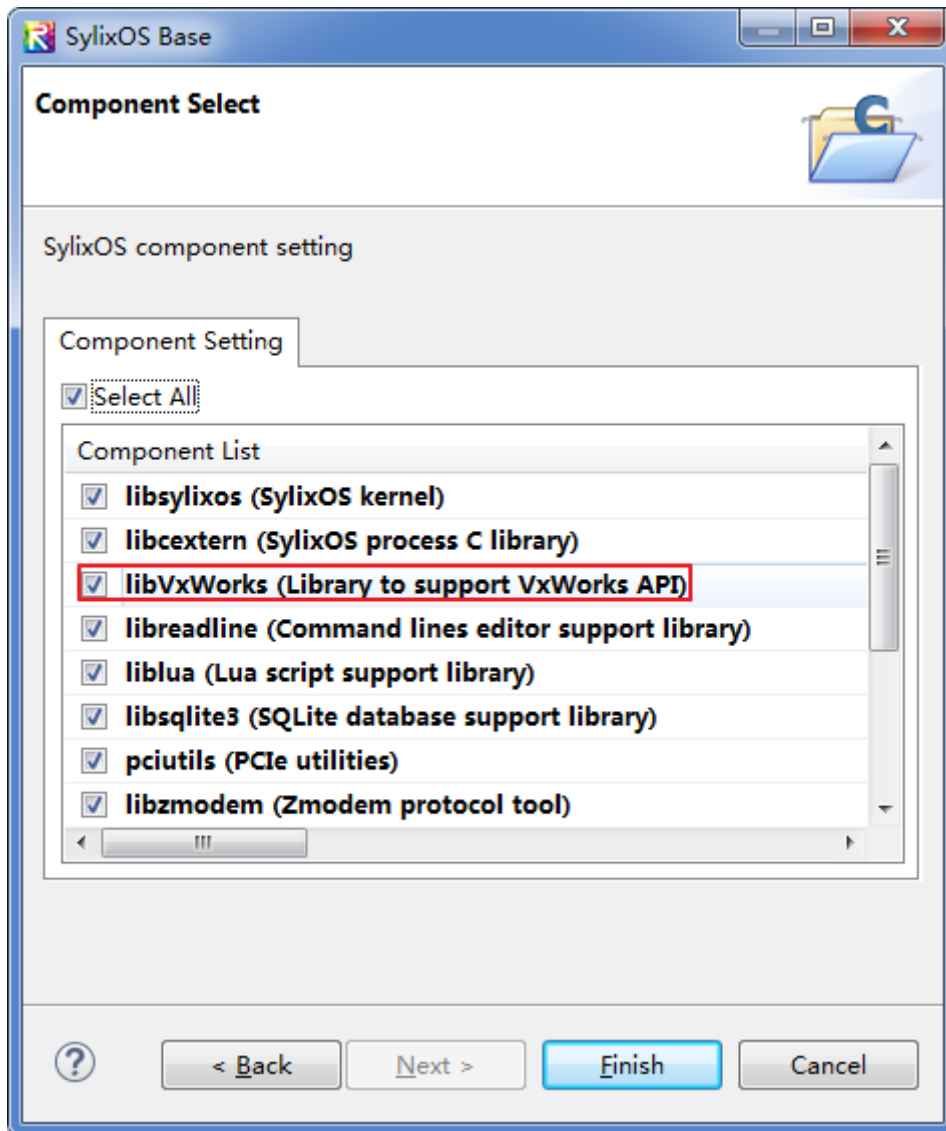


Figure 22.1 Selection of Default Library

The created SylixOS base project directory contains: libcextern (standard C library), libsylixos (SylixOS kernel library), libVxWorks (VxWorks-compliant library), etc., as shown in Figure 22.2.

Compile the SylixOS base project. After successful programming, the libVxWorks.so file will be generated under libVxWorks/Debug. This file is the dynamic library that needs to be linked when the VxWorks application runs under SylixOS.

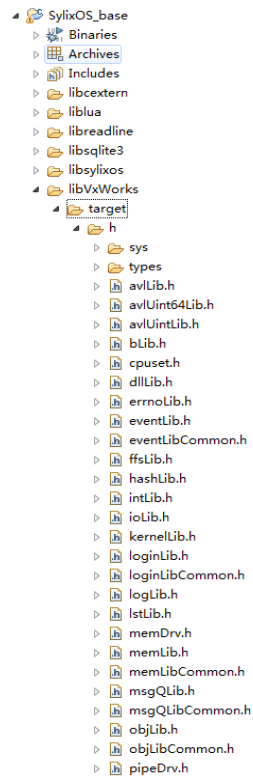


Figure 22.2 libVxWorks Directory Structure

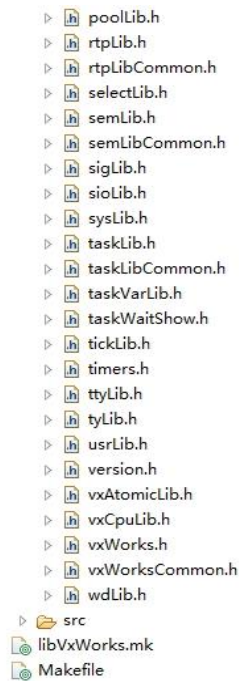


Figure 22.3 libVxWorks Directory Structure (continued)

2. Creation of VxWorks Application

Create a VxWorks application project (see Section 4.1.2). After the creation is complete, modify the project's target Makefile (*.mk) file, as shown in Figure 22.4.

```

*****
# Header file search path (eg. LOCAL_INC_PATH := -I"Your header files search path")
*****
LOCAL_INC_PATH := \
-I"${SVLIXOS_BASE_PATH}/libUxWorks/target/h"

*****
# Pre-defined macro (eg. -DYOUR_MACRO=1)
*****
LOCAL_DSYMBOL :=

*****
# Compiler flags
*****
LOCAL_CFLAGS :=
LOCAL_CXXFLAGS := |

*****
# Depend library (eg. LOCAL_DEPEND_LIB := -la LOCAL_DEPEND_LIB_PATH := -L"Your library search path")
*****
LOCAL_DEPEND_LIB := \
-lUxWorks

LOCAL_DEPEND_LIB_PATH := \
-L"${SVLIXOS_BASE_PATH}/libUxWorks/${OUTDIR}"

```

Figure 22.4 Modification of Makefile

- LOCAL_INC_PATH: including the header file directory, with the libVxWorks header file path added;
- LOCAL_DEPEND_LIB: including the dependent dynamic library, with the VxWorks dynamic library added;
- LOCAL_DEPEND_LIB_PATH: including the dependent dynamic library path, with the VxWorks dynamic library path added.

