GE Fanuc Automation

Programmable Control Products

C Programmer's Toolkit for Series 90™ PLCs

User's Manual

GFK-0646E *August 1998*

Warnings, Cautions, and Notes as Used in this Publication

Warning

Warning notices are used in this publication to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist in this equipment or may be associated with its use.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.

Caution

Caution notices are used where equipment might be damaged if care is not taken.

Note

Notes merely call attention to information that is especially significant to understanding and operating the equipment.

This document is based on information available at the time of its publication. While efforts have been made to be accurate, the information contained herein does not purport to cover all details or variations in hardware or software, nor to provide for every possible contingency in connection with installation, operation, or maintenance. Features may be described herein which are not present in all hardware and software systems. GE Fanuc Automation assumes no obligation of notice to holders of this document with respect to changes subsequently made.

GE Fanuc Automation makes no representation or warranty, expressed, implied, or statutory with respect to, and assumes no responsibility for the accuracy, completeness, sufficiency, or usefulness of the information contained herein. No warranties of merchantability or fitness for purpose shall apply.

Preface

This manual contains essential information about the construction of C applications for Series 90™ -70 and 90-30 programmable controllers. This manual is written for the experienced programmer who is familiar with both the C programming language and with the operation of Series 90 PLCs. Readers new to the C programming language or to Series 90 PLCs may wish to familiarize themselves with those topics by first reading publications listed at the end of this section.

Content of this Manual

Chapter 1. Introduction: describes the C Programmer's Toolkit, the types of C applications that can be created, and how they can be appplied to control applications.

Chapter 2. Installation: explains how to install the C Programmer's Toolkit and installation requirements for Microsoft® C compiler.

Chapter 3. Writing a C Application: describes the creation of a C application and some important differences between the Series 90-70 and 90-30 applications.

Chapter 4. Example Series 90-70 C Application Development: describes the Series 90-70 examples provided in the C Programmer's Toolkit and presents step-by-step descriptions of how to build and debug a C application under MS-DOS® and how to build, store, and debug a C application in the Series 90-70 PLC.

Chapter 5. Example Series 90-30 C Application Development: describes the Series 90-30 examples provided in the C Programmer's Toolkit and presents step-by-step descriptions of how to build, store, and debug a C application in the Series 90-30 PLC.

Chapter 6. C Application Development Using Multiple C Source Files: describes the files in the MULTISRC example subdirectory and how to apply the concepts used in the MULTISRC example in your C application development.

Chapter 7. C Application Debugger: describes the installation and operation of the C Debugger in the Series 90-70 PLC environment.

Chapter 8. GE Fanuc Support Services and Consultation: describes the consultation services provided by GE Fanuc to each purchaser of the C Programmer's Toolkit.

Appendix A. Standard C Library Functions Supported in the Series 90 PLC: lists each of the standard C library routines provided by Microsoft and the Series 90 PLC functions provided in the C Programmer's Toolkit that are available for use in C applications within the Series 90 PLC.

Appendix B. C Programming Toolkit Files: lists each of the files (and associated MS-DOS directory) created during the installation of the C Programmer's Toolkit.

Appendix C. C Macros for PLC Access: lists the macros provided by the C Programmer's Toolkit to ease accessing PLC reference and fault memory.

Appendix D. Calculating PLC Memory Usage for a C Application: describes how to compute the amount of PLC user program memory required for a given C application.

Appendix E. 90-70 CPU Execution Time for printf( ): describes how executing a printf( ) function impacts the PLC sweep of executing a printf( ) function.

Related Publications

For more information, refer to these publications:

Series 90™ -70 Programmable Controller Installation Manual (GFK-0262): This manual describes the modules of a Series 90-70 PLC system and explains system setup.

Logicmaster™ 90-70 Programming Software User's Manual (GFK-0263): This manual describes how to use the Logicmaster 90-70 programming software to program, configure, monitor, or control a Series 90-70 PLC and/or a remote drop.

Series 90™ -70 Programmable Controller Reference Manual (GFK-0265): This manual describes system operation of a Series 90-70 PLC system, fault explanation and correction, and the Series 90-70 instruction set.

Series 90™ -30 Programmable Controller Installation Manual (GFK-0356): This manual describes installation of a Series 90-30 PLC system.

Using CIMPLICITY Control (GFK-0263): This manual describes how to use the CIMPLICITY Control software to program, configure, monitor, or control a Series 90 PLC.

Series 90™ -30/20/Micro Programmable Controller Reference Manual (GFK-0467): This manual describes system operation of a Series 90-30 PLC system, fault explanation and correction, and the Series 90-30 instruction set.

The C Primer, Hancock, Les, and Morris Krieger. New York: McGraw-Hill Book Co., Inc., 1982

C: A Reference Manual. Harbison, Samuel P. and Steele, Greg L. Englewood Cliffs, New Jersey: Prentice-Hall Software Series, 2nd Edition, 1987.

The C Programming Language. Kernighan, Brian W., and Ritchie, Dennis M. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 2nd Edition, 1987.

Programming in C. Kochan, Stephen. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Learning to Program in C. Plum, Thomas. Cardiff, New Jersey: Plum Hall, Inc., 1983.

We Welcome Your Comments and Suggestions

At GE Fanuc Automation, we strive to produce quality technical documentation. After you have used this manual, please take a few moments to complete and return the Reader's Comment Card located on the next page.

David Bruton
Sr. Technical Writer

Chapter 1 Introduction

Release 4.0 of the Series 90-70 C Programmer's Toolkit and the Series 90-30 C Programmer's Toolkit contains libraries, utilities, and documentation required to create C applications for the Series 90 PLC. C blocks, C function blocks (hereafter know as C FBKs), and standalone C programs are constructed using the ANSI C programming language and standard tools on a personal computer. The C blocks and C FBKs are imported into a 90-70 PLC application program through the use of the Logicmaster 90-70 Librarian functions, while standalone C programs are placed into a Logicmaster folder by the C Programmer's Toolkit. Using the Logicmaster 90-70 Librarian, C blocks and C FBKs can be called from ladder logic or invoked by an I/O or timed interrupt. In the Windows-based programming software, use the New Block or New Program feature to insert C subroutine blocks or C main programs. The standalone C programs operate independent of the ladder logic and of each other.

There are five types of C applications:

- **Series 90-70 C Blocks**: C blocks are imported into a 90-70 PLC application through the Logicmaster Librarian functions. C blocks may be called from ladder logic or invoked by an I/O or timed interrupt.
- **Series 90-70 C Function Blocks (C FBKs)**: C FBKs are imported into a 90-70 PLC application through the Logicmaster Librarian functions.
- **Series 90-70 Standalone C Programs**: Standalone C programs are imported into a 90-70 PLC application by the C Programmer's Toolkit. Standalone C programs operate independent of the ladder logic and of each other. Standalone C programs may also be imported into an Logicmaster 90-70 folder via the standalone merge utility available in Release 6 and later of Logicmaster 90.
- **Series 90-30 C Main Programs**: Using the C Programmer's Toolkit and Release 2 of the Windows-based programming software, beginning with Release 8 351 and 352 CPUs, the Main program can be a C program instead of the main program being LD or SFC (for LD and SFC it is called_MAIN). When doing so, there can be no subordinate blocks of any kind, and the C Main program must have the same name as the Resource.
- **Series 90-30 C Subroutine Blocks**: Unlike the 90-70 C blocks, the 90-30 C subroutine blocks cannot have parameters associated with them. You can have multiple 90-30 C subroutine blocks.

In this document, the word *blocks* refers to C blocks, C FBKs, and C subroutine blocks unless otherwise specified.

C blocks and C function blocks are compatible with 90-70 CPUs Release 4.0 and greater and are best suited to solving custom calculations not available in standard Series 90

instructions. C blocks are the same as the C applications built with previous versions of the toolkit. The build procedure and functionality of C blocks match previous versions to the C Programmer's Toolkit. Beginning with Release 8, models 351 and 352 90-30 CPUs can accommodate C subroutine blocks and a C main program.

A **Series 90-70 C block** may be up to 64,000 bytes in size, provided there is sufficient PLC memory. Examples of calculations which might be performed in C blocks include:

- Ramp / soak profiling
- Lead / lag calculation
- Message generation
- Input selection

C FBKs (Series 90-70 only) are a faster type of C block. C FBKs do not support calls to any of the Microsoft runtime library functions. This restriction reduces the startup code size thereby improving execution time for C FBKs. C FBKs are imported into an Logicmaster 90 folder in the same manner as C blocks. C FBKs are compatible with 90-70 CPU Release 4.0 and later. The same C FBK may be called from the ladder logic program AND used as an interrupt block. C FBKs are better suited for simpler calculations than C blocks because they do not support calls to runtime library functions. C FBKs are suited for calculations including:

- Arithmetic operations
- PID
- Sorting, moving and copying data

Standalone C programs (Series 90-70 only) are also supported in Release 3.0 or later of the C Programmer's Toolkit. Standalone C programs run independently of the ladder program and other standalone C programs. Standalone C programs require Release 6.0 or later 90-70 CPUs and can support programs of up to 512 kilobytes. Standalone C programs are best suited for applications controlling an entire process or performing long calculations including:

- Interpreters
- Complete control application
- Complete diagnostic applications

Release 4.0 of the 90-70 C Programmer's Toolkit (Professional version), now supports source level, full symbolic debugging of C applications executing in Release 6.0 (or later) 90-70 PLCs (please refer to Chapter 7, "C Application Debugger," for more information).

Developing 90-70 C applications requires Version 6.0, Version 7.0, or Version 8.0 (Visual C / C++ ™ Version 1.00, 1.50, and higher) of the Microsoft® C compiler and Version 6.0 or later of the Microsoft assembler (MASM). (Please note that Microsoft C Version 6.0 requires MASM Version 5.1 or later.) The 90-70 C Programmer's Toolkit includes a utility that must be used to create a version of the Microsoft standard libraries in which the CPUs data segment (DS) and stack segment (SS) are not assumed to be the same.

C main programs and **C subroutine blocks** (Series 90-30 only) are supported in Release 4.0 or later of the 90-30 C Programmer's Toolkit. C subroutine blocks may be up to 81,920 bytes in size. Both C main programs and C Subroutine blocks require Release 8.0 90-30 (351 and 352) CPUs. C main programs and C subroutine blocks are suited for calculations including:

- Ramp / soak profiling
- Lead / lag calculation
- Input selection
- Arithmetic operations
- PID
- Sorting, moving and copying data

Developing 90-30 C applications requires Version 8.0 (Visual C / C++ Version 1.00, 1.50, and higher) of the Microsoft C compiler which comes with the toolkit.

Maximum block length may be up to 81,920 bytes.

Chapter 2 Installation

This chapter explains how to install the Series 90-70 or 90-30 C Programmer‘s Toolkit software on your personal computer and how to create a version of the standard C libraries which is compatible with the Series 90-70 PLC.

This chapter is divided into three sections. The first two sections describe the two steps required to prepare your computer for developing Series 90-70 C applications. The necessary equipment and software packages required for the installation process are described below. After that, each section describes one step of the installation process in detail. The third section describes the steps for installing the Series 90-30 version of the C Programmer‘s Toolkit.

Section	Title	Description	Page
1	Installing the C Programmer’s Toolkit for Series 90-70 and 90-30 PLCs	Describes how to install the C Programmer’s Toolkit for Series 90-70 and 90-30 PLC on your personal computer.	2-3
2	Installing the Microsoft Visual C Compiler	Describes how to install the Release 8 Microsoft Visual C compiler (MSVC) that comes with the Toolkit package. For information about installing priviously released compilers, refer to Appendix F.	2-7

What You Will Need

Before you can begin the installation procedure, you must have the following equipment:

- An MS-DOS based computer with a hard disk and MS-DOS Version 3.0 or later. If Microsoft C Version 8.0 (Microsoft Visual C / C++ v1.00 or later) is used, Microsoft Windows v3.1 and MS-DOS 5.0 or later are required.

Note

- Microsoft C Version 6.0, Version 7.0, or Visual C++ software.
- Series 90 C Development Software: C Programmer's Toolkit for Series 90-70 and Series 90-30, Standard version (IC641SWP709) or C Programmer's Toolkit for Series 90-70 and Series 90-30, Professional version (IC641SWP719).

Note

(Series 90-30 Only) The Series 90-30 Toolkit only supports Microsoft C Version 8.00 (Microsoft Visual C / C++ version 1.0 or later), which is supplied with the Toolkit.

Section 1: Installing the C Programmer's Toolkit for Series 90-70 and 90-30 PLCs

This section describes how to install the C Programmer's Toolkit software for Series 90-70 and 90-30 PLCs on your personal computer and how to set up your computer to use the Toolkit.

Note

The C Programmer's Toolkit (IC641SWP709E) includes the following disks:
One (1) 3.5 inch disk (C Programmer's Toolkit for Series 90-70 PLCs)
One (1) 3.5 inch disk (C Programmer's Toolkit for Series 90-30 PLCs)
Four (4) 3.5 inch disks (Microsoft C Compiler disks)

The Professional Package (IC641SWP719B) includes all of the above disks plus the purchased S/W package for the Softscope Debugger

Installing a Toolkit

From the MS-DOS Prompt

You will perform Step 1 differently from the Run command in Windows 95 or Windows NT 4.0 than from the DOS prompt, but the other steps are the same.

1. As noted above, there are other diskettes in the package, but the C Programmer's Toolkit itself is on one 3.5-inch installation diskette (i.e., one for the 90-30 and a separate one for the 90-70). Put your 3.5-inch installation diskette in your 3.5- inch drive. Type **`a:install`** (or **`b:install`** if *b* is the drive letter designating your 3.5-inch drive) and press the Enter key.

From the Start Menu in Windows 95 and Windows NT 4.0

1. As noted above, there are other diskettes in the package, but the C Programmer's Toolkit itself is on one 3.5-inch installation diskette (i.e., one for the 90-30 and a separate one for the 90-70). Put your 3.5-inch installation diskette in your 3.5- inch drive. Click the start button and select Run. If the Run dialog box does not default to the A drive (or the B drive if your 3.5 disk drive is *b*), use the down arrow button or the Browse button to select **`install.exe`**; then press the OK button.

From the File Manager or Explorer

1. As noted above, there are other diskettes in the package, but the C Programmer's Toolkit itself is on one 3.5-inch installation diskette (i.e., one for the 90-30 and a separate one for the 90-70). Put your 3.5-inch installation diskette in your 3.5- inch drive. In File Manager, click the A button to open a window displaying the contents of the A drive (or the B drive if your 3.5 disk drive is *b*). In Explorer, click the A drive

graphic (or the B drive if your 3.5 disk drive is *b*) under My computer. Then double-click **install.exe**.

Note

The diskettes are write-protected and should be kept safe after installation.

2. In response to the prompt from the installation program, type the letter of your hard drive.
3. The installation program creates the following directories on the specified hard drive:

If installing the 90-30 toolkit:

```
\s9030c
\s9030c\example1
\s9030c\example2
\s9030c\multisrc
```

If installing the 90-70 toolkit:

```
\s9070c
\s9070c\example1
\s9070c\example2
\s9070c\multisrc
\s9070c\exfbk
\s9070c\exsap
```

The files listed in Appendix B are copied to these directories and the libraries are registered.

Updating the AUTOEXEC.BAT (or AUTOEXEC.NT) File

The **autoexec.bat** file must be modified in order for the C Programmer's Toolkit to work. The **PATH** statement must be updated to include the path to the toolkit, and an environment variable must be defined for the C Programmer's Toolkit macros.

Note

If you are installing both the 90-30 and the 90-70 toolkit, your **autoexec.bat** file will need to include the directory path designators shown below, e.g., both **\s9070c** and **\s9030c.** If you are using only one of the two toolkits, you will only need to specify the path appropriate for that toolkit.

The installation routine will modify your **autoexec.bat** file to include these changes and place it in **autoexec.bat** under **\s9070c** and **\s9030c** (if both versions are being installed). The installation routine will offer to put this path into the root **autoexec.bat** file. If you decline this you must either have the changes in place already or make the changes yourself, as described below.

Note

On a Windows NT® system, you can adjust your AUTOEXEC.NT file instead. For either Windows NT, Windows 95, or Windows 3.x, you can make these changes through a batch file that you run prior to running the Toolkit.

Updating the MS-DOS Search Path

The MS-DOS operating system in your computer uses the search path to find programs that are not in the current directory. As noted above, the installation routine will modify your **autoexec.bat** file to include a path to the toolkit directories, or you can modify the file yourself. The **<drive>:\s9070c** directory or the **<drive>:\s9030c** directory (or both), where **<drive>** is the name of the drive on which you installed the C Programmer's Toolkit, should be added to the search path so that the C development software tools can be used without typing the name of the directory where they are stored.

To determine what directories are currently in the search path, type **path** at the MS-DOS prompt and press the Enter key. MS-DOS will display a list of the directories, separated by semicolons, in the current search path on your computer's display. If no search path has been defined, the words "No Path" are displayed on the screen. If the **<drive>:\s9030c** directories and **<drive>:\s9070c** are not already included in the search path, they should be added.

The search path is defined in the **autoexec.bat** file, located in the root directory of the disk drive from which MS-DOS is started (the boot drive, which is usually drive C). Using any text editor program, edit the **autoexec.bat** file. If the file does not contain a **PATH** command similar to **path=c:\dos**, add:
path=<drive>:\s9070c;c:\s9030c as the first line of the file—see Note below. (The **<drive>** means you should put the name of the drive here, usually **C** or **D**.)

If there is already a **PATH** command, add: **;<drive>:\s9070c** or **;<drive>:\s9070c** (or both) *at the end* of the path definition.

Note

If you have the 90-70 Toolkit in addition to the 90-30 Toolkit which you are installing, the **<drive>:\s9030c** (e.g., **c:\s9030c)** part of your path statement ***mustfollow*** the **c:\s9070c** path designation; for example: **path=c:\s9070c;c:\s9030c** (in a real example, there would be other parts to the path).

If there is no **autoexec.bat** file in the root directory of the boot drive, create one with your text editor. Include only a **PATH** command, as above, which specifies the **<drive>:\s9030c** or **<drive>:\s9070c** directory on the correct hard drive.

Note

The **autoexec.bat** file may begin with an ECHO OFF command on the first line. If one is present, the **PATH** command should appear on the second line.

Adding the Environment Variable

The C Programmer's Toolkit requires an environment variable to access files it needs. The **C30_PATH** and **C70_PATH** (or both) macros should be added to the **autoexec.bat** to provide this path. Add the line:

`set C30_PATH=<drive>:\s9030c`
Or
`set C70_PATH=<drive>:\s9070c` to your **autoexec.bat** file, where **<drive>** is the name of the drive on which you installed the C Programmer's Toolkit (e.g., **`set C70_PATH=c:\s9070c`**). If you are installing both, you need to add both environment variable statements to your **`autoexec.bat`** file.

Section 2: Installing the Microsoft Visual C Compiler

Release 4.0 of the C Programmer's Toolkit ***comes with Release 8.0 of the Microsoft Visual C compiler (MSVC)***. If you are installing an earlier version of the MSVC, refer to Appendix F, "Installing Earlier Compilers."

Note

If you are planning on using the C Programmer's Toolkit for Series 90-30 PLCs, Release 8.0 of MSVC is the only compiler that is compatible.

From the MS-DOS Prompt

You will perform Step 1 differently from the Run command in Windows 95 or Windows NT 4.0 than from the DOS prompt, but the other steps are the same.