The following program shows the use of binary semaphores in VxWorks. This program creates two tasks by calling the taskSpawn function. Semaphore 1 (semId1) is taken in task A, and then Semaphore 2 (semId2) given. Task B starts running and gives Semaphore 1 (semId1), thus achieving the synchronization of Task A and Task B.

Program List 22.2 Use of VxWorks semaphores

```

#include <vxWorks.h>
#include <taskLib.h>
#include <semLib.h>
#include <stdio.h>

#define TASK_PRI          (98)          /* Task priority
    */

#define TASK_STACK_SIZE  (5000)       /* Task stack size
    */

LOCAL SEM_ID    semId1;               /* Binary semaphore 1
    */

LOCAL SEM_ID    semId2;               /* Binary semaphore 2
    */

```

```
LOCAL BOOL      flag;
LOCAL int       num = 4;

LOCAL STATUS taskA()
{
    int i;

    for (i = 0; i < num; i++) {
        if (semTake(semId1, WAIT_FOREVER) == ERROR) {
            perror("taskA: semTake");
            return (ERROR);
        }

        printf("[done-%d]taskA: Releasing semId2 [taskB process]\n", i + 1);

        if (semGive(semId2) == ERROR) {
            perror("taskA: semGive");
            return (ERROR);
        }
    }

    return (OK);
}

LOCAL STATUS taskB()
{
    int i;

    for (i = 0; i < num; i++) {
        if (semTake(semId2, WAIT_FOREVER) == ERROR) {
            perror("taskB: semTake");
            return (ERROR);
        }

        printf("[done-%d]taskB: Releasing semId1 [taskA process]\n", i + 1);

        if (semGive(semId1) == ERROR) {
            perror("taskB: semGive");
            return (ERROR);
        }
    }

    flag = FALSE;

    return (OK);
}
```

```
}

STATUS synch ()
{
    flag = TRUE;

    if ((semId1 = semBCreate(SEM_Q_PRIORITY, SEM_FULL)) == (SEM_ID)NULL) {
        perror("synch: semBCreate");
        return (ERROR);
    }

    if ((semId2 = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY)) == (SEM_ID)NULL) {
        perror("synch: semBCreate");
        return (ERROR);
    }

    if (taskSpawn("tTaskA", TASK_PRI, 0, TASK_STACK_SIZE, (FUNCPTR)taskA, 0,
                 0, 0, 0, 0, 0, 0, 0, 0) == ERROR) {
        perror("synch: taskSpawn");
        return (ERROR);
    }

    if (taskSpawn("tTaskB", TASK_PRI, 0, TASK_STACK_SIZE, (FUNCPTR)taskB, 0,
                 0, 0, 0, 0, 0, 0, 0, 0) == ERROR) {
        perror("synch: taskSpawn");
        return (ERROR);
    }

    while (flag)
        taskDelay(sysClkRateGet());

    if (semDelete(semId1) == ERROR) {
        perror("synch: semDelete");
        return (ERROR);
    }

    if (semDelete(semId2) == ERROR) {
        perror("synch: semDelete");
        return (ERROR);
    }

    printf("synch now completed.\n");

    return (OK);
}
```

```
int main(int argc, char * argv[])
{
    synch ();

    return (OK);
}
```

Run the program under SylixOS Shell and the results are as follows:

```
# ./vxworks_test
VxWork compatibility library enter.
[done-1]taskA: Releasing semId2 [taskB process]
[done-1]taskB: Releasing semId1 [taskA process]
[done-2]taskA: Releasing semId2 [taskB process]
[done-2]taskB: Releasing semId1 [taskA process]
[done-3]taskA: Releasing semId2 [taskB process]
[done-3]taskB: Releasing semId1 [taskA process]
[done-4]taskA: Releasing semId2 [taskB process]
[done-4]taskB: Releasing semId1 [taskA process]
synch now completed.
VxWork compatibility library exit.
```

From the running results of the operation, it can be seen that the VxWorks application can run normally under SylixOS without any modification. And the running result is the same as the program expectation (synchronous running of task A and task B).

Appendix A Standard Header File

A.1 C Standard Header File

Table A.1 C Standard Header File

Header file name	Description
<assert.h>	Verifier assertion
<ctype.h>	Character type
<errno.h>	Error code
<inttypes.h>	Integer format conversion
<limits.h>	Implemented constant
<locale.h>	Local category
<setjmp.h>	Non-local goto
<signal.h>	Signal
<stdarg.h>	Variable parameter table
<stdint.h>	Integral
<stdio.h>	Standard I/O library
<stdlib.h>	Utility library functions
<string.h>	String manipulation
<time.h>	Time and date
<wchar.h>	Support for extended multi-byte and wide character
<wctype.h>	Support for wide character classification and mapping

A.2 POSIX Standard Header File

Table A.2 POSIX Standard Header File

Header file name	Description	Header file name	Description
< dirent.h>	Directory item	<dlfcn.h>	Dynamic link library operation function
<fcntl.h>	File control	<fmtmsg.h>	Message display structure
<fnmatch.h>	File name matching type	<ftw.h>	File tree roaming
<grp.h>	Group file	<iconv.h>	Code set conversion utility program
<netdb.h>	Network database operations	<langinfo.h>	Language information constant
<pwd.h>	Password file	<libgen.h>	Definition of pattern matching function
<regex.h>	Regular expression	<monetary.h>	Currency type

< tar.h>	TAR archived value	<ndbm.h>	Database operation
< termios.h>	Terminal I/O	<nl_types.h>	Message category
< unistd.h>	Symbolic constant	<poll.h>	Polling function
< arpa/inet.h>	Internet definition	<search.h>	Search list
< netinet/in.h>	Internet address family	<strings.h>	String manipulation
<netinet/tcp.h>	Definition of Transmission Control Protocol	<syslog.h>	System error logging
<sys/mman.h>	Memory Management Statement	<ulimit.h>	User restrictions
< sys/select.h>	select function	<utmpx.h>	User account database
< sys/socket.h>	Socket interface	<sys/ipc.h>	IPC mechanism
< sys/stat.h>	File status	<sys/msg.h>	Message queue
< sys/times.h>	Process time	<sys/resource.h>	Resource operation
< sys/ types.h>	Basic system data type	<sys/sem.h>	Semaphore
<sys/un.h>	Definition of UNIX domain socket	<sys/shm.h>	Shared memory
<sys/utsname.h>	System name	<sys/statvfs.h>	File system information
<sys/wait.h>	Process control	<sys/time.h>	Time type
<cpio.h>	Cpio archive value	<sys/timeb.h>	Definitions of additional date and time
<sys/uio.h>	Vector I/O operation	<aio.h>	Asynchronous I/O
<mqueue.h>	Message queue	<pthread.h>	Thread
<sched.h>	Perform scheduling	<semaphore.h>	Semaphore
<spawn.h>	Real-time spawn interface	<stropts.h>	XSI STREAMS Interface
<trace.h>	Time tracking	<sched_rms.h> ^①	RMS scheduling

① The POSIX standard does not define this section. This is an extension to POSIX by SylixOS.



Annex B SylixOS Error Number

B.1 POSIX Error Number

Table B.1 POSIX Error Number

Error Number	Description	Error Number	Description
EPERM	Not owner	EEXIST	File exists
ENOENT	No such file or directory	EXDEV	Cross-device link
ESRCH	No such process	ENODEV	No such device
EINTR	Interrupted system call	ENOTDIR	Not a directory
EIO	I/O error	EISDIR	Is a directory
ENXIO	No such device or address	EINVAL	Invalid argument or format
E2BIG	Arg list too long or over flow	ENFILE	File table overflow
ENOEXEC	Exec format error	EMFILE	Too many open files
EBADF	Bad file number	ENOTTY	Not a typewriter
ECHILD	No children	ENAMETOOLONG	File name too long
EAGAIN	No more processes or operation would block	EFBIG	File too large
ENOMEM	Not enough core	ENOSPC	No space left on device
EACCES	Permission denied or can not access	ESPIPE	Illegal seek
EFAULT	Bad address	EROFS	Read-only file system
ENOTEMPTY	Directory not empty	EMLINK	Too many links
EBUSY	Mount device busy	EPIPE	Broken pipe
EDEADLK	Resource deadlock avoided	ENOLCK	No locks available
ENOTSUP	Unsupported value	EMSGSIZE	Message size
EDOM	Argument too large	ERANGE	Result too large
ECANCELED	Operation canceled	EWRPROTECT	Write protect
EFORMAT	Invalid format	ENOSR	Insufficient memory
EBADMSG	Invalid STREAMS message	ENODATA	Missing expected message data
ETIME	STREAMS timeout occurred	ENOMSG	Unexpected message type

B.2 IPC/Web Error Number

Table B.2 Web Error Number

Error Number	Description	Error Number	Description
--------------	-------------	--------------	-------------

EDESTADDRREQ	Destination address required	ETOOMANYREFS	Too many references: can't splice
EPROTOTYPE	Protocol wrong type for socket	ETIMEDOUT	Connection timed out
ENOPROTOOPT	Protocol not available	ECONNREFUSED	Connection refused
EPROTONOSUPPORT	Protocol not supported	ENETDOWN	Network is down
ESOCKTNOSUPPORT	Socket type not supported	ETXTBSY	Text file busy
EOPNOTSUPP	Operation not supported on socket	ELOOP	Too many levels of symbolic links
EPFNOSUPPORT	Protocol family not supported	EHOSTUNREACH	Host unreachable
EAFNOSUPPORT	Addr family not supported	ENOTBLK	Block device required
EADDRINUSE	Address already in use	EHOSTDOWN	Host is down
EADDRNOTAVAIL	Can't assign requested address	EINPROGRESS	Operation now in progress
ENOTSOCK	Socket operation on non-socket	EALREADY	Operation already in progress
ENETUNREACH	Network unreachable	ENOSYS	Function not implemented
ENETRESET	Network dropped connection on reset	ESHUTDOWN	Can't send after socket shutdown
ECONNABORTED	Software caused connection abort	ENOTCONN	Socket is not connected
ECONNRESET	Connection reset by peer	EISCONN	Socket is already connected
ENOBUFS	No buffer space available		

B.3 SylixOS Kernel Error Number

Table B.3 Kernel Error Number

Error Number	Description
ERROR_NONE	No error (0)
PX_ERROR	Error (-1)
ERROR_KERNEL_PNAME_NULL	Invalid name
ERROR_KERNEL_PNAME_TOO_LONG	Name too long
ERROR_KERNEL_HANDLE_NULL	Invalid handle
ERROR_KERNEL_IN_ISR	Kernel in interrupt service mode
ERROR_KERNEL_RUNNING	Kernel is running
ERROR_KERNEL_NOT_RUNNING	Kernel is not running
ERROR_KERNEL_OBJECT_NULL	Invalid object
ERROR_KERNEL_LOW_MEMORY	Kernel not enough memory
ERROR_KERNEL_BUFFER_NULL	Invalid buffer
ERROR_KERNEL_OPTION	Unsupported option
ERROR_KERNEL_VECTOR_NULL	Invalid vector
ERROR_KERNEL_HOOK_NULL	Invalid hook
ERROR_KERNEL_OPT_NULL	Invalid option
ERROR_KERNEL_MEMORY	Invalid address
ERROR_KERNEL_LOCK	Kernel locked
ERROR_KERNEL_CPU_NULL	Invalid cpu
ERROR_KERNEL_HOOK_FULL	Hook table full
ERROR_KERNEL_KEY_CONFLICT	Key conflict
ERROR_DPMA_NULL	Invalid DPMA
ERROR_DPMA_FULL	DPMA full
ERROR_DPMA_OVERFLOW	DPMA overflow
ERROR_LOADER_FORMAT	Invalid format
ERROR_LOADER_ARCH	Invalid architectural
ERROR_LOADER_RELOCATE	Relocate error
ERROR_LOADER_EXPORT_SYM	Can not export symbol(s)
ERROR_LOADER_NO_MODULE	Can not find module
ERROR_LOADER_CREATE	Can not create module
ERROR_LOADER_NO_INIT	Can not find initial routien
ERROR_LOADER_NO_ENTRY	Can not find entry routien
ERROR_LOADER_PARAM_NULL	Invalid parameter(s)
ERROR_LOADER_UNEXPECTED	Unexpected error
ERROR_LOADER_NO_SYMBOL	Can not find symbol

ERROR_LOADER_VERSION	Module version not fix to current os
ERROR_HOTPLUG_POLL_NODE_NULL	No hotplug node
ERROR_HOTPLUG_MESSAGE_NULL	No hotplug message
ERROR_SIGNAL_SIGQUEUE_NODES_NULL	Not enough sigqueue node
ERROR_EXCE_LOST	Exception message lost
ERROR_LOG_LOST	Log message lost
ERROR_LOG_FMT	Invalid log format
ERROR_LOG_FDSET_NULL	Invalid fd set
ERROR_SYSTEM_HOOK_NULL	Invalid hook
ERROR_SYSTEM_LOW_MEMORY	System not enough memory
ERROR_RMS_FULL	RMS full
ERROR_RMS_NULL	Invalid RMS
ERROR_RMS_TICK	RMS tick
ERROR_RMS_WAS_CHANGED	RMS was changed
ERROR_RMS_STATUS	RMS status
ERROR_INTER_LEVEL_NULL	Invalid interrupt level
ERROR_TIME_NULL	Invalid time
ERROR_EVENT_MAX_COUNTER_NULL	Invalid event max counter
ERROR_EVENT_INIT_COUNTER	Invalid event counter
ERROR_EVENT_NULL	Invalid event
ERROR_EVENT_FULL	Event full
ERROR_EVENT_TYPE	Event type
ERROR_EVENT_WAS_DELETED	Event was delete
ERROR_EVENT_NOT_OWN	Event not own
ERROR_EVENTSET_NULL	Invalid eventset
ERROR_EVENTSET_FULL	Eventset full
ERROR_EVENTSET_TYPE	Eventset type
ERROR_EVENTSET_WAIT_TYPE	Eventset wait type
ERROR_EVENTSET_WAS_DELETED	Eventset was delete
ERROR_EVENTSET_OPTION	Eventset option
ERROR_POWERM_NODE	Invalid PowerM node
ERROR_POWERM_TIME	Invalid PowerM time
ERROR_POWERM_FUNCTION	Invalid PowerM function
ERROR_POWERM_NULL	Invalid PowerM
ERROR_POWERM_FULL	PowerM full
ERROR_POWERM_STATUS	PowerM status