1. MSVC comes on four 3.5-inch installation diskettes. Select the first installation diskette and place it in the drive. Type **`a:install`** and press the Enter key.

Note

If you have a monochrome LCD or plasma screen, enter **`a:install /nocolor`**.

From the Start Menu in Windows 95 and Windows NT 4.0

1. MSVC comes on four 3.5-inch installation diskettes. Select the first installation diskette and place it in the drive. Click the start button and select Run. If the Run dialog box does not default to the A drive, use the down arrow button or the Browse button to select **`install.exe`**; then press the OK button.

From the File Manager or Explorer

1. MSVC comes on four 3.5-inch installation diskettes. Select the first installation diskette and place it in the drive. In File Manager, click the A button to open a window displaying the contents of the A drive. In Explorer, click the A drive graphic under My computer. Then double-click **`install.exe`**.

Note

The diskettes are write-protected and should be kept safe after installation.

2. In response to the prompt from the installation program, type the letter of your hard drive.

3. The installation program creates the following directories on the specified hard drive:

```
\MSVC
\MSVC\HELP
\MSVC\BIN
\MSVC\INCLUDE
\MSVC\LIB
```

Updating the System Files

AUTOEXEC.BAT

Microsoft C command line tools use several of your PC's environment variables. In MS-DOS, Windows 3.1, Windows 3.11 and Windows 95 these environment variables can be set conveniently from AUTOEXEC.BAT each time your PC starts. INSTALL modifies the PATH statement in AUTOEXEC.BAT:

```
PATH C:\MSVC\BIN;...
```

and adds these commands:

```
set HELPFILES=C:\MSVC\HELP
set INCLUDE=C:\MSVC\INCLUDE
set LIB=C:\MSVC\LIB
```

In order to use Microsoft C command line tools from inside a Windows 3.x or Windows 95 application (for example, a file editor that supports compilation from a menu or button), you must define these environment variables from AUTOEXEC.BAT. In Windows NT, they are assigned using the System tool in Control Panel.

If you run Microsoft C command line tools only from an MS-DOS prompt window, you can assign the environment variables by running

```
>C:\MSVC\BIN\MSVCVARS.BAT
```

from the MS-DOS prompt in the same window.

If you elected NOT to let INSTALL change the working copy of AUTOEXEC.BAT and you want to do so now, rename the existing version and copy the modified version to the boot directory:

```
> C:
> CD \
> REN AUTOEXEC.BAT AUTOEXEC.MC0
> COPY C:\MSVC\AUTOEXEC.BAT
```

If you installed the software at a different drive or directory, change the COPY command accordingly.

CONFIG.SYS

If your PC has a CONFIG.SYS file in the boot directory, INSTALL checks it for a FILES command:

```
FILES=50
```

Microsoft C opens a lot of files. If the FILES value is less than 50, INSTALL modifies CONFIG.SYS. This change ***must*** be made to avoid errors. If you elected ***not*** to let INSTALL change the working copy of CONFIG.SYS, you must rename the existing version and copy the modified version to the boot directory:

```
> C:
> CD \
> REN CONFIG.SYS CONFIG.MC0
> COPY C:\MSVC\CONFIG.SYS
```

If you installed the software at a different drive or directory, change the COPY command accordingly.

SYSTEM.INI

For Windows 3.1 only, INSTALL modifies the [386Enh] section of SYSTEM.INI in the Windows directory (C:\WINDOWS by default) as follows:

```
[386Enh]
device=c:\msvc\bin\dosxnt.386
```

If you use Windows 3.1, this change ***must*** be made to avoid errors. If you elected ***not*** to let INSTALL change the working copy of SYSTEM.INI, you must rename the existing version and copy the modified version to the Windows directory. First, find the Windows directory:

```
> DIR /S C:\SYSTEM.INI
```

The DIR command should find at least two copies of SYSTEM.INI: the working copy in the Windows directory and the modified copy created by INSTALL. If your computer has more than one Windows directory, find the directory for the working copy by using the SET command:

```
> SET|FIND /I "WINDIR"
```

followed by Enter. MS-DOS should print something like

```
windir=C:\WINDOWS
```

to your screen. Use your actual Windows directory in the steps below:

```
> C:
> CD \WINDOWS
> REN SYSTEM.INI SYSTEM.MC0
> COPY C:\MSVC\SYSTEM.INI
```

If you installed the software at a different drive or directory, change the COPY command accordingly.

Setting Up Windows Help:

Note

There is no manual available for the MSVC compiler. These online help files are the only documentation available for this software.

If you use this software on a PC running Microsoft Windows 3.1 or later, you can access the online help files provided with this distribution.

Windows 3.1, Windows 3.11 or Windows NT 3.51

1. Select File / New.. from the Program Manager tool bar.
2. In the New Program Object dialog, click the Program Group or Personal Program Group button and then click OK.
3. In the Program Group Properties dialog, type "Microsoft C version 8.00c" in the Description box (Do not include the quote marks.) and click OK. A new program group window will be created.
4. Open File Manager and arrange its window so that the new program group window and File Manager are both visible on your desktop.
5. In File Manager, select the C:\MSVC\HELP folder.
6. Hold the Ctrl key down and click on ERRORS.HLP, MSCXX.HLP, README.HLP and TOOLS.HLP. With the Ctrl key still down, position the mouse cursor over one of the highlighted files and then press and hold the left mouse button.
7. Drag the group of files to the new program group window and release the mouse button and then the Ctrl key. The new window will have four Help icons.
8. Select the MSCXX icon and then select File / Properties... from the Program Manager tool bar. In the Program Item Properties dialog, change the Description from **`Mscxx`** to **`Language Reference`**.

Windows 95 or Windows NT 4.0

1. From the Start button, select Settings and Taskbar...
2. On the Taskbar Properties sheet, select the Start Menu Programs tab and click the Add button.
3. In the Create Shortcut window, click the Browse button. In the Browse window, double click the MSVC folder and then the Help folder. Click the scroll button in the Files of type: window and then select All Files. Double click the Errors.hlp icon.
4. Back in the Create Shortcut window, click the Next button.
5. In the Select Program Folder window, click the New Folder button and type **`Microsoft C version 8.00c`**; then press the Enter key.
6. In the Select a Title for the Program window, change "Errors.hlp" to "Errors", and then click the Finish button.

7. Back on the Taskbar Properties sheet, click the Add button again.
8. Browse the MSVC\Help folder again from the Create Shortcut window and double click Mscxx.hlp. Back on the Create Shortcut window click Next. This time do not click New Folder in the Select Program Folder window. Instead, select the Microsoft C version 8.00c folder and click Next.
9. In the Select a Title for the Program window, change **`Mscxx.hlp`** to **`Language Reference`**, and then click the Finish button.

Repeat these steps for Readme.hlp and Tools.hlp.

Chapter 3 Writing a C Application

This chapter contains information needed to write C applications for the Series 90 PLC. It includes details on declaring parameters, accessing CPU reference memory, and using standard library routines.

Chapter 3 contains the following sections:

The C source code used to build C applications may be created using the editor of your choice, provided that the output from your editor is compatible with the Microsoft C compiler. (Word processors are not recommended for editing C source code.)

It is also recommended that each C application be developed in its own subdirectory. One approach would be to create a **\APPS** subdirectory off the hard disk root directory. As each application is developed, a new subdirectory under **\APPS** is created; for example, **\APPS\RAMP**, **\APPS\LIMIT**, **\APPS\PRESS**, . . . etc.

Caution

C applications should not be developed in the root directory of a hard disk. When building for execution in the MS-DOS environment, the C development software will create, if necessary, and write files to the DOS directory under the current default directory.

Note

Application names must conform to Logicmaster 90 naming conventions (7 characters long, first character must be a letter).

Section 1: Series 90-70 C Block and C FBK Structure

A C block or C FBK can be invoked in one of three ways:

1. As a subblock of main
2. As an I/O or timed interrupt block
3. As a subblock of an interrupt block

Blocks invoked as a subblock of main or as a subblock of an interrupt block may have up to seven input/output parameter pairs. Blocks invoked as an I/O or timed interrupt cannot have parameters. C blocks may not be used in the same program as a subblock of main and as an interrupt block; however, C FBKs may be used in both types of blocks in the same program. Calls to blocks and block interrupt declarations are denoted with the word **`external`**, since the block is developed externally from Logicmaster 90 software and then imported into the application program folder. Shown below are two ladder logic rungs containing external block calls with zero and three parameter pairs, respectively:

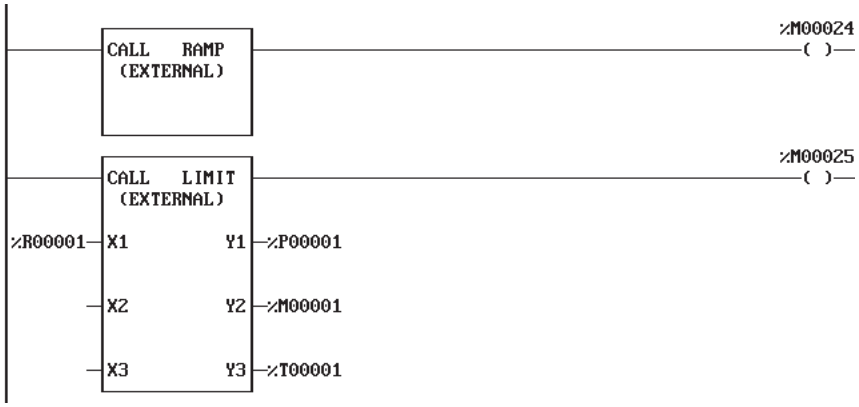


Figure 3-1. Ladder Logic Calls to 90-70 C Blocks

Note

The OK output is present regardless of whether the block has parameters and is set based on the function result (either OK or ERROR). Appropriate definitions of OK and ERROR are given in the **PLCC9070.H** file.

Each block is written as a separate application. The .EXE file produced by the build process of a block must be added to the Logicmaster 90 Librarian and may be imported into a program folder from the Librarian.

The main function in each block must always be called **main**. Any legal C declaration and code may be used in a C block. C FBKs, however, cannot have any declaration or code references to library routines, since calls to library routines will generate unresolved external reference errors during linking. If an unresolved external (either explicit or implicit) cannot be eliminated, then the C FBK must be rebuilt as a C block.

The file **PLCC9070.H**, installed as part of the C Programmer's Toolkit, should be included in the block source file(s). **PLCC9070.H** contains declarations, definitions, and macros used in writing blocks.

The following example shows the basic components of a block with no parameters:

```
#include "plcc9070.h"      /* Series 90-70 interface file */

main ()
{
  /* value of function block OK output determined by returned value */
  return(OK);
}
```

Variable Declarations

Global and static variables may be used in a block. The space allocated for them is taken from the 64,000 byte maximum space allowed for each block. Local, or automatic, variables are allocated on the stack. The Series 90-70 PLC guarantees that a minimum of 1092 bytes is available on the stack prior to calling a block; therefore, the total stack space used for local variables and for saving the return address when making calls to other functions should not be allowed to exceed 1092 bytes for blocks.

Stack Checking

If the PLC ever detects that there is not enough space available on the stack when executing a CALL to a block, an application fault will be logged in the PLC fault table and the block will not be called. If you exceed the size for the stack in a C block by making multiple or recursive calls, the stack checking for the C block will detect this and exit without finishing. If you exceed the size for the stack in a C FBK, there is no stack checking so a page fault may occur or the PLC may get invalid data.

Parameter Declarations

Up to seven input/output parameter pairs may be specified for a block (with the exception of blocks invoked as an I/O or timed interrupt block, which cannot have parameters). Any legal Logicmaster 90 reference type may be specified. However, data flow, boolean flow, and indirect references cannot be used as parameters for blocks. In the block source, parameters are declared as pointers to objects of the desired type. All parameters, including constants, are passed as pointers.

The order of the parameter declarations must match the CALL instruction parameter order, with the input parameters followed by the output parameters. The declaration code shown below could be used for a block that has two input/output parameter pairs:

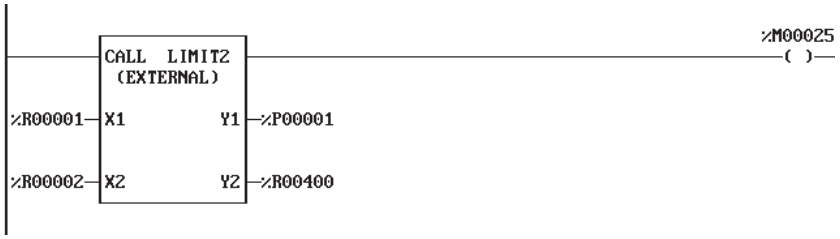


Figure 3-2. Matching Parameters Between Call and External Block

```
main(in1, in2, out1, out2)

    /* X1 - pointer to a single integer */
    int *in1;

    /* X2 - pointer to a 256 element array of integers */
    int  in2[256];

    /* Y1 - pointer to a structure containing an integer */
    /*      and a floating point variable */
    struct {
          int   a;
          float b;
       } *out1;

    /* Y2 - pointer to an unsigned integer */
    word *out2;
```

It is not required that all of the CALL instruction parameters be used. If a CALL instruction parameter is not used, a NULL pointer is passed as that parameter's value. The parameter must still be declared, so that subsequent parameters are lined up correctly with their pointers. In the following example, a NULL pointer is passed in for the second and third input parameters.

.

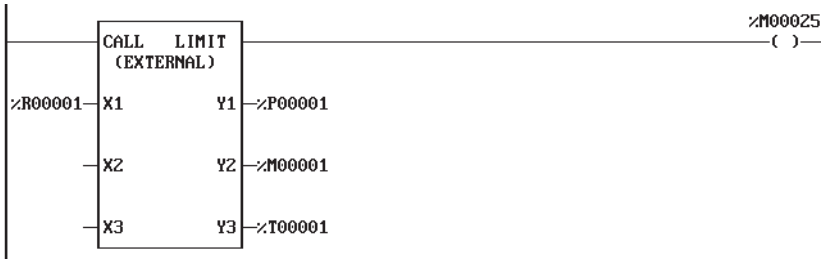


Figure 3-3. Reserving Space For Unused Parameters to an External Block

```
main(x1, x2, x3, y1, y2, y3)
    int *x1;
    int *x2;           /* placeholder for unused parameter */
    int *x3;           /* placeholder for unused parameter */
    int *y1;
    int *y2;
    int *y3;
{
  *y1 = *x1;           /* Copy value at x1 to y1 */
  *y2 = *x1 * 2;       /* copy twice the value at x1 to y2 */
  *y3 = *x1 * 3;       /* Copy three times the value at x1 to y3 */
  return(OK)
}
```

Parameter Pointer Validation

The ladder logic program provides the variables that are passed into the block's **main( )**. Since **main( )** cannot be guaranteed that all necessary parameters to main were provided, **main( )** must check for these parameters. If you modify the block shown on the previous page to illustrate these checks, the modified block will appear as:

```
main(x1, x2, x3, y1, y2, y3)
    int *x1;
    int *x2;           /* placeholder for unused parameter */
    int *x3;           /* placeholder for unused parameter */
    int *y1;
    int *y2;
    int *y3;
{
  /* Ensure that required parameters were provided by caller */
  /* No need to check x2 and x3 since they are not used.     */
     if ((x1==NULL)||(y1==NULL)||(y2==NULL)||(y3==NULL))
         return(ERROR);

  /* Required parameters are present. */
       *y1 = *x1;      /* Copy value at x1 to y1 */
       *y2 = *x1 * 2;  /* copy twice the value at x1 to y2 */
       *y3 = *x1 * 3;  /* Copy three times the value at x1 to y3 */
       return(OK)
}
```

Section 2: Series 90-70 Standalone C Program Structure

Standalone C programs are scheduled to run in the PLC through Logicmaster 90-70 software's Program Specification screen. This scheduling specifies when the program is run, how the program is run, what the I/O specifications are for the program, how the I/O specification copies are handled, and what the stack size is for the program. A standalone C program may be scheduled only once in a folder. For additional information, refer to the *Logicmaster 90-70 Programming Software User's Manual.*

Each standalone C program is written as a separate application. The files produced by the build process of a standalone C program are imported into a program folder. This folder must be created before the standalone C program is built. The main function in each standalone C program must always be called **`main`**. Any legal C declaration and code may be used in a standalone C program.

The file **`PLCC9070.H`**, installed as part of the C Programmer's Toolkit, should be included in the source file(s). **`PLCC9070.H`** contains declarations, definitions, and macros used in writing standalone C programs.

The following example shows the basic components for a standalone C program with no I/O parameters:

```
#include "plcc9070.h"      /* Series 90-70 interface file */

main ()
{

  return(OK);
}
```

Note

If the return value from a standalone C program is not OK then the output specifications will not be updated. This includes runtime error exits and PLC error exits. PLCC9070.H provides definitions for OK and error return values.

Variable Declarations

Global and static variables may be used in a standalone C program. Global and static variables are allocated from program space. Local, or automatic, variables are allocated on the stack. The Series 90-70 PLC guarantees that the stack space configured is available; therefore, the total stack space used for local variables and for saving the return address when making calls to other functions should not be allowed to exceed the stack space configured.

Standalone C Program Stack

The stack size for a standalone C program can range between 1K and 64K and is allocated in 1K increments (where 1K is 1024 bytes). If the standalone C program

exceeds the stack limit during execution the stack checking function inside the standalone C program will detect this problem and exit with a runtime error.

Standalone C Program I/O Specifications

Up to eight input and eight output data areas can be declared for standalone C programs. These are declared with the macros **IN1_*type***, **IN2_*type***, ..., etc. and **OUT1_*type***, **OUT2_*type***, etc. where ***type*** is **B**, **W**, **D**, or **F** for byte, word, double word, and float access respectively. The macro calls have two parameters:

```
IN1_B(arrayname, arraylength);
```

where ***arrayname*** is the name of the array to allocate and ***arraylength*** is the length of the array in bytes, words, or Dwords, depending on which is used. The macros will allocate an array of the type specified. For example, **IN1_F(x1, 10)** will cause an array **float x1[10]** to be created. These arrays are included in the size of the standalone C program. Unlike the parameter passing with blocks, the I/O specifications can have a different number of inputs and outputs. The I/O specifications can even be discontinuous, that is, **IN1_B**, and **IN3_W** can be used without **IN2_B** or **IN2_W** The following is an example using I/O specifications:

```
#define “plcc9070.h

IN1_B (x1, 10)
IN2_W (x2, 20)
OUT1_B (y1, 9)
OUT2_W (y2, 8)

main()
{
 y1[5] = x1[0] + x2[5];
 y2[7] = x2[19] * x1[9];
}
```

Note

All I/O specification macros must appear in all uppercase letters because C is a case-sensitive language, and the definition of the macro uses all uppercase for the macro name.

The I/O specifications are also declared on the Logicmaster 90-70 Program Specification screen or the the Windows-based programming software Header window. The I/O specifications listed on the screen must match the I/O specifications declared in the C program if the program is to operate correctly. For example, if the program declares IN1_W(x1, 3), IN8_B(x8, 255), and OUT5_W(y5, 12) then the screen must have a 6 byte space declared at the first input specification area, a 255 byte space declared at the eighth (or last) input specification area and a 24 byte specification area for the fifth output specification area. The other I/O specification areas **must** be empty. The user must ensure that the program's declaration and those of the Windows-based programming software or of Logicmaster match. Both programming packages will provide default values for the I/O specification areas the first time a standalone C program is imported.