B.4 Thread Error Number

Table B.4 Thread Error Number

Error Number	Description
ERROR_THREAD_STACKSIZE_LACK	Not enough stack
ERROR_THREAD_STACK_NULL	Invalid stack
ERROR_THREAD_FP_STACK_NULL	Invalid FP stack
ERROR_THREAD_ATTR_NULL	Invalid attribute
ERROR_THREAD_PRIORITY_WRONG	Invalid priority
ERROR_THREAD_WAIT_TIMEOUT	Wait timed out
ERROR_THREAD_NULL	Invalid thread
ERROR_THREAD_FULL	Thread full
ERROR_THREAD_NOT_INIT	Thread not initialized
ERROR_THREAD_NOT_SUSPEND	Thread not suspend
ERROR_THREAD_VAR_FULL	Thread var full
ERROR_THREAD_VAR_NULL	Invalid thread var
ERROR_THREAD_VAR_NOT_EXIST	Thread var not exist
ERROR_THREAD_NOT_READY	Thread not ready
ERROR_THREAD_IN_SAFE	Thread in safe mode
ERROR_THREAD_OTHER_DELETE	Thread has been delete by other
ERROR_THREAD_JOIN_SELF	Thread join self
ERROR_THREAD_DETACHED	Thread detached
ERROR_THREAD_JOIN	Thread join
ERROR_THREAD_NOT_SLEEP	Thread not sleep
ERROR_THREAD_NOTEPAD_INDEX	Invalid notepad index
ERROR_THREAD_OPTION	Invalid option
ERROR_THREAD_RESTART_SELF	Thread restart self
ERROR_THREAD_DELETE_SELF	Thread delete self
ERROR_THREAD_NEED_SIGNAL_SPT	Thread need signal support
ERROR_THREAD_DISCANCEL	Thread discancel
ERROR_THREADPOOL_NULL	Invalid threadpool
ERROR_THREADPOOL_FULL	Threadpool full
ERROR_THREADPOOL_MAX_COUNTER	Invalid threadpool Max counter

B.5 Message Queue Error Number

Table B.5 Message Queue Error Number

Error Number	Description
ERROR_MSGQUEUE_MAX_COUNTER_NULL	Invalid MQ max counter
ERROR_MSGQUEUE_MAX_LEN_NULL	Invalid MQ max length
ERROR_MSGQUEUE_FULL	MQ full
ERROR_MSGQUEUE_NULL	Invalid MQ
ERROR_MSGQUEUE_TYPE	MQ type
ERROR_MSGQUEUE_WAS_DELETED	MQ was delete
ERROR_MSGQUEUE_MSG_NULL	Invalid MQ message
ERROR_MSGQUEUE_MSG_LEN	Invalid MQ message length
ERROR_MSGQUEUE_OPTION	MQ option

B.6 TIMERError Number

Table B.6 timerError Number

Error Number	Description
ERROR_TIMER_FULL	Timer full
ERROR_TIMER_NULL	Invalid timer
ERROR_TIMER_CALLBACK_NULL	Invalid timer callback
ERROR_TIMER_ISR	In timer interrupt service
ERROR_TIMER_TIME	Invalid time
ERROR_TIMER_OPTION	Timer option
ERROR_RTC_NULL	Invalid RTC
ERROR_RTC_TIMEZONE	Invalid timezone

B.7 Memory Operation Error Number

Table B.7 Memory Operation Error Number

Error Number	Description
ERROR_PARTITION_FULL	Partition full
ERROR_PARTITION_NULL	Invalid partition
ERROR_PARTITION_BLOCK_COUNTER	Invalid partition block counter
ERROR_PARTITION_BLOCK_SIZE	Invalid partition block size
ERROR_PARTITION_BLOCK_USED	Partition used
ERROR_REGION_FULL	Region full
ERROR_REGION_NULL	Invalid region
ERROR_REGION_SIZE	Invalid region size
ERROR_REGION_USED	Region used
ERROR_REGION_ALIGN	Miss align
ERROR_VMM_LOW_PHYSICAL_PAGE	Not enough physical page
ERROR_VMM_LOW_LEVEL	Low level error
ERROR_VMM_PHYSICAL_PAGE	Physical page error
ERROR_VMM_VIRTUAL_PAGE	Virtual page error
ERROR_VMM_PHYSICAL_ADDR	Invalid physical address
ERROR_VMM_VIRTUAL_ADDR	Invalid virtual address
ERROR_VMM_ALIGN	Miss page align
ERROR_VMM_PAGE_INVALID	Invalid page
ERROR_VMM_LOW_PAGE	Low page
ERROR_DMA_CHANNEL_INVALID	Invalid DMA channel
ERROR_DMA_TRANSMSG_INVALID	Invalid DMA Transmessage
ERROR_DMA_DATA_TOO_LARGE	Data too large
ERROR_DMA_NO_FREE_NODE	No free DMA node
ERROR_DMA_MAX_NODE	Max DMA node in queue

B.8 I/O System Error Number

Table B.8 I/OError Number

Error Number	Description
ERROR_IOS_DRIVER_GLUT	Driver full
ERROR_IOS_FILE_OPERATIONS_NULL	Invalid file operations
ERROR_IOS_FILE_READ_PROTECTED	Read protected
ERROR_IOS_FILE_SYMLINK	symbol link
ERROR_IO_NO_DRIVER	No driver
ERROR_IO_BUFFER_ERROR	Buffer incorrect
ERROR_IO_VOLUME_ERROR	Volume incorrect
ERROR_IO_SELECT_UNSUPPORT_IN_DRIVER	Driver unsupport select
ERROR_IO_SELECT_CONTEXT	Invalid select context
ERROR_IO_SELECT_WIDTH	Invalid width
ERROR_IO_SELECT_FDSET_NULL	Invalid fd set

B.9 Shell Operation Error Number

Table B.9 ShellError Number

Error Number	Description
ERROR_TSHELL_EPARAM	Invalid shell parameter(s)
ERROR_TSHELL_OVERLAP	Keyword overlap
ERROR_TSHELL_EKEYWORD	Invalid keyword
ERROR_TSHELL_EVAR	Invalid variable
ERROR_TSHELL_CANNOT_OVERWRITE	Can not over write
ERROR_TSHELL_ENOUSER	Invalid user name
ERROR_TSHELL_EUSER	No user
ERROR_TSHELL_ELEVEL	Insufficient permissions
ERROR_TSHELL_CMDNOTFUND	Can not find command

B.10 Other Error Numbers

Table B.10 Other Error Numbers

Error Number	Description	Error Number	Description
EIDRM	Identifier removed	ELNRNG	Link number out of range

ECHRNG	Channel number out of range	EUNATCH	Protocol driver not attached
EL2NSYNC	Level 2 not synchronized	ENOCSI	No CSI structure available
EL2NSYNC	Level 2 not synchronized	EL2HLT	Level 2 halted
EL3HLT	Level 3 halted	EBADE	Invalid exchange
EL3RST	Level 3 reset	EBADR	Invalid request descriptor
EXFULL	Exchange full	ENOANO	No anode
EBADRQC	Invalid request code	EBADSLT	Invalid slot
EBFONT	Bad font file format	ENONET	Machine is not on the network
ENOPKG	Package not installed	EREMOTE	Object is remote
ENOLINK	Link has been severed	EADV	Advertise error
ESRMNT	Srmount error	ECOMM	Communication error on send
EMULTIHOP	Multihop attempted	EDOTDOT	RFS specific error
EUCLEAN	Structure needs cleaning	ENOTUNIQ	Name not unique on network
EBADFD	File descriptor in bad state	EREMCHG	Remote address changed
ELIBACC	Can not access a needed shared library	ELIBBAD	Accessing a corrupted shared library
ELIBSCN	.lib section in a.out corrupted	ELIBMAX	Attempting to link in too many shared libraries
ELIBEXEC	Cannot exec a shared library directly	ERESTART	Interrupted system call should be restarted
ESTRPIPE	Streams pipe error	EUSERS	Too many users
ESTALE	Stale NFS file handle	ENOTNAM	Not a XENIX named type file
ENAVAIL	No XENIX semaphores available	EISNAM	Is a named type file
EREMOTEIO	Remote I/O error	EDQUOT	Quota exceeded
ENOMEDIUM	No medium found	EMEDIUMTYPE	Wrong medium type
EILSEQ	Illegal byte sequence		

Annex C Description of SylixOS Makefile

C.1 Description of SylixOS Makefile

RealEvo-IDE generates a Makefile when creating a SylixOS project. In order to maintain the flexibility of the SylixOS building system, the generated Makefile file is allowed to be manually modified by the user. When you need to add or delete any file in your project, you must manually modify the Makefile file. This section mainly analyzes SylixOS Makefile file, to facilitate the users to modify the building function of custom SylixOS during use.

C.1.1 Directory Structure of SylixOS

The contents of the Makefile are related to the directory structure. The following are some typical directory structures in the SylixOS system:

1. Application and BSP Directory Structures

The SylixOS App projects, SylixOS Shared Lib projects, SylixOS Kernel Module projects and BSP (SylixOS BSP) projects have similar source directory structures. As can be seen from Figure C.1 and Figure C.2, the Makefile of the project is located in the root directory of the source code.

The directory structures of SylixOS application projects, shared library projects and kernel module project are as follows:

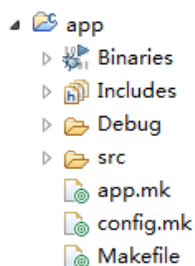


Figure C.1 Directory Structure of Project

The directory structure of the SylixOS BSP project is as below:

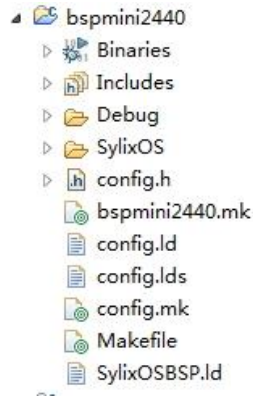


Figure C.2 Directory Structure of BSP Project

2. Directory Structure of SylixOS Base Project

SylixOS Base packs several subprojects into a unified management project. Each subproject is a subfolder under the SylixOS Base project directory, containing the subproject's Makefile file. The SylixOS Base project itself also contains the Makefile file. The Base project directory is structured as follows:

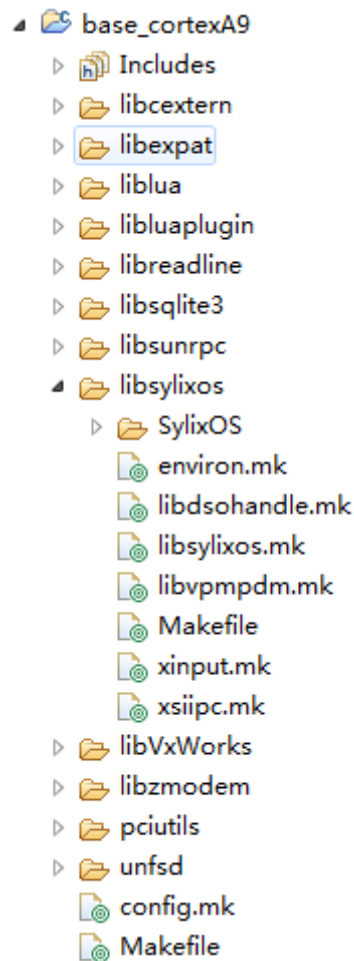


Figure C.3 Directory Structure of SylixOS Base Project

We can extract subprojects from the SylixOS Base project to build a separate project, or add new subprojects to the SylixOS Base project, the process is as follows:

- Copy project source code folder;
- Modify the SylixOS Base project's Makefile file.

The contents of the SylixOS Base project's Makefile file are as follows:

```
COMPONENTS = \
libsylixos \
libcextern \
libVxWorks \
libreadline \
liblua \
libsqlite3 \
pciutils \
libzmodem \
libexpat \
libluaplugin \
libsunrpc \
unfsd

all: $(COMPONENTS)
    @for target in $(COMPONENTS); do make -C $$target all -j100; done

clean: $(COMPONENTS)
    @for target in $(COMPONENTS); do make -C $$target clean; done
```

The above is the contents of the Makefile file of the SylixOS Base project. For the implementation, the Makefile of each subproject is called. If you want to add and delete a subproject, just add or delete the corresponding entries based on this structure.