Section 3: C Subroutine Block and C Main Program Structure (Series 90-30 Only)

For 90-30 PLCs, C blocks can be invoked in one of two ways:

1. As the main program
2. As a subblock of main

Calls to blocks and are denoted with **EXT CALL**, since the block is developed externally from the Windows-based programming software and then imported into the application program folder. The following the Windows-based programming software screen capture is of a 90-30 subroutine block:

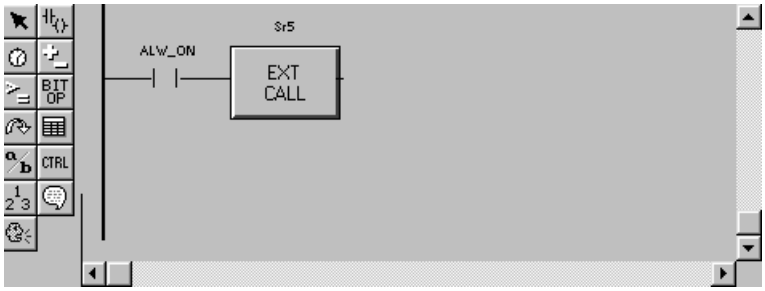


Figure 3-4. Ladder Logic Call to 90-30 C Block

Note

The ENO output of the CALL External block is always set to true.

Each block is written as a separate application. The .EXE file produced by the build process of a block may be imported into an Equipment Folder from the Windows-based programming software. (For information about adding C programs and blocks to your Windows-based programming software Equipment Folder, refer to page 3-97 and following.)

The main function in each block must always be called **main**. Any legal C declaration and code may be used in a C block.

The file **PLCC9030.H**, installed as part of the C Programmer's Toolkit, should be included in the block source file(s). **PLCC9030.H** contains declarations, definitions, and macros used in writing blocks.

The following example shows the basic components of a block:

```
#include "plcc9030.h"      /* Series 90-30 interface file */

EXE_stack_size=2048;

main ()
{
}
```

Note

The return value of a 90-30 C subroutine block is an unconditional return: no value is passed and power flow/ENO is always passed. Also note that the stack size must be explicitly stated though the **EXE_stack_size** statement.

Variable Declarations for 90-30 PLCs

Global and static variables may be used in a block. The space allocated for them is taken from the 81,920 byte maximum space allowed for each block. Local, or automatic, variables are allocated on the stack.

Stack Checking

If the PLC ever detects that there is not enough space available on the stack when executing a CALL to a block, an application fault will be logged in the PLC fault table and the block will not be called. If you exceed the size for the stack in a C block by making multiple or recursive calls, the stack checking for the C block will detect this and exit without finishing.

EXE_stack_size

A required global variable that specifies the size of the stack for the block or Main program. Stack size can be a maximum of 64 kilobytes; for most blocks the recommended size is 2048.

Section 4: PLC Reference Memory Access

Series 90 PLC reference address and diagnostic memory may be read and written directly via macros defined in **PLCC9070.H** or **PLCC9030.H**. These macros consist of a string of capitalized letters which indicate the Series 90 reference type (and in some cases, the type of operation to be performed) followed by the reference offset in parentheses. In general, PLC reference memories may be accessed using the macros in **PLCC9070.H** or **PLCC9030.H** as bits, bytes (8 bit values), words (16 bit values), double words (32 bit values), or single precision floating point numbers (32 bits).

The complete set of Series 90 reference type designators are as follows:

Reference Type	Description
%I	Discrete input references
%Q	Discrete output references
%M	Discrete internal references
%T	Discrete temporary references
%G	Discrete global data references
%GA - %GE (90-70 only)	Discrete global data references
%S	Discrete system references
%SA	Discrete maskable fault references
%SB	Discrete non-maskable fault references
%SC	Discrete fault summary references
%AI	Analog input registers
%AQ	Analog output registers
%R	System register references
†%P (90-70 only)	Program register references
†%L (90-70 only)	Local register references

† not valid for standalone C programs

How to Format a PLC Reference Access Macro

The table shown below gives the modifiers used with the PLC reference macros (listed in Appendix C). The format for usage of these macros is as follows:

The letter of reference type followed by one of the modifiers followed by a parenthetical number for the address you wish to access; e.g.,

```
RI(1)=3;     This assigns the integer value 3 to %R00001
RW(2)=0x55AA;This assigns the word value 55AAh to %R00002
MB(1)=0xAB;  This assigns the byte value AB to the 8 bits beginning with
             %M00001 (%M00001 is the Least Significant Bit.)
```

The data type modifiers are as follows:

Modifier	Description
B	Unsigned byte reference (8 bits, 0 −> 255)
W	Word reference (16 bits, 0 −> 65535)
I	Integer reference (signed 16 bits, −32768 −> 32767)
D	Double precision integer reference (signed 32 bits, −2147483648 −> 2147483647)
F	Floating point reference (32 bit IEEE floating point format)

Certain combinations of reference type designators and data type modifiers are not supported. Those combinations which are supported have macros defined in the

PLCC9070.H or **PLCC9030.H** file. Refer to Appendix C for the complete set of macros provided in **PLCC9070.H** or **PLCC9030.H**.

Macros which permit access to reference memories as bits are slightly different from those macros which access the same reference memories as bytes, words, double words, and/or floating point numbers. Bit access macros, byte access macros, word/integer access macros, word-memories-as-bytes access macros, and double word/floating point access macros are described on the following pages of this chapter.

Note

When accessing PLC reference memory, be sure that macros are in all uppercase letters as C is a case-sensitive language.

Restrictions on Macro Use (Series 90-70 Only)

Note

Restrictions on macro use do not apply if building standalone C programs. Standalone C programs require Version 6.00 or later CPU firmware. Version 6.00 is not available on the hardware platforms described in this section.

Some older versions of Series 90-70 CPU hardware do **not** support accessing any of the PLC bit memories (%I, %Q, %T, %M, %G, %GA-%GE, %S, %SA, %SB, %SC) as words, integers, or double word values. Any attempt to access the bit-oriented memories as words, integers, and/or double word values **will** result in incorrect data being read or written.

Those versions of Series 90-70 CPU hardware which **do** support accessing bit-oriented memory as words, integers, and/or double words have catalog numbers IC697CPU731P and later, IC697CPU771M and later, **and any** IC697CPU732, IC697CPU772, IC697CPU781, IC697CPU782, IC697CPM914, IC697CPM915, IC697CPM924, or IC697CPM925.

Caution

Although catalog numbers IC697CPU731P and later and IC697CPU771M and later do support accessing the PLC bit-oriented memories as words, integers, and/or double words, this capability applies only to those 90-70 CPUs which have a single suffix letter in the catalog number.

For example, the following catalog numbers all support accessing the bit-oriented memories as words, integers, and/or double words:

```
IC697CPU731P
IC697CPU731R
IC697CPU731S
IC697CPU771M
IC697CPU771P
IC697CPU771S
```

However, the following catalog numbers do not support accessing the bit-oriented memories as anything other than bits or bytes.

IC697CPU731SU
IC697CPU731SX
IC697CPU771SU
IC697CPU771SX

The C Programmer's Toolkit provides macros which access the bit-oriented memories as words, integers, and/or double words. To prevent accidental use of these macros in a program which may operate on an older Series 90-70 CPU platform (which does not support word, integer, or double word reads/writes to the bit-oriented memories), these C macros are not available as a standard, default condition.

If the target CPU platform for execution of your C application is **known** to be one which supports accessing bit-memories as words/integers/double words, then use of these C macros may be enabled by placing

```
#define  HARDWARE_SUPPORTS_WORD_CACHE
```

as the **first** line of your C source module(s) (before the **`# include PLCC9070H`** line). If the target platform supports accessing the bit-oriented memories as words/integers/double words, then use of the provided C macros is encouraged to ease programming effort.

If the target platform for execution of your C application is **not known**, please restrict your C source code to using ONLY the bit and byte macros for accessing the bit-oriented memories.

Caution

The restrictions on accessing the Series 90-70 PLC CPU bit-oriented memories as anything but bits (using the macros provided in the Toolkit) or as bytes (or chars) on certain CPU hardware platforms exist not only for the C macros provided in the Series 90-70 C Programmer's Toolkit, but also for any access to these same PLC CPU memories. Do not attempt to declare and use your own pointer or macros, which access into the Series 90-70 PLC CPU bit-oriented memories as words, integers, double words, or any other type or structure unless the CPU platform is *known* to support that access.

Bit Macros

There are three bit macros defined for each reference memory type:

Macro	Description
BIT_TST_X	Tests the specified bit
BIT_SET_ X	Sets the specified bit
BIT_CLR_X	Clears the specified bit

References in a C application to %I would use **`BIT_TST_I( )`**, **`BIT_CLR_I( )`**, or **`BIT_SET_I( )`**. The macro name indicates that %I reference memory is to be operated on and the operation is either tested (TST), cleared (CLR), or set (SET). The value contained in parentheses is the reference number of the item to be tested, cleared, or set

(for example, 120 for %I120). The bit set and bit clear macros are separate C application source statements.

Note

The bit test macros return a boolean value contained in a byte. The accessed bit is right justified (least significant bit) in the byte, that is, each of the bit test macros will evaluate to 0 if the bit is OFF or 1 if the bit is ON.

If a C application should set %Q137, %M29, and %T99 if %I120 is ON and should clear %Q137, %M29, and %T99 if %I120 is OFF:

Series 90-70 Example:

```
#include "plcc9070.h"

main() {
    if (BIT_TST_I(120)) {
       BIT_SET_Q(137);
       BIT_SET_M(29);
       BIT_SET_T(99);
    } else {
       BIT_CLR_Q(137);

       BIT_CLR_M(29);
       BIT_CLR_T(99);
    }
    return(OK);
}
```

Series 90-30 Example:

```
#include "plcc9030.h"

EXE_stack_size=2048;

main() {
    if (BIT_TST_I(120)) {
       BIT_SET_Q(137);
       BIT_SET_M(29);
       BIT_SET_T(99);
    } else {
       BIT_CLR_Q(137);

       BIT_CLR_M(29);
       BIT_CLR_T(99);
    }
    return(OK);
}
```

The bit macros for accessing word-oriented PLC memories (%R, %P, %L, %AI, and %AQ) as bits are similar to the above description except that these macros require one additional parameter, namely, the position within the word of the bit being accessed. The three forms of bit macros for accessing word-oriented PLC memory are BIT_SET_, BIT_CLR_, and BIT_TST_ (to specify the type of operation) followed by R, P, L, AI, or AQ (to specify the PLC reference memory to be used). There are two required parameters to these macros:

1. The word in the reference memory to access (1 to highest reference available in the specified PLC memory).
2. The bit in the selected word to use (bit numbers 1 to 16, with bit 1 being the least significant or rightmost bit).

To illustrate the bit macros for word-oriented memory, consider the following section of a C application:

```
if (BIT_TST_R(135, 6) )
    BIT_SET_P(13, 4);
else
    BIT_CLR_AI(2,1);
```

This portion of a C application checks the sixth bit in %R135. If the bit is on (1), then the fourth bit in %P13 is to be set ON (1); otherwise, the first bit in %AI2 is to be set OFF (0).

Note

The %L and %P macros are not available to standalone C programs or 90-30 C programs.

As with the reference memory, there are three bit macros defined for use with I/O specifications:

Macro	Description
BIT_TST(name, x)	Tests the specified bit
BIT_SET(name, x)	Sets the specified bit
BIT_CLR(name, x	Clears the specified bit

References in a Series 90-70 standalone C program would use **BIT_TST( )**, **BIT_CLR( )**, or **BIT_SET( )**. The macro name indicates that the operation is either test (TST), clear (CLR), or set (SET). The first value contained in the parentheses is the name of the I/O specification. The second value contained in parentheses is the number to be tested, cleared, or set (for example, 120 for the 120th bit in the array). The bit set and bit clear macros are separate standalone C program source statements.

This standalone C program will set the 137th bit of the first output specification, the 29th of the second, and the 99th of the third output specification if the 120th bit of the first input specification is ON and will clear them if the input specification bit is OFF.

Series 90-70 Example:

```
#include “plcc9070.h”

IN1_B(in1,30);
OUT_B(ot1,30);

main() {
    if (BIT_TST(in1,120)) {
       BIT_SET(ot1,137);
       BIT_SET(ot2,29);
       BIT_SET(ot3,99);
    } else {
       BIT_CLR(ot1,137);

       BIT_CLR(ot2,29);
       BIT_CLR(ot3,99);
    }
    return(OK);
}
```

Note

The "BIT_" macros used to access bits in word-oriented memories use a 1 to 16 bit numbering scheme, with bit 1 being the least significant bit and bit 16 being the most significant bit.

Byte Macros

Macros are provided to access the PLC bit memories as bytes. These macros are IB(x), QB(x), MB(x), TB(x), GB(x), GAB(x), GBB(x), GCB(x), GDB(x), GEB(x), SB(x), SAB(x), SBB(x), and SCB(x). The parameter x in each of these macros should be replaced with the reference address of a bit which is contained in the byte; for example, if the byte containing %M123 is needed, use MB(123). The byte access macros may appear on either the left-hand or right-hand side of a C statement.

The example that follows will set the byte starting at %T9 and ending at %T16 equal to the byte starting at %Q65 and ending at %Q72.

Series 90-70 Example:

```
#include "plcc9070.h"

main() {
    TB(13) = QB(72);
    .
    .
    .

    return(OK);
}
```

Series 90-30 Example:

```
#include "plcc9030.h"

EXE_stack_size=2048;

main() {
    TB(13) = QB(72);
    .
    .
    .

    return(OK);
}
```

Accessing bytes from word-oriented memories (%R, %P, %L, %AQ, and %AI) requires an additional parameter to indicate which byte is to be read or written. The symbols **HIBYTE** and **LOBYTE** are defined in **PLCC9070.H** or **PLCC9030.H** for this purpose. For example, your C application requires that the low byte of %R5 be read into a C application local variable and then copied into the high byte of %R17:

Series 90-70 Example:

```
#include “plcc9070.h”

main() {
    byte      abytvar;

    abytvar = RB(5,LOBYTE);           /* read low byte of %R5 */
    RB(17,HIBYTE) = abytvar;          /* write high byte of %R17 */
    .
    .
    .

    return(OK);
}
```

Series 90-30 Example:

```
#include “plcc9030.h”

EXE_stack_size=2048;

main() {
    byte      abytvar;

    abytvar = RB(5,LOBYTE);           /* read low byte of %R5 */
    RB(17,HIBYTE) = abytvar;          /* write high byte of %R17 */
    .
    .
    .

    return(OK);
}
```

Note

The %L and %P macros are not available to standalone C programs nor on 90-30 blocks and programs. Also, %GA–%GE memory types are not supported on 90-30 PLCs.

Integer/Word Macros

All PLC reference memories may be accessed as 16-bit 2's complement integers (ints) or as 16-bit unsigned integers (words).† As an example, a C application needs to read %R123 as an unsigned 16-bit integer and write %P13 as a 2's complement 16-bit integer and store the values in separate local C source variables:

Series 90-70 Example:

```
#define  HARDWARE_SUPPORTS_WORD_CACHE
#include "plcc9070.h"

main() {
    word      word_val;
    int       int_val = -133;

    word_val = RW(123);/* read %R123 as a word */
    PI(13) = int_val;  /* copy 2's complement integer to %P00013 */
    .
    .
    .

    return(OK);
}
```

Series 90-30 Example:

```
#include "plcc9030.h"

EXE_stack_size=2048;

main() {
    word      word_val;
    int       int_val = -133;

    word_val = RW(123);/* read %R123 as a word */
    RI(13) = int_val;  /* copy 2's complement integer to %R0013 */
    .
    .
    .

}
```

† There are certain restrictions on the use of the integer, word, and double word macros. See "Restrictions on Macro Use."

Note

The %L and %P macros are not available to standalone C programs or to any 90-30 program or block.

Double Word/Floating Point Macros

All PLC reference memories may be accessed as 32-bit unsigned integers (dwords), but only the word-oriented memories (%R, %P, %L, %AQ, and %AI) may be accessed as 32-bit floating point numbers (float).† As an example, a C application needs to read %R77 as a double word and write %P6 as a floating point value:

Series 90-70 Example

```
#define  HARDWARE_SUPPORTS_WORD_CACHE
#include "plcc9070.h"

main() {
    dword     dword_val;
    float     fp_val = 15.56;

    dword_val = RD(77);  /* read %R77 as a double word */
    PF(6) = fp_val;      /* write %P6 as single precision floating point */
    .
    .
    .

    return(OK);
}
```

Series 90-30 Example

```
#include "plcc9030.h"

EXE_stack_size=2048;

main() {
    dword     dword_val;
    float     fp_val = 15.56;

    dword_val = RD(77);  /* read %R77 as a double word */
    RF(6) = fp_val;      /* write %R6 as single precision floating point */
    .
    .
    .

}
```

Note

The %L and %P macros are not available to standalone C programs. Also, %L and %P memory does not exist in the 90-30 CPU; therefore, %L and %P macros are non-applicable to the 90-30 CPUs. In addition, of the 90-30 CPUs, only CPU352 is capable of hardware-based floating point operations.

† There are certain restrictions on the use of the integer, word, and double word macros. See "Restrictions on Macro Use."

Reference Memory Size Macros

Macros are defined in **PLCC9070.H** or **PLCC9030.H** for determining the size of each memory type. These macros are in the form *X*_SIZE, where *X* is the memory type letter I, Q, M, T, G, S, R, AI, AQ, P, or L. Each of these size macros returns an unsigned integer value equal to the highest reference available in the specified reference memory. If the last available reference in the %I table is %I1280, when a C application uses the I_SIZE macro, the value 1280 will be returned.

Caution

The reference memory size macros should be used to determine the size of the memory types written within a C application. Reads and writes outside of the configured range can result in incorrect data or PLC CPU failure.

For example, a C application is created that takes an index as a single input parameter into the register table. The application is designed to index into the register table using the input parameter and copy the located value to the single output location (MOVE from source array registers [input parameter] to output parameter). This C application is to be designed so that it may be run on any Series 90-70 PLC, regardless of differing register memory table sizes:

```
Series 90-70 Example
#include “plcc9070.h”

main(word *X1, int *Y1) {
    if ((X1 != NULL)&&
       (Y1 != NULL)) {
       if (*X1 > R_SIZE) {
          /* Index into registers is too large! */
          return(ERROR);
       } else {
          /* Index into registers and copy value to output parameter*/
           *Y1 = RI(*X1);
       }

       return(OK);
    }
    else return (ERROR);
}
```

Note

The %L and %P macros are not available to standalone C programs. The example shown above is a Series 90-70 example. The %L and %P macros are not available on a Series 90-30 program or block.

Transition, Alarm, and Fault Macros

Transition, alarm, and fault bits associated with reference memory can also be referenced. In addition, the special system %S contacts **FST_SCN**, **T_10MS**, **T_100MS**, **T_SEC, T_MIN**, **ALW_ON**, **ALW_OFF**, **SY_FULL**, **IO_FULL**, **FST_EXE** (see Note), and **LST_SCN** (see Note) are supported for C blocks and FBKs. Series 90-70 standalone C programs support **FST_SCN**, **ALW_ON**, **ALW_OFF**, **SY_FULL**, and **IO_FULL**, although the **FST_SCN** here is high (1) on the first execution after a STOP−> RUN transition or a RUN-MODE-STORE.

Note

FST_EXE is supported for Series 90-70 C blocks only. **LST_SCN** is supported for Series 90-30 C main and C subroutine blocks only. All others, i.e., **FST_SCN**, **T_10MS**, **T_100MS**, **T_SEC, T_MIN**, **ALW_ON**, **ALW_OFF**, **SY_FULL**, and **IO_FULL** are supported for both target databases.

The following macros are available for a Series 90-70 folder:

Macros (Series 90-70 Only)

```
/* Used with macros which access DIAGNOSTIC memory(s) */
#define HI_ALARM_MSK    0x02
#define LO_ALARM_MSK    0x01
#define OVERRANGE_MSK   (0x40 | 0x8)
#define UNDERRANGE_MSK  0x20

#define BIT_TST_I_DIAG(x)  ((((*i_diag_mem)[(x-1)>>3])&(bit_mask
   [(x-1)&7])) != 0)
#define BIT_TST_Q_DIAG(x)  ((((*q_diag_mem)[(x-1)>>3])&(bit_mask
   [(x-1)&7])) != 0)
#define IB_DIAG(x)     ((*i_diag_mem)[(x-1)>>3])
#define QB_DIAG(x)     ((*q_diag_mem)[(x-1)>>3])
#define AIB_DIAG(x)    ((*ai_diag_mem)[x-1])
#define AQB_DIAG(x)    ((*aq_diag_mem)[x-1])
#define AI_HIALRM(x)((((*ai_diag_mem)[x-1])&(HI_ALARM_MSK))>>1)
#define AI_LOALRM(x)(((*ai_diag_mem)[x-1])&(LO_ALARM_MSK))
#define AIB_FAULT(x)(((*ai_diag_mem)[x-1])&(~(HI_ALARM_MSK
   |LO_ALARM_MSK)))
#define AQB_FAULT(x)(((*aq_diag_mem)[x-1])&(~(HI_ALARM_MSK
   |LO_ALARM_MSK)))
#define AI_OVERRANGE(x)((((((*ai_diag_mem)[x-1])&(OVERRANGE_MSK))
   >>3)|((((*ai_diag_mem)[x-1])&(OVERRANGE_MSK))>>6 ) & 1)
#define AI_UNDERRANGE(x)   ((((*ai_diag_mem)[x-1])&(UNDERRANGE_MSK))
   >>5)


/* DEFINEs and STRUCTUREs used by the PLC */
/*      RACK/SLOT/BLOCK fault macros      */
struct  rack_slot_refs_rec {
    unsigned    rack_summary:1;
    unsigned    rack_failure:1;
    unsigned    slot_faults:14;         /* Slots 0 through 9
   are used */
};
```

```
#define NUM_BYTES_BUS_REFS          4   /* 2 buses per slot,
   4 slots per byte */
#define NUM_BYTES_PER_MODULE_REFS   8   /* 32 modules per bus,
   2 buses per slot */
#define NUM_BYTES_PER_FIP_DROP_REFS 32  /* 32 bytes per FIP drop */
#define MAX_VME_SLOTS              10   /* 10 slots per VME
   rack – max */
#define MAX_VME_RACKS               8   /* 8 VME racks – max */



struct rack_reference_rec {
    struct rack_slot_refs_rec rack_slot_refs;
    unsigned char             bus_refs[NUM_BYTES_BUS_REFS];
    unsigned char
mod_refs[MAX_VME_SLOTS][NUM_BYTES_PER_MODULE_REFS];
};

struct fip_rack_reference_rec{
   unsigned char
drop_refs[MAX_VME_SLOTS][NUM_BYTES_PER_FIP_DROP_REFS];
};

struct rack_reference_with_fip_rec{
   struct rack_reference_rec rs_rpt_refs[MAX_VME_RACKS];
   struct fip_rack_reference_rec fip_drop_refs[MAX_VME_RACKS];
};

extern struct rack_reference_with_fip_rec (*rsb_mem);
#define rs_memory ((*rsb_mem).rs_rpt_refs)
#define fip_memory ((*rsb_mem).fip_drop_refs)

/* Macros for accessing RACK/SLOT/BLOCK fault information */
#define RACKX(r)
(rs_memory[r].rack_slot_refs.rack_summary)
#define SLOTX(r,s)
(((rs_memory[r].rack_slot_refs.slot_faults)&(bit_mask[s]))>>(s))
#define BLOCKX(r,s,b,sba)
(((rs_memory[r].mod_refs[s][(sba>>3)+(4*(int)(b==2))])&(bit_mask
   [sba&7]))>>(sba&7))
#define RSMB(x)                 (((unsigned int *)rs_memory)[x])

/* Macros for accessing FIP fault information */
/* NOTE: The FIPX macro is only valid for 90–70 CPUs */
/* with firmware release 6.00 and later */
#define FIPX(r,s,sba)
((((fip_memory[r].drop_refs[s][sba>>3])&(bit_mask[sba&7])))>>(sba&7))
```

Refer to Chapter 3, Section 1 of the *Series 90-70 Programmable Controller Reference Manual* (GFK-0265) for more information on fault references.

Section 5: Standard Library Routines

Appendix A contains a complete list of the standard C library routines supported by C blocks and standalone C programs (please note that C FBKs do not support any library routines). The routines work as documented by the compiler manufacturer, with the exception of the **printf( )** and **sprintf( )** routines. When executing under MS-DOS, the code used to perform **printf( )** and **sprintf( )** is Microsoft's. When executing in the Series 90 PLC, the code used to perform these two functions is in the Series 90 CPU firmware.

printf( ) and sprintf( ) — Series 90-70 Only

The implementations of **printf( )** and **sprintf( )** available in the Series 90-70 PLC support a subset of the ANSI-defined interface to these functions. In the Series 90-70 PLC, the following **printf( )** / **sprintf( )** formats are supported:

```
%[flags][width].[precision][{F|N|h|l|L}]type
```

Table 3-1. printf( ) Formats Supported in the PLC

Field Name	Field Values / Description
flags	– minus flag + plus flag ' ' blank flag # pound flag
width	can be constant or asterisk (*) to indicate width is in argument list
precision	can be constant or asterisk (*) to indicate precision is in argument list
F|N|h|l|L	only support l
type	o u x X d i c s - for all models
	e f g E G - for models with a math coprocessor only

The **printf( )** routine is used to queue messages in a character buffer to be transmitted from the Series 90-70 CPU serial port. Multiple messages may be queued, with a limit of 2048 characters pending at any one time. The **printf( )** function returns the number of characters placed into the character print queue. If **printf( )** is unable to queue an entire message because of character buffer overflow, the portion of the message which will fit into the queue is inserted and the remaining trailing characters are truncated. One of the built-in functions, **PLCC_chars_in_printf_q( )**, may be used to determine the number of characters currently in the buffer.

Note

Although the transmission of a string out the CPUs serial port is performed in the background of the PLC execution, the formatting of a string by **printf( )** before it is placed into the message queue is performed in its entirety at the time **printf( )** is called. On a Model 731 90-70 CPU, which has a default (200msec) watchdog timeout value, formatting a string of 2048 characters will cause the watchdog timer to expire and the PLC to go to **STOP/HALT** mode. Refer to appendix E, *Series 90-70 CPU Execution Time for printf( )*, for example timing for **printf( )**.

Message queuing is enabled when the CPU serial port is configured for **message generation** mode. When this mode is selected, the SNP protocol is active on the serial port only when the CPU is in **STOP** mode. In **RUN** mode, use of the serial port is reserved for the **printf( )** function.

When a **printf( )** is attempted in the PLC and the CPU is not configured for **message generation** mode, a fault will be logged in the PLC fault table. The fault will be displayed as:

EXE block runtime error: *<block name>* attempted print

where *<block name>* is the name given the block when it was imported into the Logicmaster 90 folder, or

Program runtime error: *<prog name>* attempted print

where *<prog name>* is the name of the standalone C program. All runtime errors generated out of a standalone program will be prefixed with "**Program runtime error**."

Note

When configured for **message generation** mode, the serial port will be used for **printf( )** output **only** when the PLC is in **RUN** mode. In **STOP** mode, the PLC serial port will revert back to SNP communications. On the transition from **RUN** to **STOP**, the PLC will complete transmitting any **printf( )** output to the serial port prior to reverting back to SNP on the serial port. On the transition from **STOP** to **RUN**, SNP communication is dropped immediately.

Changing the PLC Configuration to Enable printf( ) Output

In order to use **printf( )** in the Series 90-70 PLC, the configuration stored to the CPU must indicate that the serial port is to be used for **printf( )** output rather than for SNP. To change the CPU's configuration, use the Logicmaster 90 configuration software package as follows:

1. Once in the configuration software, select I/O configuration.
2. While on the Rack 0 configuration display, cursor over to the CPU module and press the Zoom softkey.

3. The following display should appear:

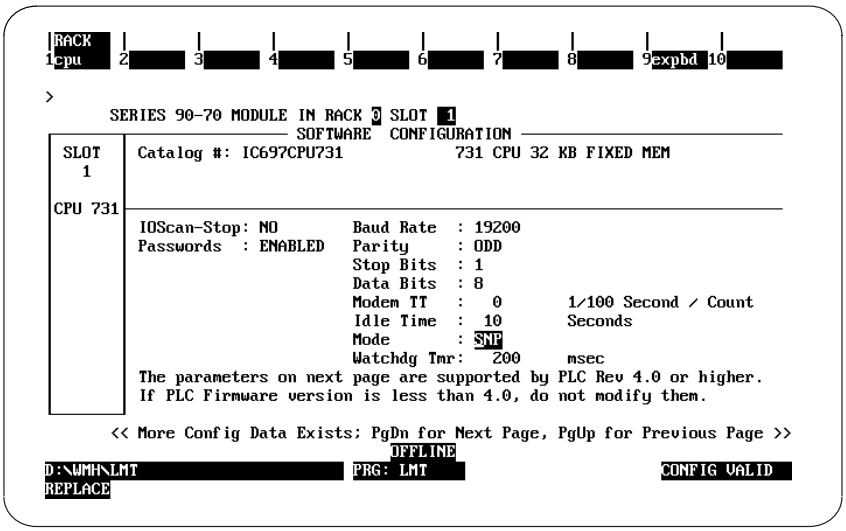


Figure 3-5. Default 90-70 CPU Configuration Data

4. Cursor down to the *Mode* field. The current mode selection is for SNP. Press the Tab key to change the selection to **MSG**.

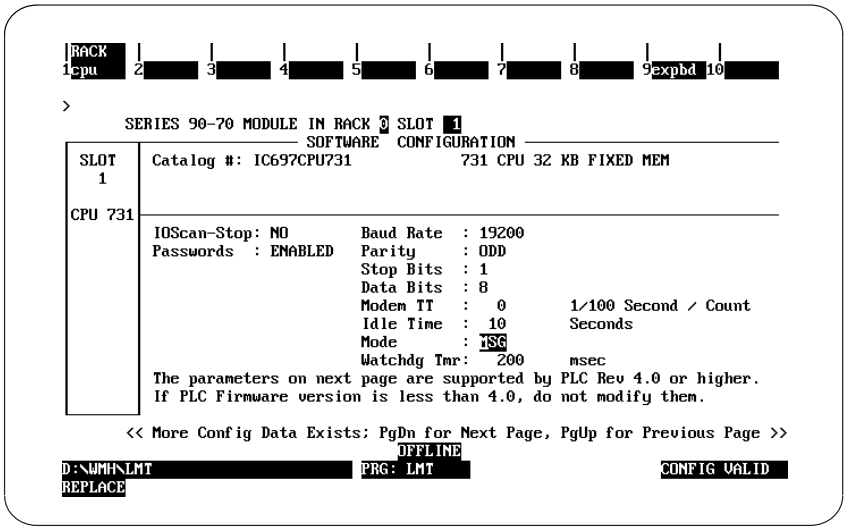


Figure 3-6. 90-70 CPU Configuration Modified for Message Generation Mode

5. Having modified the *Mode* field to **MSG**, store this configuration to the CPU to enable **`printf( )`** output on the serial port.

Differences in printf( ) and sprintf( ) — Series 90-70 Only

Printf( ) and **sprintf( )** will both format an output string. The basic difference between the two functions is that **printf( )** typically prints the formatted string to the standard output device and **sprintf( )** typically prints the formatted string to a user-specified buffer. Also, each of these functions returns the number of characters written to either the output port **printf( )** or to the output buffer **sprintf( )**. These operations are the standard definitions of these two functions.

When executed in the PLC, both **printf( )** and **sprintf( )** use the same string formatting code. Therefore, in general, the resulting format of a string from an **sprintf( )** to a buffer and a **printf( )** to the serial port will be the same. Similarly, the return status from each function, the number of characters written to the buffer (by **sprintf( )**) or to the serial port (by **printf( )**), will generally be the same for the same set of parameters.

However, there are two exceptions to this rule: 1) the handling of the **'\n'** character and 2) filling the serial port print queue. When the **'\n'** character (newline character, or 0A hexadecimal) is encountered by the low-level **printf( )** code, an **'\r'** (carriage return, or 0D hexadecimal) is inserted into the string immediately following the newline character. This is done to mimic the **printf( )** operation in the DOS environment. For each **'\r'** character inserted into the output string as a result of encountering a **'\n'** character, the total number of characters written by **printf( )** is incremented by 1. In contrast, the function **sprintf( )** does not add anything to the formatted string as a result of encountering a **'\n'** character. This difference in handling the **'\n'** character can cause the returned number of characters printed value from an **sprintf( )** and a **printf( )** to be different for the same input.

The filling of the serial port print queue is the second difference between **printf( )** and **sprintf( )** is a result of the **printf( )** output in the PLC being queued for transmission out of the serial port. Since the serial port queue used by **printf( )** is limited to 2048 characters and characters are being removed from the queue a byte at a time (exact rate of character removal from the queue is dependent upon the configured baud rate for the CPU serial port), it is possible for **printf( )** (and possibly multiple **printf( )**s from multiple blocks) to fill the queue faster than it can be emptied. The function **sprintf( )** has no queue associated with it; therefore, it is not limited by a queue size. It is only limited by the user-supplied destination buffer.

Example of printf( ) and sprintf( ) Returning the Same Value

The following C source code, when executed in the 90-70 PLC, will result in the same value being returned from both **printf( )** and **sprintf( )**:

```
main() {
    int    printf_chars, sprintf_chars;
    char   buffer[100];

    printf_chars  = printf("Hello world!");
    sprintf_chars = sprintf( buffer, "Hello world!");

    return(OK);
}
```

After executing the above example, both **sprintf_chars** and **printf_chars** will be equal to 12.

Example of printf( ) and sprintf( ) Returning Different Values

The following C source code, when executed in the Series 90-70 PLC, will result in different values being returned from **printf( )** and **sprintf( )**:

```
main() {
    int    printf_chars, sprintf_chars;
    char   buffer[100];

    printf_chars  = printf("Hello world!\n");
    sprintf_chars = sprintf( buffer, "Hello world!\n");

    return(OK);
}
```

After executing the above example, **sprintf_chars** is equal to 13, but **printf_chars** is equal to 14. Again, the additional character printed by **printf( )** is the **'\r'** (carriage return) character. Similarly, if the **printf( )** and **sprintf( )** lines are

```
    printf_chars  = printf("Hello world!\n\n");
    sprintf_chars = sprintf( buffer, "Hello world!\n\n");
```

After executing these two lines in the Series 90-70 PLC, **sprintf_chars** is equal to 14, but **printf_chars** is now equal to 16. An additional **'\r'** character is printed for each **'\n'** character encountered.

Floating Point and printf( ) and sprintf( )

Floating point numbers may only be formatted and printed using **printf( )** and/or **sprintf( )** on those Series 90-70 CPUs which have a math coprocessor (CPU models 732, 772, 782, 914, 915, 924, and 925). This restriction applies to the **printf( )** and **sprintf( )** format specifiers e, f, g, E, and G.

Note

An attempt to format and print (using **printf( )** or **sprintf( )**) a floating point number on a Series 90-70 CPU which does **not** have a math coprocessor will result in **printf( )** returning a status of −1 and no characters being printed.

GE Fanuc Functions

Additional functions are provided by the C Programmer's Toolkit in support of the Series 90 PLC CPU's operations. These functions are defined in **PLCC9070.H** and **PLCC9030.H**.

A description of the functions is provided in the sections that follow.

General PLC Functions

The following functions make PLC features available to C applications.

PLCC_read_elapsed_clock

```
int PLCC_read_elapsed_clock(struct elapsed_clock_rec
   *elapsed_clock_value);

struct elapsed_clock_rec {
   unsigned long seconds;
   word   hundred_usecs;
}
```

This function returns the current time from the PLC in **elapsed_time_clock_value**. The return value is 0 if successful, −1 if unsuccessful.

PLCC_chars_in_printf_q (Series 90-70 Only)

```
int PLCC_chars_in_printf_q(void);
```

This function returns the number of characters in the PLC printf queue.

PLCC_gen_alarm (Series 90-70 Only)

```
int PLCC_gen_alarm(word error_code, char *fault_string);
```

This function puts the fault described by **error_code** and **fault_string** into the PLC fault table. This function will return 0 if successful and −1 if unsuccessful.

PLCC_get_plc_version

```
int PLCC_get_plc_version(struct PLC_ver_info_rec *PLC_ver_info);

struct PLC_ver_info_rec {
   word   family;   /* Host PLC product line */
   word   model;    /* Specific Model of PLC */
   byte   sw_ver;   /* Major Version of PLC firmware */
   byte   sw_rev;   /* Minor Revision of PLC firmware */
};
```

This function returns the PLC information through the structure passed to the routine. All the data returned from the PLC will be hexadecimal. For example, the PLC will return hexadecimal 782 (1922 decimal) in the model field for a 782 PLC. The function will return 0 if successful and −1 if unsuccessful.

VME Functions (Series 90-70 Only)

The following functions are based on the VME functions available in ladder logic.

PLCC_VME_config_read

```
word PLCC_VME_config_read(void *buffer, word length, byte rack,
   byte slot, long offset);
```

This function will read from a VME module at the location specified by rack, slot and offset using the VME module's configuration data.

PLCC_VME_config_write

```
word PLCC_VME_config_write(void *buffer, word length, byte rack,
   byte slot, long offset);
```

This function will write to a VME module at the location specified by rack, slot and offset using the VME module's configuration data.

PLCC_VME_set_amcode

```
byte PLCC_VME_set_amcode(byte amcode);
```

This function will set the address mode of the VME bus. This function must be called before any of the VMERD or VMEWRT functions may be used.

This function returns the value passed in when it is successful. All other return values indicate an invalid amcode input parameter.

VMERD (BYTE, WORD)—Series 90-70 Only

The VME Read (VMERD) function is used to read data from the VME bus.

Note

Using a VME function (VMERD, VMEWRT, VMERMW, or VMETST) requires additional information on the correct way to address the VME board. This information may be obtained from one of two sources. For a qualified VME board, the VME board vendor may issue application notes on the correct use of the board. Otherwise, refer to the *Guidelines for the Selection of Third-Party VME Modules*, GFK-0448.

PLCC_VME_read_byte

```
byte PLCC_VME_read_byte(byte *value, unsigned long address);
```

This function will read a byte from the VME bus.

PLCC_VME_read_word

```
byte PLCC_VME_read_word(word *value, unsigned long_address);
```

This function will read a word from the VME bus. The length parameter specifies the number of bytes to be read.

PLCC_VME_read_block

```
byte PLCC_VME_read_block(void *buffer, word length, unsigned long
address);
```

This function will read a block of data from the VME bus. The length parameter specifies the number of bytes to be read.

VMEWRT (BYTE, WORD)—Series 90-70 Only

The VME Write (VMEWRT) function is used to write data to the VME bus.