C.1.2 config.mk File

All the projects generated by RealEvo-IDE include a config.mk file. The config.mk file is generated by RealEvo-IDE and contains the basic settings of the SylixOS building system, such as the toolchain, Base project path, and so on. When compiling SylixOS with RealEvo-IDE, the config.mk file must be modified using the RealEvo-IDE Fig. configuration interface (refer to RealEvo-IDE User Manual). If you use other development environments, you can manually modify it. The following is the instructions for config.mk file configuration.

```
#*****
# SylixOS Base Project path
```

```
*****
SYLIXOS_BASE_PATH := D:/sylixos/book/source/sylixos-base

*****
# Toolchain prefix
*****
TOOLCHAIN_PREFIX := arm-sylixos-eabi-

*****
# Debug options (debug or release)
*****
DEBUG_LEVEL = debug
*****
# NOTICE: libsylixos, BSP and other kernel modules projects CAN NOT use vfp!
*****
FPUFLAGS = disable
CPUFLAGS = arm920t
```

Variable analysis:

- SYLIXOS_BASE_PATH is the path of the SylixOS Base project. If it is SylixOS Base project itself, the default setting is "..";
- TOOLCHAIN_PREFIX is the prefix of toolchain;
- DEBUG_LEVEL is the debugging mode (including Debug and Release);
- FPUFLAGS is setting of a floating-point processor, and its value varies according to the compiler. The FPUFLAGS configuration is only effective for applications and shared library projects by default;
- CPUFLAGS is the processor mode and its value varies according to the compiler.

C.1.3 Makefile File

The Makefile file is generated by RealEvo-IDE when the project is created, allowing manual modification. This section describes its structure. Due to the huge amount of contents in the Makefile file, this section only describes the key configuration.

1. config.mk File Location

As mentioned earlier, the SylixOS Base project consists of several subprojects. These subprojects can also be constructed as separate projects. So the config.mk file exists in both the subproject top folder and the SylixOS Base project top folder. When calling the Makefile file of a sub-project, Makefile will first search the config.mk file in the parent folder. If such file does not exist, Makefile will search it in the same folder. The source codes are as follows:

```

#*****
# include config.mk
#*****
CONFIG_MK_EXIST = $(shell if [ -f ../config.mk ]; then echo exist; else echo
notexist; fi;)
ifeq ($(CONFIG_MK_EXIST), exist)
include ../config.mk
else
CONFIG_MK_EXIST = $(shell if [ -f config.mk ]; then echo exist; else echo notexist;
fi;)
ifeq ($(CONFIG_MK_EXIST), exist)
include config.mk
else
CONFIG_MK_EXIST =
endif
endif

```

2. Target Makefile

The target Makefiles for application projects, shared library projects, kernel module projects, and BSP projects are defined by the following statements. Multiple targets can be compiled by including different *.mk files:

```

#*****
# Include targets makefiles
#*****
include app.mk

```

The following section of the custom target Makefile defines the list of compiled source files:

```

#*****
# Source list

```



```
*****  
LOCAL_SRCS = \  
src/app.c
```

Variable analysis:

- 变量 LOCAL_SRCS 是
The variable LOCAL_SRCS is the list of source file paths. The paths use the directory where the Makefile is located as the current directory. The LOCAL_SRCS is generally used to add files to the project.

Note: If the option “Custom Makefile configuration” in RealEvo-IDE is selected, RealEvo-IDE will automatically scan the added new file and add it to the LOCAL_SRCS list.

3. Header File Path

The following code sets the header file path.

```
*****  
# Header file search path (eg. LOCAL_INC_PATH := -I"Your header files search  
path")  
*****  
LOCAL_INC_PATH := \  
-I"include"
```

Variable analysis:

- LOCAL_INC_PATH is a list of header files.

Note: The list of header paths can be configured via RealEvo-IDE as shown in Figure C.4.

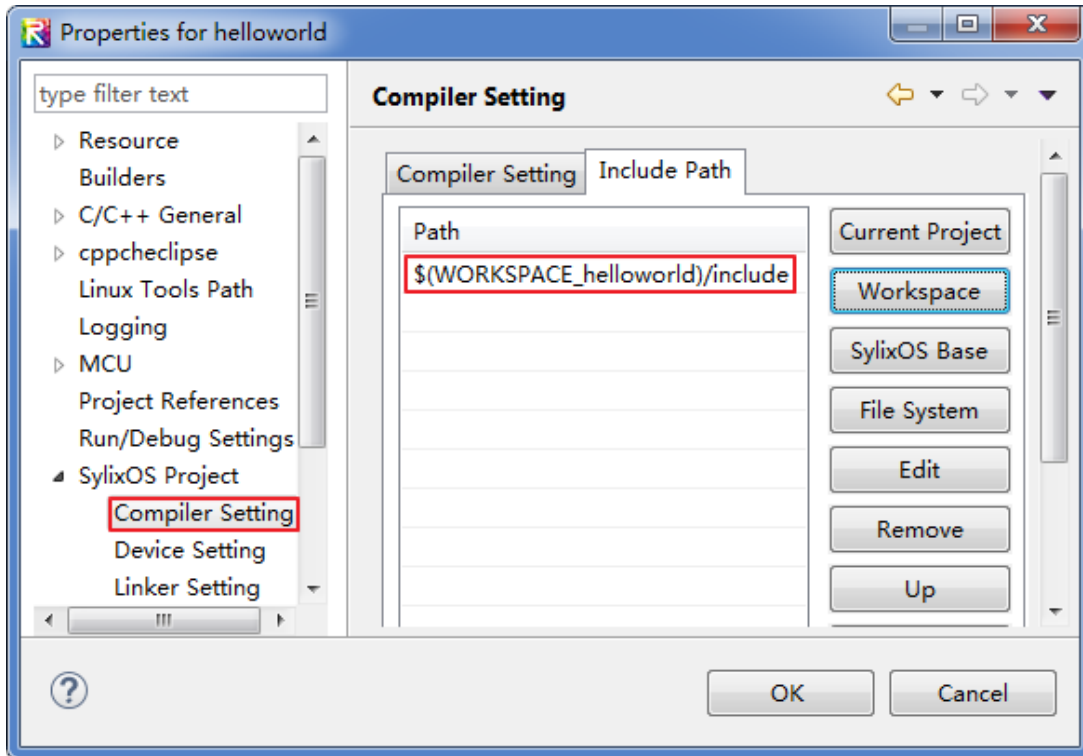


Figure C.4 Configuration of Header File Path

4. Predefined Macros

The following code sets the predefined macros of the compiler.

```
#*****  
# Pre-defined macro (eg. -DYOUR_MACRO=1)  
#*****  
LOCAL_DSMBOL := -DDEBUG
```

Variable analysis:

- LOCAL_DSMBOL is a list of predefined macros.

Note: The predefined macros can be configured via the RealEvo-IDE as shown in Figure C.5.