Note

Using a VME function (VMERD, VMEWRT, VMERMW, or VMETST) requires additional information on the correct way to address the VME board. This information may be obtained from one of two sources. For a qualified VME board, the VME board vendor may issue application notes on the correct use of the board. Otherwise, refer to the *Guidelines for the Selection of Third-Party VME Modules,* GFK-0448.

PLCC_VME_write_byte

```
byte PLCC_VME_write_byte(byte value, unsigned long address);
```

This function will write a byte for data to the VME bus.

PLCC_VME_write_word

```
byte PLCC_VME_write_word(word value, unsigned long address);
```

This function will write a word to the VME bus.

PLCC_VME_write_block

```
byte PLCC_VME_write_block(void *buffer, word length, unsigned long
address);
```

This function will write a block of data to the VME bus. The length parameter specifies the number of bytes to be written.

Return Status for VME Functions

Each of these VME functions:

```
extern byte _far PLCC_VME_read_byte(byte *value, unsigned long address);
extern byte _far PLCC_VME_read_word(word *value, unsigned long address);
extern byte _far PLCC_VME_read_block(void *buffer, word length, unsigned long address);
extern byte _far PLCC_VME_write_byte(byte value, unsigned long address);
extern byte _far PLCC_VME_write_word(word value, unsigned long address);
extern byte _far PLCC_VME_write_block(void *buffer, word length, unsigned long address)
```

have the function return values shown in the following table:

Table 3-2. Return Status for VME Functions

Return Status (in hexadecimal)	Meaning
0xFF	Operation successful
All other return values have the following format: 11xx x1xx	 Active bits (failures) are 0, inactive are 1.
bit 0	SYSFAIL* occurred during operation
bit 1	BERR* occurred during operation
bit 2	always 1
bit 3	Bus grant error occurred during operation
bit 4	Bus Hog occurred during operation
bit 5	Bus Interrupt Acknowledge failure
bit 6	always 1
bit 7	always 1

Each of these two VME functions:

extern byte _far PLCC_VME_config_read (void *buffer, word length, byte rack, byte slot, long offset)

and

extern byte _far PLCC_VME_config_write (void *buffer, word length, byte rack, byte slot, long offset)

have the function return values shown in the following table:

Table 3-3. Return Status and Meaning from VME_config_read and write

Return Status	Meaning
0000	operation successful
1	Config Errors
	Board at specified rack, slot is not configured as a foreign VME module.
	Board at specified rack, slot is foreign VME but not configured for "Bus Interface" mode.
2	Range Errors
	slot < 2 or slot > 9 or rack > 7
	slot address is in slot+subslot form for IRACK addressing and subslot is not A or B
	The most significant byte of the offset parameter is not 0. (The offset parameter, OFF, is a DWORD.)
	The relative address (i.e., 1 based, ignoring configured dual port base address) of the last data element to be read/written is > the configured dual port size.
	The absolute address (i.e., including the dual port base address) of the last data element is > 24 bits long for standard AM codes or > 16 bits long for short AM codes.
	The absolute address of the first data element is even and the configured Interface Type is Odd Byte Only.
	The absolute Address of the first data element is odd and the configured Interface Type is Word Access or Single Word Access.
	The specified AM code is unsupported.
All other return status values:	Interpret meaning by examining each bit of the return value using the following guide:
11xx x1xx	Active bits (failures) are 0, inactive are 1
bit 0	SYSFAIL* occurred during operation
bit 1	BERR* occurred during operation
bit 2	always 1
bit 3	Bus grant error occurred during operation
bit 4	Bus Hog occurred during operation
bit 5	Bus Interrupt Acknowledge failure
bit 6	always 1
bit 7	always 1

Service Request Functions

The following functions are patterned after the service request (SVC_REQ) in ladder logic.

PLCC_const_sweep_timer

```
int PLCC_const_sweep_timer(struct const_sweep_timer_rec *x);

/* input structure */
struct const_sweep_input_rec {
   word   action;
   word   timer_value;
};

/* structure with return value */
struct const_sweep_output_rec {
   word   sweep mode;
   word   current_time_value;
};

struct const_sweep_timer_rec {
   union {
      struct const_sweep_input_rec  input;
      struct const_sweep_output_rec output;
   };
};

/* sweep mode values – these determine which action is to be taken */
#define DISABLE_CONSTANT_SWEEP_MODE  0
#define ENABLE_CONSTANT_SWEEP_MODE   1
#define CHANGE_TIMER_VALUE           2
#define READ_TIMER_VALUE_AND_STATE   3
```

This function is the C interface to service request #1 (Change/Read Constant Sweep Timer).

This function can be used to

- Disable constant sweep time mode
- Enable constant sweep time mode and use the old timer value
- Enable constant sweep time mode and use a new timer value
- Set a new timer value only
- Read constant sweep mode state timer and value

Setting sweep_mode to DISABLE_CONSTANT_SWEEP_MODE will disable the constant sweep timer. Setting sweep_mode to ENABLE_CONSTANT_SWEEP_MODE will enable the constant with the value in sweep_timer, or keep the current value if the sweep_timer is 0. Setting the sweep_mode to CHANGE_TIMER_VALUE will change the constant sweep timer to the value in timer_value. Setting sweep_mode to READ_TIMER_VALUE_AND_STATE will set sweep_enabled to 1 if the constant sweep timer is enabled, and will set the current constant sweep timer value to the current_value. This function will return 1 if successful and 0 if unsuccessful, and −1 if not supported.

PLCC_read_window_values

```
int PLCC_read_window_values(struct read_window_values_rec *x);

struct read_window_values_rec {
   byte   prog_win_time;
   byte   prog_win_mode;
   byte   sys_comm_win_time;
   byte   sys_comm_win_mode;
   byte   background_win_time;
   byte   background_win_mode;
};

/*
 * window modes
 */
#define LIMITED_MODE                0
#define CONSTANT_MODE               1
#define RUN_TO_COMPLETION_MODE      2
```

This function is the C interface to service request #2 (Read Window Values). This function will return the mode and time for the programmer communications window, the system communications window, and the background task window in the structure. The possible values for the mode fields are LIMITED_MODE, CONSTANT_MODE, and RUN_TO_COMPLETION_MODE. The time fields contain the time values in milliseconds. This function will return 1 if successful and 0 if unsuccessful.

PLCC_change_prog_comm_window

```
int PLCC_change_prog_comm_window(struct change_prog_comm_window_rec
*x);

struct change_prog_comm_window_rec {
   byte   time;
   byte   mode;
};

/*
 * window modes
 */
#define LIMITED_MODE             0
#define CONSTANT_MODE            1 (Not supported on Series 90-30)
#define RUN_TO_COMPLETION_MODE   2
```

This function is the C interface to service request #3 (Change Programmer Communications Window State and Values). This function will change the programmer communications window state and timer to the values specified in the structure. The mode will be changed to one of the three states LIMITED_MODE, CONSTANT_MODE, or RUN_TO_COMPLETION_MODE depending on the value in the mode field. The timer value should be from 1 to 255 milliseconds. This function will return 1 if successful and 0 if unsuccessful.

Note

Window time values cannot be changed on the Series 90-30.

PLCC_change_system_comm_window

```
int PLCC_change_system_comm_window(struct
change_system_comm_window_rec *x);

struct change_system_comm_window_rec {
   byte   time;
   byte   mode;
};

/*
 * window modes
 */
#define LIMITED_MODE             0
#define CONSTANT_MODE            1 (Not supported on Series 90-30)
#define RUN_TO_COMPLETION_MODE   2
```

This function is the C interface to service request #4 (Change System Communications Window State and Values). This function will change the system communications window state and timer to the values specified in the structure. The mode will be changed to one of the three states LIMITED_MODE, CONSTANT_MODE, or RUN_TO_COMPLETION_MODE depending on the value in the mode field. The timer value should be from 1 to 255 milliseconds. This function will return 1 if successful and 0 if unsuccessful.

Note

Window time values cannot be changed on the Series 90-30.

PLCC_change_background_window (Series 90-70 Only)

```
int PLCC_change_background_window(struct change_background_window_rec
*x);

struct change_background_window_rec {
   byte   time;
   byte   mode;
};

/*
 * window modes
 */
#define LIMITED_MODE             0
#define CONSTANT_MODE            1
#define RUN_TO_COMPLETION_MODE   2
```

This function is the C interface to service request #5 (Change_Background Window State and Values). This function will change the background window state and timer to the values specified in the structure. The mode will be changed to one of the three states LIMITED_MODE, CONSTANT_MODE, or RUN_TO_COMPLETION_MODE depending on the value in the mode field. The timer value should be from 1 to 255 milliseconds. This function will return 1 if successful and 0 if unsuccessful.

PLCC_number_of_words_in_chksm

```
int PLCC_number_of_words_in_chksm(struct number_of_words_in_chksm_rec
*x);

struct number_word_of_words_in_chksm_rec {
   word   read_set;
   word   word_count;
};

#define READ_CHECKSUM_WORDS   0
#define SET_CHECKSUM_WORDS    1
```

This function is the C interface to service request #6 (Change/Read Checksum Task State and Number of Words to Checksum). This function will either read the current checksum word count or set a new checksum word count depending on the value in read_set. If read_set is READ_CHECKSUM then the function will read the current word count and return it in word_count. If the read_set is SET_CHECKSUM then the function will set the current word count to word_count rounded to the nearest multiple of 8. To disable the checksums set the word_count to 0. The function will fail if the read_write field is set to a value other than 0 or 1. This function will return 1 if successful and 0 if unsuccessful.

PLCC_tod_clock

```
int PLCC_tod_clock(struct tod_clock_rec *x);

struct num_tod_rec {
   word   year;
   word   month;
   word   day_of_month;
   word   hours;
   word   minutes;
   word   seconds;
   word   day_of_week;
};

struct BCD_tod_rec {
   byte   year;
   byte   month;
   byte   day_of_month;
   byte   hours;
   byte   minutes;
   byte   seconds;
   byte   day_of_week;
   byte   null;
};
```

```
struct unpacked_BCD_rec {
   byte   yearlo;
   byte   yearhi;
   byte   monthlo;
   byte   monthhi;
   byte   day_of_month_lo;
   byte   day_of_month_hi;
   byte   hourslo;
   byte   hourshi;
   byte   minslo;
   byte   minshi;
   byte   secslo;
   byte   secshi;
   word   day_of_week;
};

struct ASCII_tod_rec {
   byte   yearhi;
   byte   yearlo;
   byte   space1;
   byte   monthhi;
   byte   monthlo;
   byte   space2;
   byte   day_of_month_hi;
   byte   day_of_month_lo;
   byte   space3;
   byte   hourslo;
   byte   hourshi;
   byte   colon1;
   byte   minshi;
   byte   minslo;
   byte   colon2;
   byte   secshi;
   byte   secslo;
   byte   space4;
   byte   day_of_week_hi;
   byte   day_of_week_lo;
};

struct tod_clock_rec {
   word   read_write;
   word   format;
   union {
      struct num_tod_rec      num_tod;
      struct BCD_tod_rec      BCD_tod;
      struct unpacked_BCD_rec unpacked_BCD_tod;
      struct ASCII_tod_rec    ASCII_tod
};
```

```
#define READ_CLOCK  0
#define WRITE_CLOCK 1

#define NUMERIC_DATA_FORMAT   0 (not supported on 9030)
#define BCD_FORMAT            1
#define UNPACKED_BCD_FORMAT   2 (not supported on 9030)
#define PACKED_ASCII_FORMAT   3

#define SUNDAY      1
#define MONDAY      2
#define TUESDAY     3
#define WEDNESDAY   4
#define THURSDAY    5
#define FRIDAY      6
#define SATURDAY    7
```

This function is the C interface to service request #7 (Change/Read Time-of-Day Clock State and Values). If read_write is zero then the function will read the Time-of-Day Clock into the structure passed. If read_write is one then the function will write the values in the structure to the time_of_day_clock. The format will be based on the format field in the structure (NUMERIC_DATA_FORMAT, BCD_FORMAT, UNPACKED_BCD_FORMAT, and PACKED_ASCII_FORMAT). This function will return 1 if successful and 0 if unsuccessful. The function will fail in the following instances:

- If read_write is some number other than 0 or 1
- If format is some number other than 0 - 3
- If data for a write does not match format

For all the formats the hours are 24 hour and the days of the week are defined as macros in **PLCC9070.H** or **PLCC9030.h**. The packed BCD format needs the null field to be 0, as shown in the following example:

Series 90-70 Example

```
#include "plcc9070.h"

int main()
{
        struct  tod_clock_rec data;

        data.read_write = 1;
        data.format = BCD_FORMAT;

        /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
        data.BCD_tod.year = 0x94;
        data.BCD_tod.month = 8;
        data.BCD_tod.day_of_month = 9;
        data.BCD_tod.hours = 0x13;
        data.BCD_tod.minutes = 0x13;
        data.BCD_tod.seconds = 8;
        data.BCD_tod.day_of_week = TUESDAY;
        data.BCD_tod.null = 0;
        PLCC_tod_clock (& data)
}
```

Series 90-30 Example

```
#include “plcc9030.h”

EXE_stack_size=2048;

int main()
{
        struct  tod_clock_rec data;

        data.read_write = 1;
        data.format = BCD_FORMAT;

        /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
        data.BCD_tod.year = 0x94;
        data.BCD_tod.month = 8;
        data.BCD_tod.day_of_month = 9;
        data.BCD_tod.hours = 0x13;
        data.BCD_tod.minutes = 0x13;
        data.BCD_tod.seconds = 8;
        data.BCD_tod.day_of_week = TUESDAY;
        data.BCD_tod.null = 0;
        PLCC_tod_clock (& data)

}
```

The unpacked format should have a digit in every byte (including the day of the week) as shown in the following 90-70 example:

This example is for the Series 90-70 Only.

```
#include “plcc9070.h”

int main()
{
        struct  tod_clock_rec data;

        data.read_write = 1;
        data.format = UNPACKED_BCD_FORMAT;

        /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
        data.unpacked_BCD_tod.yearhi = 9;
        data.unpacked_BCD_tod.yearlo = 4;
        data.unpacked_BCD_tod.monthhi = 0;
        data.unpacked_BCD_tod.monthlo = 8;
        data.unpacked_BCD_tod.day_of_month_hi = 0;
        data.unpacked_BCD_tod.day_of_month_lo = 9;
        data.unpacked_BCD_tod.hourshi = 1;
        data.unpacked_BCD_tod.hourslo = 3;
        data.unpacked_BCD_tod.minshi = 1;
        data.unpacked_BCD_tod.minslo = 3;
        data.unpacked_BCD_tod.secshi = 0;
        data.unpacked_BCD_tod.secslo = 8;
        data.unpacked_BCD_tod.day_of_week = TUESDAY;
        PLCC_tod_clock (& data)

}
```

The packed ASCII format should have an ASCII character in every byte as shown in the following example:

Series 90-70 Example

```
#include “plcc9070.h”

int main()
{
        struct  tod_clock_rec data;

        data.read_write = 1;
        data.format = PACKED_ASCII_FORMAT;

        /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
        data.ASCII_tod.yearhi = ’9’;
        data.ASCII_tod.yearlo = ’4’;
        data.ASCII_tod.space1 = ’ ’;
        data.ASCII_tod.monthhi = ’0’;
        data.ASCII_tod.monthlo = ’8’;
        data.ASCII_tod.space2 = ’ ’;
        data.ASCII_tod.day_of_month_hi = ’0’;
        data.ASCII_tod.day_of_month_lo = ’9’;
        data.ASCII_tod.space3 = ’ ’;
        data.ASCII_tod.hourshi = ’1’;
        data.ASCII_tod.hourslo = ’3’;
        data.ASCII.tod.colon1 = ’:’;
        data.ASCII_tod.minshi = ’1’;
        data.ASCII_tod.minslo = ’3’;
        data.ASCII_tod.colon2 = ’:’;
        data.ASCII_tod.secshi = ’0’;
        data.ASCII_tod.secslo = ’8’;

        /* place 0 ASCII (30 hex) in the high byte for the number */
        data.ASCII_date.day_of_weekhi = ’0’;

        /* place TUESDAY(3) plus 30 hex into the lo      */
        /* byte to make the number an ASCII character    */
        data.ASCII_date.day_of_weeklo = TUESDAY+0x30;
        PLCC_tod_clock_rec (& data)

}
```

PLCC_reset_watchdog_timer

```
int PLCC_reset_watchdog_timer(void);
```

This function is the C interface to service request #8 (Reset Watchdog Timer). This function will reset the watchdog timer during the sweep. When the watchdog timer expires, the PLC shuts down without warning. This function allows the timer to be refreshed during a time-consuming task. This function will return 1 if successful and 0 if unsuccessful.

Caution

Be careful resetting the watchdog timer. It may affect the process.

PLCC_time_since_start_of_sweep

```
int PLCC_time_since_start_of_sweep(struct
time_since_start_of_sweep_rec *x);

struct time_since_start_of_sweep_rec {
   word   time_since_start_of_sweep;
};
```

This function is the C interface to service request #9 (Read Sweep Time from Beginning of Sweep). The function will read the time in milliseconds from the beginning of the sweep. The function will return 1 if successful and 0 if unsuccessful.

PLCC_read_folder_name

```
int PLCC_read_folder_name(struct read_folder_name_rec *x);

struct read_folder_name {
   char   folder_name[8];
};
```

This function is the C interface to service request #10 (Read Folder Name This Application is In). This function will return the application folder name as a NULL terminated string. The function will return 1 if successful and 0 if unsuccessful.

PLCC_read_PLC_ID

```
int PLCC_read_PLC_ID(struct read_PLC_ID_rec *x);

struct read_PLC_ID_rec {
   char PLC_ID[8];
};
```

This function is based on service request #11 (Read PLC ID). The function will return the name of the Series 90 PLC (in ASCII). The function will return 1 if successful and 0 if unsuccessful.

PLCC_read_PLC_state

```
int PLCC_read_PLC_state(struct read_PLC_state_rec *x);

struct read_PLC_state_rec {
   word   state;
};

#define RUN_DISABLED   1
#define RUN_ENABLED    2
```

This function is based on service request #12 (Read PLC Run State). This function will return the PLC run state (RUN_DISABLED or RUN_ENABLED). The function will return 1 if successful and 0 if unsuccessful.

PLCC_shut_down_plc

```
int PLCC_shut_down_plc(void);
```

This function is the C interface to service request #13 (Shut Down/Stop PLC). The function will stop the PLC at the end of the current sweep. All outputs will go to their values at the beginning of the next sweep and "STOPPED by SVC 13" information fault will be logged in the PLC fault table. The function will return 1 if successful, and 0 if unsuccessful.

PLCC_clear_fault_tables

```
int PLCC_clear_fault_tables(struct clear_fault_tables_rec *x);

struct clear_fault_tables_rec {
   word table;
};

#define PLC_FAULT_TABLE    0
#define IO_FAULT_TABLE     1
```

This function is the C interface to service request #14 (Clear Fault Tables). The function will clear the fault table according to the value (PLC_FAULT_TABLE or IO_FAULT_TABLE). The function will return 1 if successful and 0 if unsuccessful.

PLCC_read_last_fault

```
int PLCC_read_last_fault(struct read_last_fault_rec *x);

struct PLC_entry_rec {
   byte   long_short;
   byte   reserved[3];
   word   PLC_fault_address[2];
   word   fault_group;
   word   error_code;
   word   fault_specific_data[12];
   word   time_stamp[3];
};
```

```
struct IO_entry_rec{
   byte   long_short;
   byte   reference_address[3];
   word   IO_fault_address[3];
   word   fault_group;
   byte   fault_category;
   byte   fault_type;
   byte   fault_description;
   byte   fault_specific_data[21];
   word   time_stamp[3];
};

struct read_last_fault_rec {
   word   table;
   union {
      struct    PLC_entry_rec PLC_entry;
      struct    IO_entry_rec  IO_entry_rec;
   };
};

#define PLC_FAULT_TABLE    0
#define IO_FAULT_TABLE     1
```

This function is the C interface to service request #15 (Read Last-Logged Fault Table Entry). The function will return the last fault table entry of the table specified in the table field (PLC_FAULT_TABLE, or IO_FAULT_TABLE). The function returns a 1 if successful and a 0 if unsuccessful.

In the return data, the long/short indicator defines the quantity of fault data present in the fault entry. In the PLC fault table, a long/short value of 00 represents 8 bytes of fault extra data present in the fault entry, and 01 represents 24 bytes of fault extra data. In the I/O fault table, 02 represents 5 bytes of fault specific data, and 03 represents 21 bytes.

PLCC_mask_IO_interrupts (Series 90-70 Only)

```
int PLCC_mask_IO_interrupts(struct mask_IO_interrupts_rec *x);

struct mask_IO_interrupts_rec {
   word   mask;
   word   memory_type;
   word   memory_address;
};

#define MASK     1
#define UNMASK   0

/*
 * memory types
 */
#define IBIT  70
#define AIMEM 10
```

This function is the C interface to service request #17 (Mask/Unmask I/O Interrupt). The function will mask or unmask I/O interrupts according to the value in mask (MASK or UNMASK). The memory type of the input to mask or unmask is in memory type. The value for the memory_type can be %I (IBIT) or %AI (AIMEM). The address specified must match a Series 90 input module with maskable channel and interrupts enabled. This function will return 1 if successful and 0 if unsuccessful.

PLCC_read_IO_override_status

```
int PLCC_read_IO_override_status(struct read_IO_override_status_rec *x);

   struct read_IO_override_status_rec {
      word   override_status;
   };

   #define OVERRIDES_SET      1
   #define NO_OVERRIDES_SET   0
```

This function is the C interface to service request #18 (Read I/O Override Status). The function will return the override_status (OVERRIDES_SET, or NO_OVERRIDES_SET). The function will return 1 if successful and 0 if unsuccessful.

PLCC_set_run_enable (Series 90-70 Only)

```
   int PLCC_set_run_enable(struct set_run_enable_rec *x);

   struct set_run_enable_rec {
      word   enable;
   };

   #define RUN_ENABLED    1
   #define RUN_DISABLED   2
```

This function is the C interface to service request #19 (Set Run Enable/Disable). The function will set the PLC in either RUN_ENABLED or RUN_DISABLED depending on what value was passed in the structure. The function will return 1 if successful and 0 if unsuccessful.

Use SVCREQ function #19 to permit the ladder program to control the **RUN** mode of the CPU.

PLCC_read_fault_tables (Series 90-70 Only)

```
   int PLCC_read_fault_tables(struct read_fault_tables_rec *x);

   struct PLC_entry_rec {
      byte   long_short;
      byte   reserved[3];
      word   PLC_fault_address[2];
      word   fault_group;
      word   error_code;
      word   fault_specific_data[12];
      word   time_stamp[3];
   };

   struct IO_entry_rec{
      byte   long_short;
      byte   reference_address[3];
      word   IO_fault_address[3];
      word   fault_group;
      byte   fault_category;
      byte   fault_type;
      byte   fault_description;
      byte   fault_specific_data[21];
      word   time_stamp[3];
   };
```

```
struct time_of_day_clock_rec{
    byte   seconds;
    byte   minutes;
    byte   hours;
    byte   day_of_month;
    byte   month;
    byte   year;
}

struct read_fault_tables_rec {
    word   table;
    word   zero;
    word   reserved[13];
    struct time_of_day_clk_rec time_since_clear;
    word   num_faults_since_clear;
    word   num_faults_in_queue;
    word   num_faults_read;
    union {
        struct PLC_entry_rec PLC_faults[NUM_PLC_FAULT_ENTRIES];
        struct IO_entry_rec  IO_faults[NUM_IO_FAULT_ENTRIES];
    };
};

#define PLC_FAULT_TABLE 0
#define IO_FAULT_TABLE  1
```

This function is the C interface to service request #20 (Read Fault Tables). The function will read the fault table specified in the table field (PLC_FAULT_TABLE or IO_FAULT_TABLE). The function will return the table in an array of PLC_faults or IO_faults. The zero field and the reserved fields do not hold fault data. The **time_since_clear** fields are BCD numbers with seconds in the low order nibble and tens of seconds in the high order nibble. The **num_faults_since_clear** field shows the number of faults that have occurred since the table was last cleared. The **num_faults_read** field shows the number of faults read into the arrays for I/O and PLC faults; there is room for the entire table, but only the **num_faults_read** field will have valid data. The function will return 1 if successful, and 0 if unsuccessful.

In the return data, the long/short indicator defines the quantity of fault data present in the fault entry. In the PLC fault table, a long/short value of 00 represents 8 bytes of fault extra data present in the fault entry, and 01 represents 24 bytes of fault extra data. In the I/O fault table, 02 represents 5 bytes of fault specific data, and 03 represents 21 bytes.

There are a maximum of 16 PLC fault table entries and 32 I/O fault table entries. If the fault table read is empty, no data is returned.

PLCC_mask_timed_interrupts (Series 90-70 Only)

```
int PLCC_mask_timed_interrupts(struct mask_timed_interrupts_rec *x);

struct mask_timed_interrupts_rec {
   word   mask_toggle;
   word   status;
};

#define READ_INTERRUPT_MASK   0
#define WRITE_INTERRUPT_MASK  1
#define MASK                  1
#define UNMASK                0
```

This function is the C interface to service request #22 (Mask/Unmask Timed Interrupts). This function returns 1 if successful 0 if unsuccessful. Use this function to mask or unmasked timed interrupts and to read the current mask. When the interrupts are masked, the PLC CPU will not execute any interrupt block that is associated with a timed interrupt. Timed interrupts are masked/unmasked as a group. They cannot be individually masked or unmasked.

To read current mask, set **mask_toggle** to 0.

To change current mask to unmask timed interrupts, set **mask_toggle** to 1 and **status** to 0.

To change current mask to mask timed interrupts, set **mask_toggle** to 1 and **status** to 1.

Successful execution will occur unless some number other than 0 or 1 is entered as the requested operation or mask value.

PLCC_sus_res_HSC_interrupts (Series 90-70 Only)

```
int PLCC_sus_res_HSC_interrupts(struct sus_HSC_interrupts_rec *x);

struct sus_res_HSC_interrupts_rec {
   word   enable;
   word   memory_type;
   word   reference_address
};

/*
 * memory types
 */
#define IBIT  70
#define AIMEM 10

#define ENABLE   0
#define DISABLE  1
```

This function is based on service request #32 (Suspend High Speed Counter Interrupts). The function will enable or disable the high speed counter interrupts for a given address and memory type.

PLCC_get_escm_status (Series 90-70 Only)

```
extern int _far PLCC_get_escm_status (struc escm_status_rec *);
struc escm_status_rec {
   word port_number;
   word port_status;

};
#define port_1   1
#define port_2   2
```

If the function return value is zero (0), the function was not successful, usually indicating that the PLC does not support ESCM ports (see Note below). If the function return value is one (1), the function was successful.

This function also returns a status word for Ports 1 or 2 (word port_status). The bit values for that word are shown in the following table:

Table 3-4. Port_Status for the PLCC_get_escm_status Function

Port Status	Meaning
bit 0	PORTN_OK: Requested port is ready. If value is 1, the port is ready. If value is 0, the port is not usable.
bit 1	PORTN_ACTIVE: There is activity on this port. If value is 1, the port is active. If value is 0, the port is inactive.
bit 2	PORTN_DISABLED: Requested port is disabled. If value is 1, the port is disabled. If value is 0, the port is enabled.
bit 3	PORTN_FUSE_BLOWN: Requested port's fuse is blown (for Port 2) or supply voltage is not within range (for Port 1). If value is 1, the fuse is blown (or voltage not within range). If value is 0, the fuse (or supply voltage) is okay.

Note

This function is supported ***only*** for Series 90-70 PLCs which support Ports 1 and 2, specifically, CGR models beginning with Release 7.85 and all CPX models.

Module Communications

PLCC_comm_req

```
int PLCC_comm_req(struct comm_req_rec *x);

struct status_addr {
   word   seg_selector;
   word   offset;
};

struct comm_req_command_blk_rec {
   word   length;
   word   wait;
   struct status_addr  status;
   word   idle_timeout;
   word   max_comm_time;
   word   data[128];
};

struct comm_req_rec {
   struct comm_req_command_blk_rec   *command_blk;
   byte   slot;
   byte   rack;
   dword  task_id;
};
```

This function is based on the **COMM_REQ** ladder logic block. The function returns 1 if successful and 0 if unsuccessful.

Ladder Function Blocks

For PLCC_do_io, the Offset and Length for Word types is in units of Words. For Bit types, the Offset and Length is in units of Bits. Offset and Length is 1-based.

PLCC_do_io

```
int PLCC_do_io(struct do_io_rec *x);

struct do_io_rec {
   byte   start_mem_type;
   word   start_mem_offset;
   word   length;
   byte   alt_mem_type;
   word   alt_mem_offset;
};

#define NULL_SEGSEL    0xFF  (Only valid for alt_mem_type)

/*
 * memory types
 */
#define I_MEM    16
#define Q_MEM    18
#define R_MEM    8
#define AI_MEM   10
#define AQ_MEM   12
#define WORD_CONSTANT  14 (Used for Enhanced DO_IO on 90-30)
```

Note

Refer to the DO I/O section of Chapter 4 or the *Series 90-30/20/Micro Programmable Controllers Reference Manual* (GFK-0467) or the online help in the Windows-based programming software for restrictions and extentions of the 90-30 DO I/O—make sure you have a 90-30 open so that you will receive 90-30 context sensitive help.

This function is used to update inputs or outputs for one scan while the program is running. This function can be used in conjunction with a suspend I/O function , which stops the normal I/O scan. (Suspend I/O is a 90-70 only.) It can also be used to update selected I/O during the program, in addition to the normal I/O scan.

If input references are specified, the function allows the most recent values of inputs to be obtained for program logic. If output references are specified, **`PLCC_do_io`** updates outputs based on the most current values stored in I/O memory. I/O points are serviced in increments of entire I/O modules; the PLC adjusts the references, if necessary, while the function executes. The **`PLCC_do_io`** function will not scan I/O modules that are not configured.

Note

The **`PLCC_do_io`** function is supported for Series 90 I/O modules only. It does not support Genius I/O modules or FIP I/O modules.

When this function executes, the input point specified by **`start_mem_type`** and **`start_mem_offset`** and the bits included (as specified by length) are scanned. If **`alternate_mem_type`** and **`alternate_mem_offset`** is defined, a copy of the data is placed in alternate memory, and the real input points are not updated. If this function references output data, data specified in **`start_mem_type`** and **`start_mem_offset`** is written to the output modules. If alt locations are defined, the alternate data is written to the output modules.

Execution of the function continues until either all inputs in the selected range have reported or all outputs have been serviced on the I/O cards.

The function return a 1 unless one or more of the following is true (in which case it returns a 0):

- Not all references of the type specified are present within the selected range
- The CPU is not able to properly handle the temporary list of I/O created by the function
- The range specified includes I/O modules that are associated with a “Loss of I/O Module” fault

Note

If the function is used with timed or I/O interrupts, transitional contacts associated with scanned inputs may not operate as expected.

Enhanced DO I/O (Series 90-30 Only)

```
main ()
{

struct do_io_rec doio;
doio.start_mem=I_MEM;
doio.start_mem_offset=1;
doio.length=16;
doio.alt_mem_type=WORD_CONSTANT;
doio.alt_mem_offset=5;

PLCC_do_io(&doio);

}
```

This enhanced DO I/O function can only be used on a single discrete input or discrete output 8-point, 16-point, or 32-point module. The enhanced DO I/O function is faster than the standard DO I/O because the only checking it performs is to check the state of the module in the slot specified to see if the module is okay. For more information about the Enhanced DO I/O function, refer to the Enhanced DO I/O section of the *Series 90-30/20/Micro Programmable Controllers Reference Manual* (GFK-0467) or the online help within the Windows-based programming software.

Note

To use the Enhanced DO I/O, the alt_mem_type must be set to WORD_CONSTANT as shown above, and the alt_mem_offset is always the slot number.

PLCC_sus_io (Series 90-70 Only)

```
int PLCC_sus_io(void);
```

This function is used to stop normal I/O scans from occurring for **one** CPU sweep. During the next output scan, all outputs are held at their current states. During the next input scan, the input references are not updated with data from inputs. However, during the input scan portion of the sweep the CPU will verify that Genius Bus Controllers have completed their previous output updates.

Note

This function suspends all I/O, both analog and discrete, whether rack I/O or Genius I/O.

The **`PLCC_sus_io`** function returns a 1 if successful, 0 if unsuccessful.

VME Semaphore Handlers (Series 90-70 Only)

The following functions are designed to enable semaphore handling on the VME bus. As a result, the functions cannot be interrupted.

VME Read Modify Write (Series 90-70 Only)

The following functions are intended to handle masking semaphores on the VME bus.

PLCC_VME_RMW_byte

```
int PLCC_VME_RMW_byte (byte op_type, byte mask, unsigned long
   address);

opt_type
#define VME_AND 0
#define VME_OR 1
```

This function will read the byte semaphore off the bus. If the semaphore is free then the function will perform the indicated function (either an AND or an OR with the mask). This function will return −1 if not supported. For a full explanation of all return values, refer to page 3-51, Table 3-5.

PLCC_VME_RMW_word

```
int PLCC_VME_RMW_word (word op_type, word mask, unsigned long
   address);

opt_type
#define VME_AND 0
#define VME_OR 1
```

This function will read the word semaphore off the bus. If the semaphore is free then the function will perform the indicated function (either an AND or an OR with the mask). This function will return −1 if not supported. For a full explanation of all return values, refer to page 3-51, Table 3-5.

VME Test and Set (Series 90-70 Only)

The following functions are intended to handle simple on or off semaphores on the VME bus.

PLCC_VME_TST_byte

```
int PLCC_VME_TST_byte (byte *semaphore_output, unsigned long address);
```

This function reads a byte-sized semaphore from the VME bus address and tests the least significant bit. The semaphore output will be 0 is the semaphore is not obtained, 1 if it is obtained. The user must release this semaphore when it is no longer needed. To release a semaphore, write 0 to the semaphore. This function will return −1 if not supported. For a full explanation of all return values, refer to page 3-51, Table 3-5.

PLCC_VME_TST_word

```
int PLCC_VME_TST_word (word *semaphore_output, unsigned long address);
```

This function reads a word-sized semaphore from the VME bus address and tests the least significant bit. The semaphore output will be 0 is the semaphore is not obtained, 1 if it is obtained. The user must free this semaphore when it is no longer needed. To release a semaphore, write 0 to the semaphore. This function will return −1 if not supported. The address must be word-aligned. For a full explanation of all return values, refer to Table 3-5.

Return Status for VME Byte Functions (Series 90-70 Only)

Each of these VME functions:

PLCC_VME__RMW_byte;
PLCC_VME__RMW_word;
PLCC_VME__TST_byte;
PLCC_VME__TST_word;

have the function return values shown in the following table:

Table 3-5. Return Status for PLCC_VME_RMW_byte Function

Return Status (in hexadecimal)	Meaning
−1	Not supported
0x00FF	Operation successful
All other return values have the following format: 0000 0000 11xx x1xx	 Active bits (failures) are 0, inactive are 1.
bit 0	SYSFAIL* occurred during operation
bit 1	BERR* occurred during operation
bit 2	always 1
bit 3	Bus grant error occurred during operation
bit 4	Bus Hog occurred during operation
bit 5	Bus Interrupt Acknowledge failure
bit 6	always 1
bit 7	always 1

PLCC_SNP_ID

```
int PLCC_SNP_ID (byte request_type, char *id_str_ptr);

request_type
#define READ_ID 0
#define WRITE_ID 1
```

This function will read or write the SNP ID string passed in through id_str_ptr to the PLC. The string should be an eight character buffer (space for seven letters and a NULL

termination). This function returns 1 if successful, 0 if unsuccessful, and −1 if unsupported.

PLCC_read_override

```
int PLCC_read_override (byte tbl_typ, word ref_num, word len,
   byte *data);

tbl_typ
#define I_OVR    0
#define Q_OVR    1
#define M_OVR    2
#define G_OVR    3
#define GA_OVR   4   (Not supported on 90-30 CPUs)
#define GB_OVR   5   (Not supported on 90-30 CPUs)
#define GC_OVR   6   (Not supported on 90-30 CPUs)
#define GD_OVR   7   (Not supported on 90-30 CPUs)
#define GE_OVR   8   (Not supported on 90-30 CPUs)
```

This function will read in the override table for the specified type. The read at the offset must be byte-aligned, that is, **ref_num** must be set to a value from the following series 1, 9, 17, 33,... The length is in bytes. The area pointed to by data must be large enough to hold the amount being read. This function returns:

−0 if successful
−2 bad_memory_type
−3 offset_not_byte_aligned
−4 reading_outside_ref_mem
−5 bad_data_pointer

Section 6: Application Considerations

When creating a C application, it is necessary to keep in mind a few items regarding the target Series 90 PLC:

1. Will floating point arithmetic be required? (Remember, in the 90-30 PLCs, only CPU352 can perform floating point operations.)
2. How big is each of the target PLCs reference memories?
3. Will it become the MAIN program (Series 90-30 only)?
4. Will the block be called from the MAIN ladder block or from some other ladder block?
5. How large is the C application likely to be?
6. What is the scheduling mode for the program (standalone C program or Ladder Diagram program—Series 90-70 only)?

All of these questions must be kept in mind while developing C applications. The following sections provide detail on each of these questions and other questions regarding the creation of C applications.

Application File Names

Application file names are limited to 7 characters. The first character in the filename must be alphabetic. Using file names of seven characters or less will avoid conflicts with Logicmaster 90 when working on C applications.

Floating Point Arithmetic

The C Programmer's Toolkit provides the capability to build C blocks or standalone C programs which implement floating point math either through the use of a math coprocessor (coprocessor required) or through software emulation (not available on 90-30 PLCs). Coprocessor- required C blocks and standalone C programs are created using the rebuilt runtime library **`llibc7.far`**. C blocks and standalone C programs which use software emulation of floating point are created using the rebuilt runtime library **`llibca.far`**. C FBKs may only use floating point if built for coprocessor model CPUs.

The performance of individual floating point math operations is much better when using the coprocessor. If, however, the target CPU is not guaranteed to have a math coprocessor present, then software emulation of floating point math must be selected.

Note

With Series 90-30 PLCs, software emulation of floating point is not supported. Of the 90-30 CPUs, only CPU352 has the floating point math coprocessor; therefore, only the 352 CPUs can perform floating point operations.

Available Reference Data Ranges

When a C application uses an index variable to select an element from PLC reference memory, the value of the index variable should **always** be checked against the size of the target PLCs reference memory. The size of any PLC reference memory can be determined using the corresponding SIZE macro. As an example, consider the following ladder logic rung and sample block, where the value in %P1 is to be used as an index into %R memory and the value at %R[%P1] is to be copied to %P2:

Note

The example shown and discussed above is for a 90-70 C block. The 90-30 block does not have parameters, and %P memory does not exist in the 90-30 CPU.