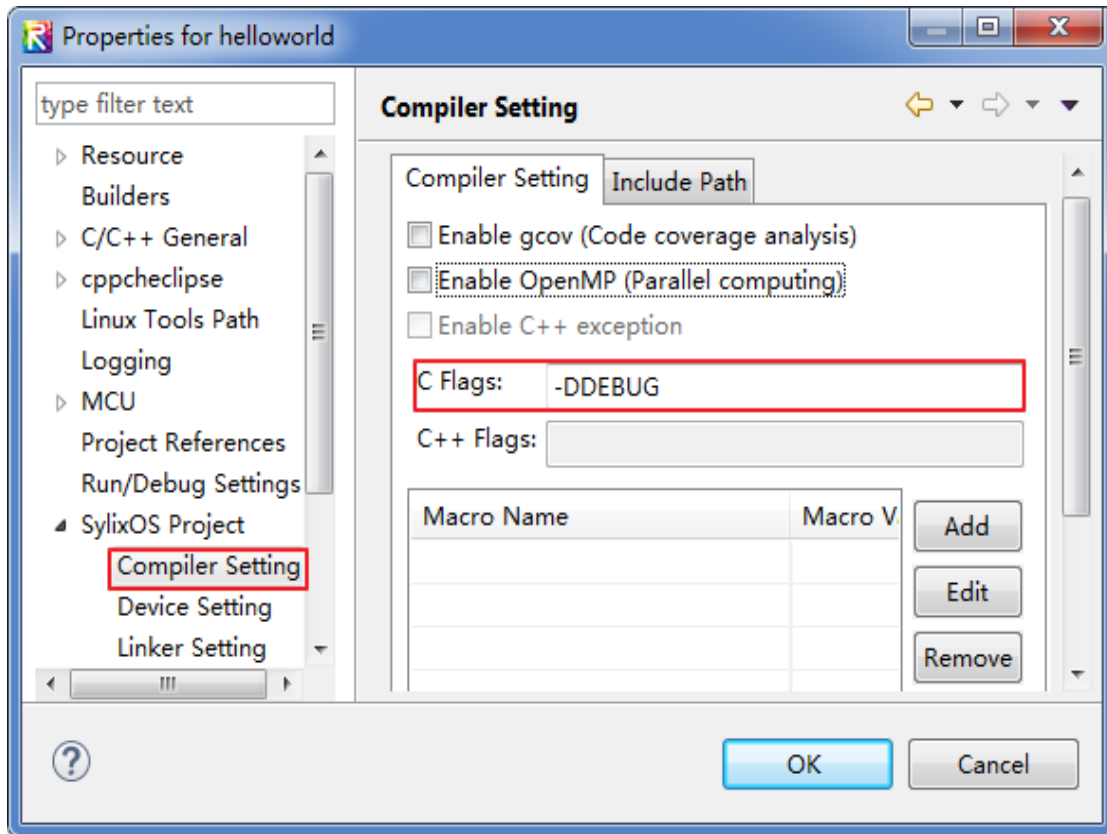


Figure C.5 Configure of Predefined Macros

5. Shared Library

The following code sets the shared library that the application depends on and the path thereof.

```

*****
# Depend library (eg. LOCAL_DEPEND_LIB := -la LOCAL_DEPEND_LIB_PATH := -L"Your
library search path")
*****
LOCAL_DEPEND_LIB := \
-lhellolibrary

LOCAL_DEPEND_LIB_PATH := \
-L"../hellolibrary/$(OUTDIR) "

```

Variable analysis:

- LOCAL_DEPEND_LIB is a list of shared libraries;
- LOCAL_DEPEND_LIB_PATH is a list of shared library paths.

Note: The list of shared libraries can be configured via RealEvo-IDE as shown in Figure C.6.

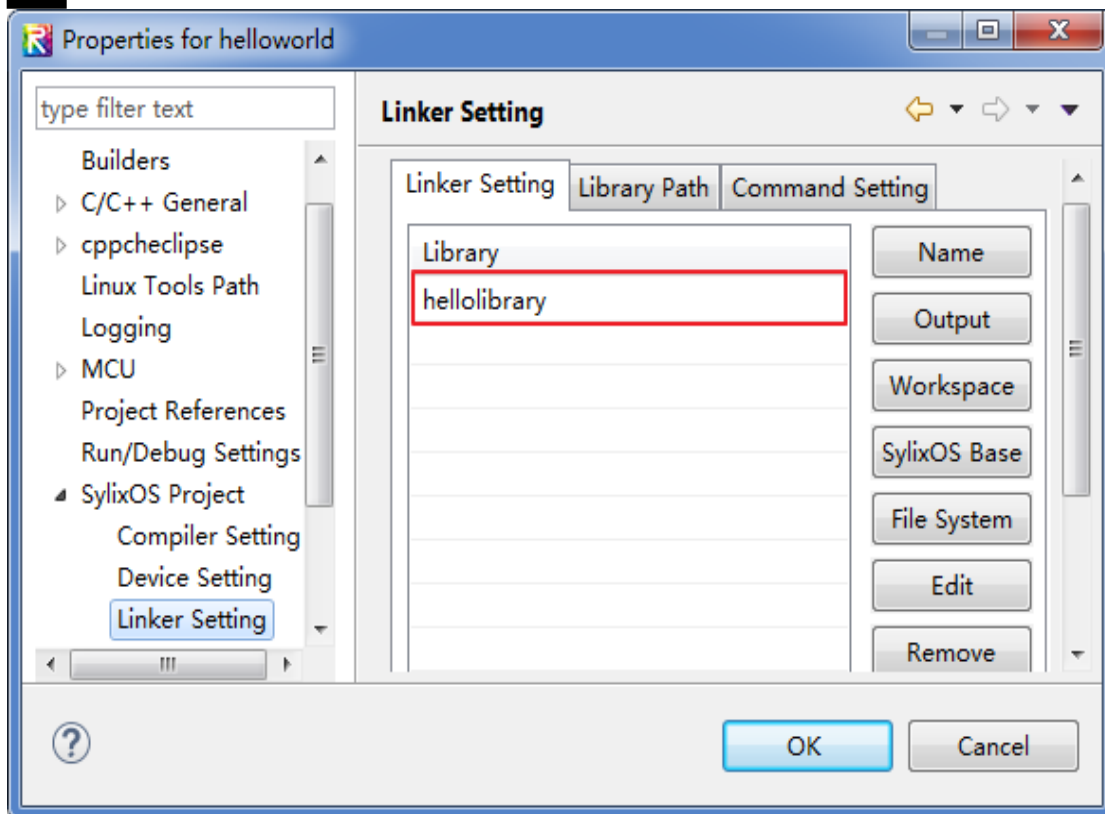


Figure C.6 Configuration of the List of Shared Library

6. Code Coverage

The following sections describe the configuration supporting the code coverage:

```

#*****
# Code coverage config
#*****
LOCAL_USE_GCOV := no

```

Note: The code coverage function can be configured and enabled via the RealEvo-IDE as shown in Figure C.7.

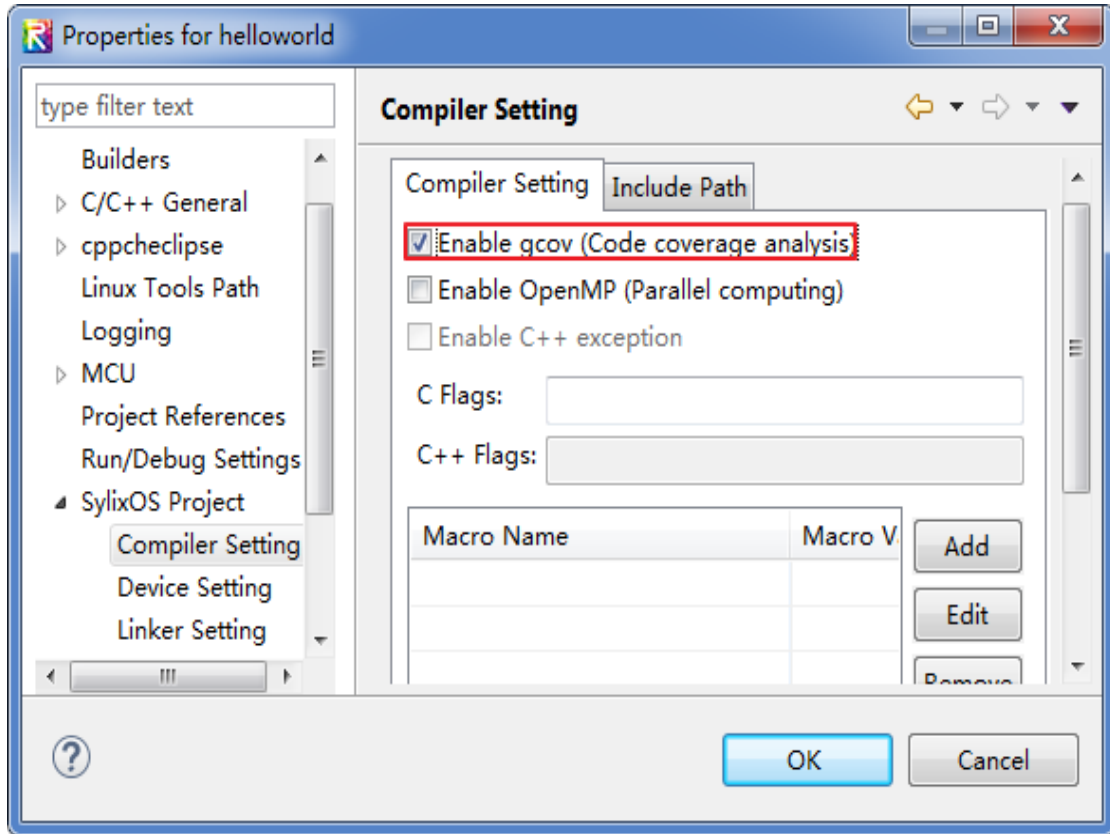


Figure C.7 Enabling of Code Coverage

7. Object file

Different files are generated in different types of projects. For the SylixOS kernel project, the following object files are inherently generated:

- Libsylixos.a is the kernel library file;
- Libvpmpdm.so, libvpmpdm.a is the process patch library file;
- Xin xinput.ko and xsiipc.ko are the driver module files that comes with the kernel;
- symbol.c and symbol.h are the kernel symbol table source files, used to establish the system kernel symbol table.

The following section defines the target types for compilation:

```
include $(APPLICATION_MK)
```

SylixOS includes the following compilation target types:

- APPLICATION_MK is used to compile common applications;
- BSP_MK is used to compile BSP programs;
- KERNEL_LIBRARY_MK is used to compile the kernel library file;
- KERNEL_MODULE_MK is used to compile kernel module file;
- LIBRARY_MK is used to compile the application library file;
- LIBSYLIXOS_MK is used to compile the SylixOS kernel;
- UNIT_TEST_MK is used to compile the multi-object file for unit test.

8. BSP link script

```
*****
# load script
*****
LOCAL_LD_SCRIPT = SylixOSBSP.ld
```

Variable analysis:

- LOCAL_LD_SCRIPT is a link script file.



Annex D Description of SylixOS Open Source Community

The SylixOS open source community, created by Beijing ACOINFO and jointly maintained by SylixOS enthusiasts, is a public platform for SylixOS open source software learning and communication. Here, you can learn and consult all updates and released information about SylixOS. The community contains the following sections:

- **Community News:** Through this section, you can learn about the recent events of SylixOS;
- **Application Porting and Development:** This section contains a discussion of issues related to SylixOS application porting and development;
- **Driver Development:** This section mainly discusses the development of drivers, or the development of BSP programs, etc;
- **OS Porting:** Through this section, you can find the knowledge about SylixOS operating system and SylixOS simulator;
- **Kernel Function Discussion:** Here you can learn SylixOS kernel function, or you can put forward suggestions to improve the kernel function;
- **Network Communication Development:** Here you can ask or answer questions about SylixOS network;
- **Business Case Presentation:** This section will present the cases about the application of SylixOS in business;
- **Integrated Development Environment:** You can solve the problems encountered in using the SylixOS integrated development environment through this section;
- **Technical Communication:** This is a technology convergence section where you can learn various kinds of technical knowledge.
- **Lurk and Roasting:** This is an open section to discuss various life issues.

SylixOS enthusiasts can freely speak in these sections to ask questions related to SylixOS, and through the SylixOS open source community, you can quickly learn about the latest developments in SylixOS.

References

Operating Systems : Internals And Design Principles (7th Edition) , William Stallings

This book describes in depth the principles of the operating system.

Unix Network Programming, Volume 1: The Sockets Networking API(3rd Edition),
W.Richard Stevens/Bill Fenner et.al.

This book describes the web socket programming method.

A Linux and UNIX System Programming Handbook, Michael Kerrisk

This book describes in depth the programming method in the Linux environment.

TCP/IP Illustrated, Volume 1: The Protocols , W.Richard Stevens

This is a complete and detailed guide to the TCP/IP protocol.

TCP/IP Illustrated, Vol 2: The Implementation, W.Richard Stevens

This book provides an in-depth analysis of the implementation of the TCP/IP protocol.

Advanced VxWorks Program Design, Li Fangmin

This book describes in detail the programming method of VxWorks.

Advanced Programming in the UNIX Environment, W.Richard Stevens/Stephen
A.Rago

This book provides a comprehensive and in-depth description of the programming method in the UNIX environment.

Embedded Real-time System Design Based on VxWorks, Wang Jingang et.al.

This is a textbook that introduces the principles and applications of embedded systems.