Figure 3-7. Range Checking Indirect References Using the SIZE Macros

```
/* The value at x1 will be used as an index into */
/* register memory.  The value at %R(x1) will be */
/* copied to y1.                                 */

main(word *x1, int *y1)
{
    /* FIRST -  check X1 & Y1 != NULL */
    /* SECOND - must range check value at x1 to ensure */
    /*          that we will stay within limits of PLC */
    /*           %R reference memory.                  */
   if ((x1 != NULL) && y1 != NULL)) {
       if (*x1 > R_SIZE) return(ERROR);

       /* Range check proved OK ==> go ahead and copy data */
       *y1 = RW(*x1);
       return(OK);
    }
    else return (ERROR);
}
```

In the above example, the index *x1 is compared to R_SIZE. If the target PLC contains 1024 registers, then R_SIZE will evaluate to 1024. If *x1 is greater than 1024 (R_SIZE), the program will return with the status ERROR which indicates that the OK output of the CALL function block should be turned OFF. With *x1 greater than R_SIZE, the C block will return with ERROR status and no attempt is made to index into register memory nor to copy any register memory value to *y1.

Note

The above example is not valid for Series 90-30 PLCs because the 90-30 PLC does not support parameter passing to the main function. Also return values have no meaning for 90-30 PLCs.

Global Variable Initialization

Global variables can be used by C applications running in a Series 90 PLC. Global variables are those which are declared outside of a function, typically outside of and before **main( )**. Both initialized and uninitialized global variables may be used.

```
int    xyz;          /* uninitialized global var */
int    abc = 123;    /* initialized global var */

main() {
    xyz = RW(1);
    RI(2) = ++abc;
    return(OK);
}
```

When a C application is compiled and linked to form an executable (**.EXE**) file, all global variables have a predefined location within the **.EXE** image. If the global variable is declared in the C source to have an initial value, the location in the **.EXE** image for that global variable will contain the initialized value. When a C application is incorporated into a Logicmaster 90 folder and that folder is stored to a Series 90 PLC, the PLC receives an image of the **.EXE** file with space pre-allocated for all global variables and with all initialized global variables already containing their predefined values. Upon receiving the **.EXE** image, the PLC will make a copy of the data portion.

Once the PLC is placed into **RUN** mode, the C application may operate upon any of its global variables. Each of the C application's global variables will retain its value from one sweep to the next sweep and will continue to do so until the PLC goes to **STOP** mode. On the transition from **STOP** mode to **RUN** mode, the PLC will re-initialize all of the C application's global data to those values in the saved copy of global data start values. (Recall that the start values were saved when the folder was stored to the PLC.)

Static Variables

The keyword "static" may be used with either global variables or variables declared inside a function (including **main( )**). These variables will retain their value from sweep to sweep like global data. If a static variable is declared with an initial value, the variable will be initialized on the first execution from store or on transition from **STOP** to **RUN** mode. If a static variable is declared without an initial value, the initial value is undefined and **must** be initialized by the C application.

Note

If C blocks are used multiple times in a ladder, static or global variables may not contain expected data from sweep to sweep. Multiple use blocks must at least receive a unique ID for each call or a unique work area to properly distinguish multiple calls.

Data Retentiveness

All global variables and static variables are either *retentive* or *non-retentive*. Values of retentive data are preserved across both power-cycles (assuming a good battery is attached) and stop-to-run transitions. Non-retentive data is reinitialized on each stop-to-run transitions using values saved when the application was first stored.

All global and static variables, which are given an initial value, will be non-retentive. In general, unitialized global data declared without the keyword "static" will be non-retentive, global data with the keyword "static" will be retentive. Very large data structures (that is, larger than 32 KB) without initialization values, however, will be non-retentive, even though they have no defined initial values. Since non-retentive data requires twice the memory space within the CPU (one for the working copy, and one for the saved copy), large uninitialized data structures should be avoided if memory usage is a concern.

The following examples were done using release 8.00 of the Microsoft C compiler. Different compiler releases may cause different results. The retentive property of a particular public variable (static variables are not listed) can be determined by examining the **.MAP** file, which is made during the compile process. Data located within the BSS section is retentive. Variables in other data sections (e.g., DATA, CONST, FAR_DATA, etc) are non-retentive. Refer to Appendix D for a detailed description of determining the CPU memory usage.

Examples :

```
int    my_var1;           /* non-retentive */

int    my_var2 = 20;      /* non-retentive, reset to 20 on stop-to-run transitions */

static int my_var3;       /* retentive */

static int my_var4 = 12;  /* non-retentive, reset to 12 on stop-to-run transitions */


static char big_array[32766]; /* retentive */

static char bigger_array[32767];  /* non-retentive due to large size */
```

Main( ) Parameter Declaration Errors for Blocks (Series 90-70 Only)

When declaring the parameters to **`main( )`** in a block, the *`type`*, *`order`*, and *`number`* of parameters must match the ladder logic call instruction **exactly**. Use the following ladder logic segment and associated C block as an example:

```
/* This rung of ladder logic calls MATH2 to                    */
/*   add the two integers X1 and X2 and place the sum in Y1    */
/*   and subtract the integer X2 from the integer X1, placing */
/*   the difference in Y2.                                     */
```

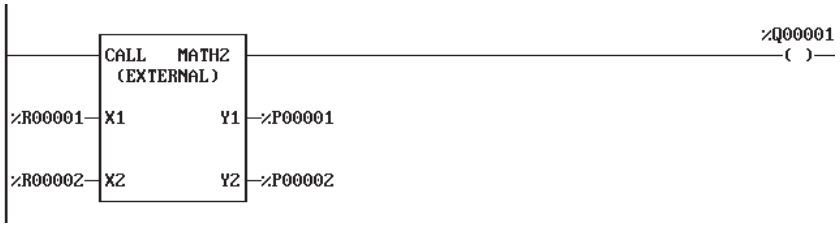


Figure 3-8. Importance of Matching Parameter Type, Order, and Number

Note

The above example is from a Series 90-70 program. Series 90-30 C block do not have parameters. In addition, %L and %P memory types are not available on the 90-30 system.

```
/* MATH2 :
 *      This function has two input parameters and two output
 *      parameters.
 *      Y1 = X1 + X2;
 *      Y2 = X1 - X2;
 */

main( int *x1, int *x2, int *y1, int *y2) {
    if (((x1 != NULL) && (y1 != NULL)) &&
    ((x2 != NULL) && (y2 != NULL))) {
       *y1 = *x1 + *x2;
       *y2 = *x1 - *x2;
       return(OK);
    }
    else return (ERROR);
}
```

As written above, the example is correct; the ladder logic call and the block declaration match. The operation of the ladder logic and the block will proceed as designed.

Type Mismatch Errors (Series 90-70 Only)

If, however, the block declaration is changed to

```
main( float *x1, float *x2, float *y1, float *y2) {
    if (((x1 != NULL) && (y1 != NULL)) &&
    ((x2 != NULL) && (y2 != NULL))) {
       *y1 = *x1 + *x2;
       *y2 = *x1 - *x2;
       return(OK);
    }
    else return (ERROR);
}
```

execution errors will occur. The block will compile and link to an executable without error. The **.EXE** file will add through the Logicmaster 90 Librarian and import to the application folder without error. Similarly, the folder will store to the Series 90-70 PLC without error. No error will appear until the ladder and block are executed. The ladder logic will call **MATH2** passing pointers to two (2) input parameters and pointers to two (2) output parameters. **MATH2** expects two (2) input parameter pointers and two (2) output parameter pointers. The error occurs because the ladder logic uses integer variables (16 bits each), but the block uses float variables (32 bits each). This results in the block using the pointer x1 to read a 32 bit floating point value which starts at %R1 (the value used in the ladder logic). The 32 bit floating point value starting at %R1 includes both %R1 and %R2, but %R2 is the reference specified in ladder logic as x2. Since the input variables overlap, unpredictable values will result from the execution of this block. Notice also that the output parameters will have a similar problem.

Parameter Ordering Errors (Series 90-70 Only)

Execution errors can also occur due to differences in the order of the parameters when calling a block and the order of the parameters in the block declaration of **main( )**. Continuing with the same example, if the ladder logic is unchanged but **main( )** is declared as

```
main ( int *xl, int *yl, int *x2, int *y2) {
  ...
}
```

an execution error will occur. No error message will be generated, just unpredictable output values. The execution error occurs because ladder logic always passes all of the specified input parameters (X1 up to X7), in order, followed by all of the specified output parameters (Y1 up to Y7), also in order. In this case, the ladder logic passes %R1, %R2, %P1, and %P2, the two input parameters followed by the two output parameters. The block associates the parameters from the ladder logic call with its own variable names, as in the following example:

```
int *xl refers to %Rl
int *yl refers to %R2
int *x2 refers to %Pl
int *y2 refers to %P2
```

When the block executes the statement:

```
*y1 = *x1 + *x2;
```

the resulting operation will to add the contents of %R1 (*x1) to the contents of %P1 (*x2) and place the sum in %R2 (*y1), which is not what the ladder logic program expects.

Since the ladder logic call to a block always specifies the parameters in order X1..X7 and Y1..Y7, the block declaration of **main( )** must specify the parameters to **main( )** in the same order.

Parameter Number Errors (Series 90-70 Only)

If the number of parameters associated with a block in ladder logic does not match the number of parameters in the declaration of **`main( )`** for the block, potentially severe execution errors will occur.

Note

It is essential that the number of parameters in a call to a block and the actual number of parameters required by the called block match; otherwise, the block will use invalid pointer variables to perform reads and writes.

Again, using our example with the ladder logic portion unchanged, the effect of a difference in the number of parameters can be illustrated in the following example:

```
main ( int *xl, int *yl) {
/* Add the contents of %Rl to the contents pointed to by xl */
/*  and then store the sum in the location pointed to by yl */
    if ((xl != NULL) && (yl != NULL)) {
       *yl = *xl + RI(l);
        return(OK);
    }
    else return (ERROR);
}
```

In this scenario, the ladder logic call will pass four parameters, %R1, %R2, %P1, and %P2. The block expects two parameters, x1 and y1, which it will associate with the passed in parameters as follows:

```
int *xl refers to %Rl
int *yl refers to %R2
%Pl and %P2 are not referenced
```

The operation of this block with regard to parameter x1 is flawless. However, when y1 is used as the pointer for storing the sum, the sum will be written to %R2, not to %P1. This will cause incorrect operation of the application.

A more severe scenario is a block declared as follows:

```
main (int *x1, int *x2, int *x3, int *y1, int *y2, int *y3) {
/* Add the contents of %Rn to the contents pointed to by xn */
/*  and then store the sum in the location pointed to by yn */
    *y1 = *x1 + RI(1);
    *y2 = *x2 + RI(2);
    *y3 = *x3 + RI(3);
    return(OK);
}
```

The above block can have catastrophic results if executed in conjunction with the example ladder logic rung. Again, the ladder logic call is passing four parameters, a pointer to %R1, a pointer to %R2, a pointer to %P1, and a pointer to %P2. The C program expects six parameters, all pointers. The block will then associate each of the declared parameters to main( ) with the pointers passed from the ladder logic call as follows:

```
int *x1 refers to %R1    /* OK */
int *x2 refers to %R2    /* OK */
int *x3 refers to %P1    /* error - wrong parameter */
int *y1 refers to %P2    /* error - wrong parameter */
int *y2 refers to an unknown value on the PLC stack
int *y3 refers to an unknown value on the PLC stack
```

The unknown values on the PLC stack will be used for y2 and y3 and will cause the C program to write erroneously into PLC memory. The exact location of the write is unpredictable.

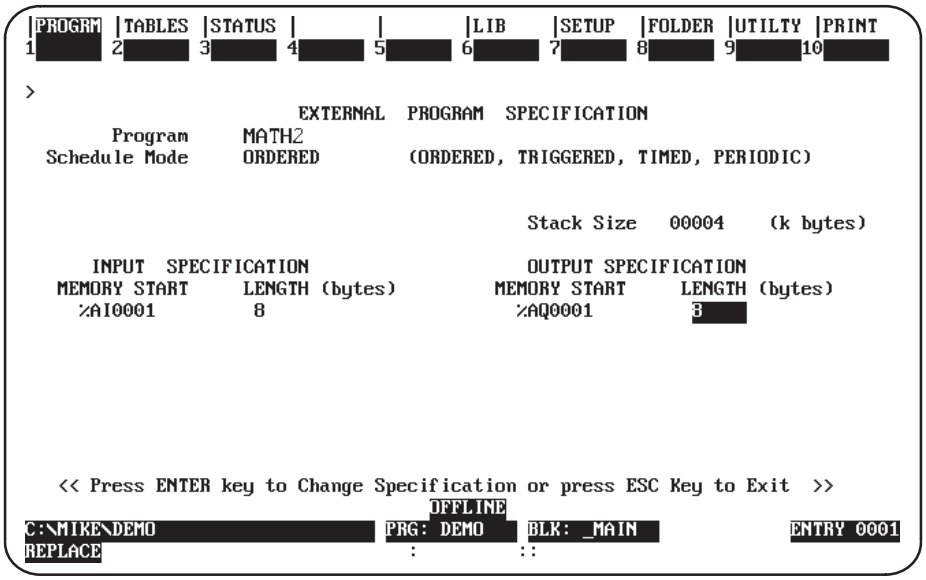
Note

Always verify that the number of parameters expected by a block and the number the ladder logic call will pass to that block are the same. Always verify that the parameters are not NULL pointers before using.

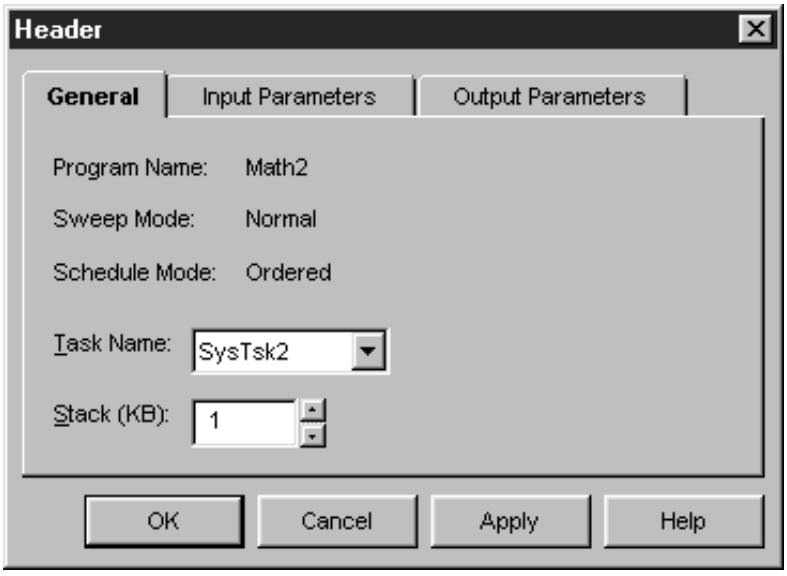
Local I/O Specification Errors (Series 90-70 Only)

When declaring the I/O specifications in the Logicmaster Program Specification Screen, the size and type of each input and output specification must be compatible with the specifications declared in the standalone C program. The following standalone I/O specification is correct:

```
/* MATH2 :
 *      This function has two input parameters and two output
 *      parameters.
 *      Y1 = X1 + X2;
 *      Y2 = X1 - X2;
 */

IN1_F(x,2)

OUT1_F(y,2)

main() {
    y[0] = x[0] + x[1];
    y[1] = x[0] - x[1];
    return(OK);
}
```



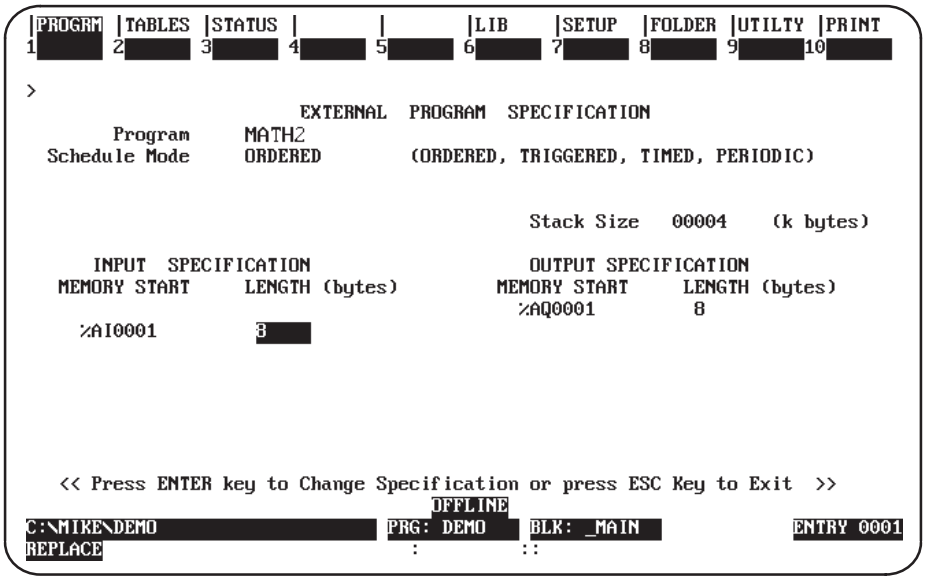
The input array **x** will have valid data at the start of the program and **%AQ0001** will have valid data after the program has finished executing.



In the Windows-based programming software, you enter Program Specification through the IEC Resource functionality. For more information, refer to the "Controlling Program and Block Execution" section of Chapter 6, "Configuring Your Software" in GFK-1295.

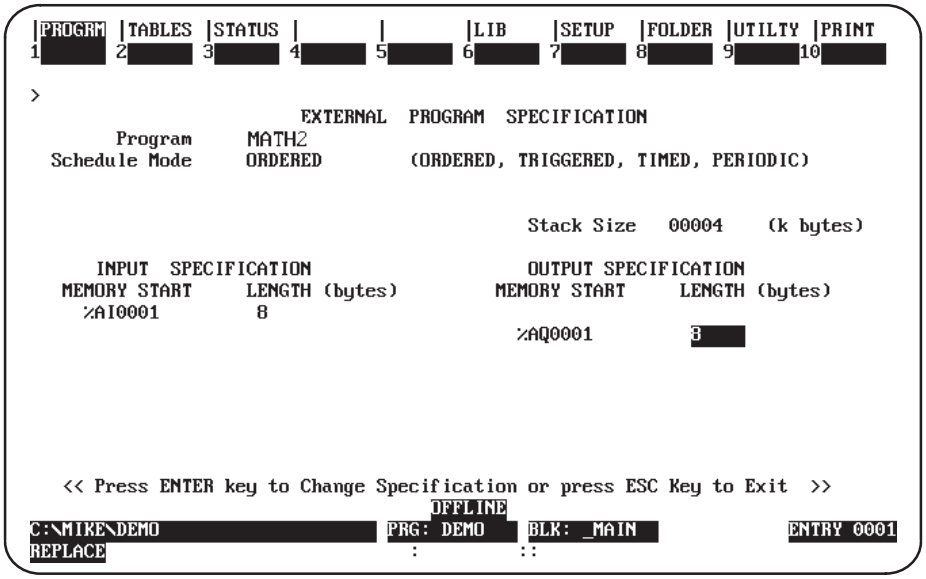
I/O Specification Missing in Logicmaster (Series 90-70 Only)

If the Program Specification is changed as follows,



the input array **x** will have unknown data at the start of the program so there is no way of knowing what **%AQ0001** will have after the program has finished executing.

Similarly, if the Program Specification appears as follows,



the input array **x** will have valid data at the start of the program but there is no way of knowing what **%AQ0001** will have after the program has finished executing.

Incompatible Type

If the standalone C program is changed as follows,

```
/* MATH2 :
 *      This function has two input parameters and two output
 *      parameters.
 *      Y1 = X1 + X2;
 *      Y2 = X1 - X2;
 */

IN1_B(x,2)

OUT1_B(y,2)

main() {
    y[0] = x[0] + x[1];
    y[1] = x[0] - x[1];
    return(OK);
}
```

the input array **x** will have valid data at the start of the program with the rest of **%AI0001** being wasted, but there is no way of knowing what the upper word of **%AQ0001** will have after the program has finished executing.

Uninitialized Pointers

Use of an uninitialized C pointer variable in your C application can cause catastrophic effects on the PLC. It is essential that all pointer variables be correctly initialized prior to use by a C application.

BAD PROGRAM - Uninitialized Pointer

```
main() {
    byte      *bad_ptr;
    int        loop;

    /* Attempt to initialize data area through */
    /* uninitialized pointer.                  */
    for (loop = 0; loop < 10; loop++) {
       *bad_ptr = 0;
    }

    return(OK);
}
```

Warning

All pointer variables in a C application, including those used by Microsoft library functions, must be initialized before they are used, or unpredictable results will occur. The use of an uninitialized pointer may result in the Series 90-70 PLC logging one of the following fatal faults in the PLC fault table and going to `STOP/HALT` mode: "EXE block runtime error: PG_FLT: <block name>" or "STA prg runtime error: PG_FLT: <program name>" where <block name> and <program name> identify of the application that used the uninitialized pointer.

Uninitialized pointers may also result from a C block or FBK user not setting all required parameters. Check parameter pointers for NULL before using.

For the 90-30 system, all pointer variables in a C application, including those used by Microsoft library functions, must be initialized before they are used, or unpredictable results will occur; however, there is no fault message.

PLC Local Registers (%P and %L) — Series 90-70 Only

C Blocks and C FBKs have access to %P and %L PLC reference memory through several macros provided in the file **PLCC9070.H** in the C Programmer's Toolkit. When referencing %P and %L from a block, the following two reference memories appear as two separate tables:

```
main( ) {              /* no parameters to main */
    PW(1) = RW(1);     /* Copy %R1 to %P1       */
    LW(1) = RW(2);     /* Copy %R2 to %L1       */
    return(OK);
}
```

The PLC memory location used as %L or %P is determined by the Series 90-70 PLC at runtime, based on the context from which the block was called. If the block is called from the **MAIN** ladder logic block, then all %L references inside the block will reference the %P table. The %P table and the %L table are the same when a block is called from **MAIN**.

If, however, the same block is called from a ladder logic program block other than **MAIN**, the %P and %L tables will be separate and unique in PLC memory. When the %P and %L tables are separate, all references to %L will affect only the calling block's %L table, and all references to %P will affect only the main program block's %P table.

When called from the **MAIN** ladder logic block, the following block will set %P1 equal to %R1 and then set %L1 equal to %R2:

```
main( ) {              /* no parameters to main */
    PW(1) = RW(1);     /* Copy %R1 to %P1       */
    LW(1) = RW(2);     /* Copy %R2 to %L1       */
    return(OK);
}
```

Since %L1 is actually %P1 in this case, this results in %P1 being set to the value contained in %R2. Again, this is because %P and %L, when used in a block, refer to the same memory table when called from **MAIN** ladder logic block. Conversely, when this same block is called from any ladder sub-block, the result will be that %P1 equals %R1 and that %L1 equals %R2.

Note

Refer to "Blocks as Timed or I/O Interrupt Blocks," for an explanation of %P and %L in interrupt blocks.

Note

The %L and %P macros are not available to standalone C programs.

%P and %L in Ladder Logic (Series 90-70 Only)

The references %P and %L refer to two of the PLC's internal memory tables. Each of these types is word-oriented.

Table 3-6. Descriptions of %P and %L

Type	Description
%P	The prefix %P is used to assign program register references, which will store program data from the main program block. This data can be accessed from all program blocks. The size of the %P data block is based on the highest %P reference in all ladder logic program blocks.
%L	The prefix %L is used to assign local register references, which will store data unique to a ladder logic program block. The size of the %L data block is based upon the highest %L reference in the associated ladder logic program block.

Both %P and %L user references have a scope associated with them. Each of these references may be available throughout the logic program, or access to these references may be limited to a single ladder logic program block.

Table 3-7. Data Scope of %P and %L

User Reference	Range	Scope
%P	Program	Accessible from any program block.
%L	Local	Accessible from within a ladder logic block. Also accessible from any external block called by the ladder logic block.

In a program block, %P should be used for program references which will be shared with other program blocks. %L are local references which can be used to restrict the use of register data to that ladder logic program block and any external block called by that ladder logic block. These references are not available to any other parts of the program.

Note

The %L and %P macros are not available to standalone C programs.

Block OK Output (Applicable to Series 90-70 Only)

In ladder logic, the function block CALL, when used with a block as the target, provides a boolean OK output. This OK output (called ENO in the Windows-based programming software) from the call is under the direct control of the block.

The OK output (ENO in the Windows-based programming software) is controlled by the return value from **`main( )`**. If **`main( )`** returns a value of zero (0), the OK output is turned ON (1). If, however, **`main( )`** returns a value which is non-zero (any value from 1 to 255), the CALL function block OK output is turned OFF (0). The OK output is defined in this manner to mimic the handling of a return status from **`main( )`** for **`.EXE`** files running under MS-DOS. The key difference is that, under MS-DOS, the exact value of the return status is accessible; and only the indication of OK or ERROR is available. (The C symbols "OK" and "ERROR" are defined in the toolkit file **`PLCC9070.H`**.)

Note

The 90-30 system always returns ENO (i.e., powerflow is always passed after the call, even if the C subroutine block returns "ERROR").

Standalone C Program Return Value (Series 90-70 Only)

If the return value from a standalone C program is anything but OK, then the local output specification for the standalone C program does not get copied to the PLC CPU. This occurs when the C code does not return OK, or when there is a problem and the standalone C program exits with an error condition. The standalone C program will exit with an error condition if there is a problem in the run-time library, or if the PLC detects a problem during the execution of the program. In all error conditions a fault will be logged.

Writes to %S Memory Using SB(x)

The %S table is for the PLC to provide status on its operation. This table is intended to be written only by the CPU firmware; therefore, it is also intended to be read-only from elsewhere in the system, specifically from the application program. Attempting to use the SB(x) macro to write into %S memory will result in a compile error when compiling the application C source file. Similarly, attempting to use the pointer variable **`sb_mem`** (provided in **`PLCC9070.H`** or **`PLCC9030.H`** and the same pointer variable used by the SB(x) macro) will result in the same compile error.

FST_EXE (Series 90-70 Only) and FST_SCN Macros

In the file **`PLCC9070.H`** (provided in the 90-70 C Programmer's Toolkit), there are two macros, **`FST_SCN`** and **`FST_EXE`**, which provide blocks with direct access to %S0001 (system first scan indication) and with direct access to %S0121 (block first execution). The **`FST_SCN`** macro references %S0001 and acts exactly like the ladder logic reference **`FST_SCN`** (%S0001). If a block is not called on the first PLC sweep, the macro **`FST_SCN`** should not be used for initializing data in the block. In this case, **`FST_SCN`** would never be true. The standalone C program has a **`FST_SCN`** macro but not a **`FST_EXE`** macro. For the standalone C program, **`FST_SCN`** is 1 for the first execution after a STOP−> RUN transition or a RUN-MODE-STORE.

The **FST_EXE** macro operates differently than the **FST_SCN** macro. There is no system status bit associated with the first call to blocks. A block inherits **FST_EXE** from the block which calls it. Therefore, if **FST_EXE** in the calling ladder logic program is true, when the block is executed, the C macro **FST_EXE** will also be true. The value of **FST_EXE** is determined by the calling ladder logic block, **not** by the block. **FST_EXE** may be TRUE (1) if the block is called multiple times from one ladder logic block or is called from multiple ladder logic blocks. If the call from the ladder logic to the block is conditional, it is possible that the block may **never** see **FST_EXE** as true.

Note

You can find the FST_SCN macro for a 90-30 application in **PLCC9030.H** (provided in the 90-30 C Programmer's Toolkit), but the FST_EXE macro is not supported for a 90-30 program.

LST_SCN Macro (Series 90-30 Only)

The **LST_SCN** macro (located in the file **PLCC9030.h**) provides access to the %S0002 (system last scan indication) bit. The **LST_SCN** macro behaves exactly like the ladder logic reference **LST_SCN**. This bit is a zero (0) on the final scan before the PLC enters STOP/NO IO mode. It is a one (1) all other times. If a C subroutine is not called on the last scan before a PLC enters STOP/NO IO mode, the **LST_SCN** macro should not be used in that block to capture data or trigger events on the last scan. In such a case, the data or events would never be triggered because the C subroutine was not called on the last scan.

The **LST_SCN** macro is only available on 90-30 PLCs. For more information, refer to the Chapter 2 of the *Series 90-30/20/Micro Reference Manual* (GFK-0467) or Chapter 2 of the *Series 90-30 System Manual* (GFK-1411).

Note

LST_SCN is not available for a power-down situation.

Runtime Error Handling

When a C application executes in a Series 90 PLC, if an error is generated from one of the runtime library functions or from incorrect interaction between the C application and the Series 90 CPU, the error will be detected and logged in the PLC fault table as an application fault on the CPU (rack 0, slot 1). Examples of such errors include, but are not limited to the following:

1. Integer divide by 0
2. Stack overflow
3. Floating point divide by 0
4. **Printf( )** to serial port when serial port is not configured for **MSG** mode (90-70 Only)
5. Floating point overflow
6. Floating point underflow

When a runtime error is logged into the PLC fault table, the fault will contain the name of the offending application and a text message describing the error. The descriptive text

message (for runtime library function errors) is the same text that would be displayed if the error had occurred while the program was executing under MS-DOS; however, since PLC fault table entries are limited in size, the text message in a C program fault may be truncated.

An example of a runtime error and the resulting PLC fault is illustrated in the following C application, **DV0.C**:

```
Series 90-70 Example
#include "plcc9070.h"

main() {
    int x=3, y=0;

    return(x/y);
}
```

```
Series 90-30 Example
#include "plcc9030.h"

EXE_stack_size=2048;

main() {
    int x=3, y=0;

    return(x/y);
}
```

When **DV0** (assume that the block name in the PLC is also DV0) executes in the PLC, an integer divide by zero runtime error is generated (y=0). The message typically displayed under DOS appears as follows:

```
run-time error R6003
- integer divide by 0
```

The faults logged in the Series 90 CPU and displayed by Logicmaster 90 or the Windows-based programming software appears as follows:

```
EXE block runtime error:DV0:R6003-integer div
 STA prg runtime error:DV0:R6003-integer div
```

Note

Attempting to execute a C application built to use the math coprocessor on a Series 90 CPU which does not contain a coprocessor will cause the following: a "No coprocessor" fault will be logged in the PLC fault table, the C application will **not** execute, and an error status will be returned to the calling ladder logic.

Note

Most floating point runtime errors (also known as floating point exceptions) result in two faults being logged in the PLC fault table. Both faults are the result of a single error. The text in the two faults is different; the text in the second fault is a continuation of the text in the first. For example, if a block named **FNC7** generated a floating point divide by 0 exception, then the single error would cause the following two faults:

```
EXE block runtime error:FNC7:M6103:MATH - float"
EXE block runtime error:FNC7:divide by 0
```

C Application Size Under MS-DOS

Series 90-70 Size Limitations

C applications are based on a PLC-executable (.EXE) version of a user's C application source. For blocks, no PLC-executable .EXE file may be larger than 64,000 bytes, as displayed by the MS-DOS directory (**`dir`**) command. For a 90-70 folder, C blocks cannot be larger than 64,000 bytes to be added to the Logicmaster 90 library. In Logicmaster, Standalone C programs have no .EXE file to inspect. Importing a standalone C program will print the size of the executable file for examination. This size cannot exceed 512 KB. When using the Windows-based programming software with a 90-70 folder, the same size limitations remain. If the file has been built for debugging, the .EXE will have symbol information which will not be loaded to the PLC (please see Chapter 7. "C Application Debugger" for more details).

Note

A Series 90-70 PLC block with an .EXE file size of less than 64,000 bytes may still be too large for the Logicmaster 90 Librarian to add the block to its library. As part of the header information of every MS-DOS .EXE file (.EXE files for MS-DOS Versions 2.0 and later) there is a field which indicates the number of additional 16-byte memory areas, called paragraphs, which are required by the block during execution. In this manual, the field which indicates the number of additional paragraphs required is referred to as the ***`MINALLOC`*** field. When attempting to add an .EXE file to the Logicmaster 90 library, the Librarian checks that the MS-DOS directory size plus the additional area required during execution is less than or equal to 64,000 bytes. This also applies to standalone C programs when they are being inserted into folders through Logicmaster or the Windows-based programming software.

Series 90-30 Size Limitations

C applications are based on a PLC-executable (.EXE) version of a user's C application source. For 90-30 folders, the total PLC memory available for C Main programs or C subroutine blocks is 80 kilobytes.

C Application Impact on PLC Memory

As displayed under MS-DOS, the size of a PLC-executable file is the application base. When the C application is stored to the CPU, the CPU must allocate more memory than merely the MS-DOS size of the PLC-executable file. The additional space allocated by the CPU is required for the following reasons:

1. To save the initial values of all C application global data
2. To maintain pertinent information regarding the C application (internal processing overhead)
3. To create additional space for the execution of the C application (if the ***`MINALLOC`*** field in the .EXE file header is not 0)
4. To create a stack from the C subroutine block (using EXE_stack_size)—Series 90-30 only

When using Logicmaster, the standalone C program has no .EXE file to view. Importing a standalone C program will print the size of the executable file. To calculate the PLC memory usage for a C application, refer to Appendix D, "Calculating PLC Memory Usage for a C Application."

Blocks as Timed or I/O Interrupt Blocks (Series 90-70 Only)

Blocks may be used in the PLC as the target of a timed or I/O interrupt with the following restrictions.

1. A block invoked as the result of a timed or I/O interrupt may not have parameters associated with the call. The block must have 0 input parameters and 0 output parameters. A block invoked as a subblock of a timed or I/O interrupt may have parameters associated with the call.
2. A C FBK may be invoked as a subblock of main and be associated with a timed or I/O interrupt (either an interrupt or a subblock of an interrupt). A C block invoked as the result of a timed or I/O interrupt may not also be called from ladder logic.
3. A block invoked as a timed or I/O interrupt is guaranteed to have 896 bytes of available stack. A block invoked as a subblock of a timed or I/O interrupt has the same stack as a block invoked as a subblock of main.
4. When a block is invoked as a timed or I/O interrupt, all references to %L memory will reference the same location in the %P table. (This action is the same as when a block is called directly from the **MAIN** ladder logic program.) When a block is invoked as the subblock of a timed or I/O interrupt block, all references to %L memory will be references to the %L of the block from which they were called.
5. Additional interrupts are not processed while a timed or I/O interrupt blocks and associated subblocks are executing.

The following example and associated text covers the issues related to using C Blocks vs. C FBKs when the same C application is going to be called during the normal execution of the LD or SFC program AND from a possible timed or I/O interrupt.

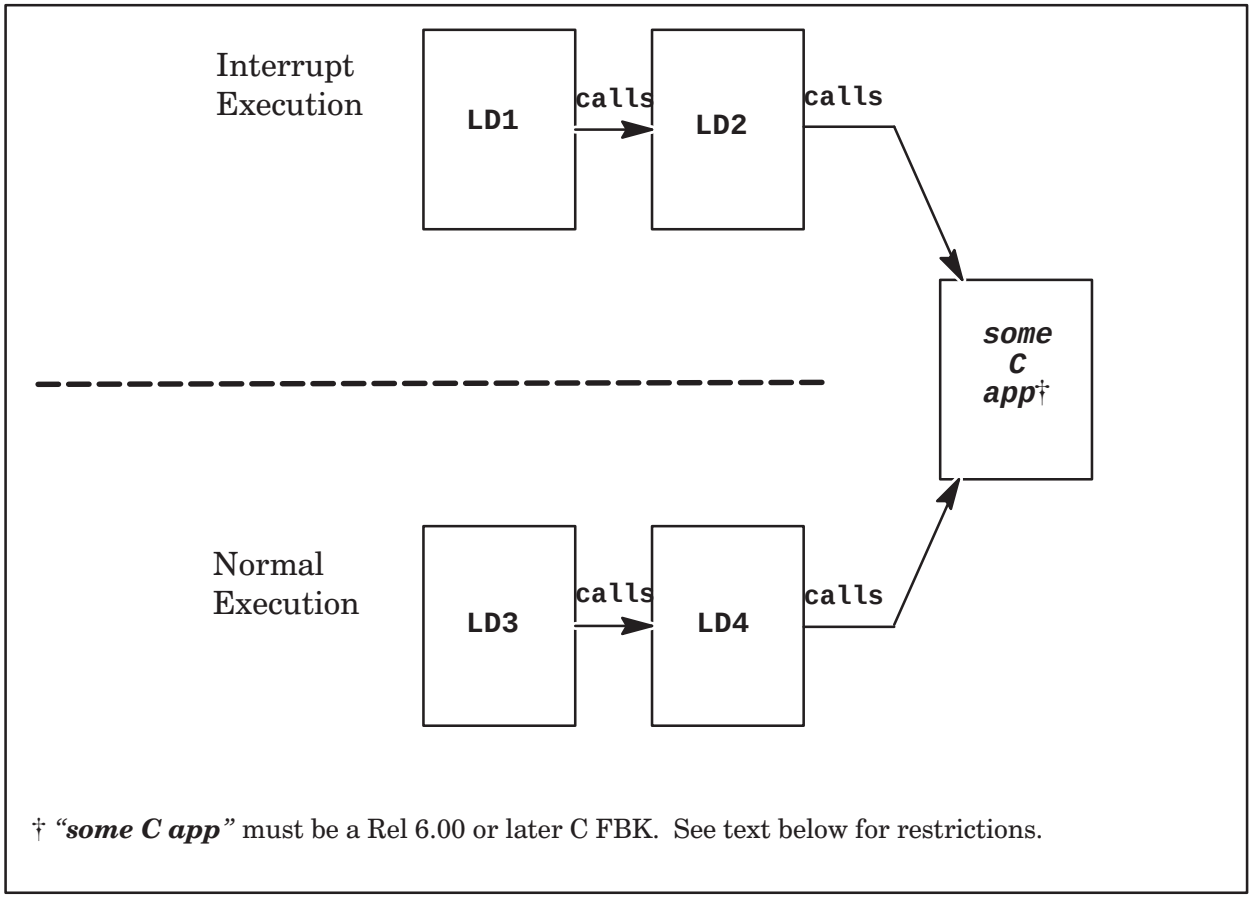


Figure 3-9. Interrupt Block Calls and C Blocks/FBKs

In the example shown in Figure 3-9. two separate execution paths are depicted: normal execution and interrupt execution. Normal execution is initiated through the standard sweep mechanism of the operating system calling the _MAIN LD/SFC block. Then through a series of calls to LD sub-blocks, the example eventually calls "*some C app*". Interrupt execution is initiated by either a timed event of by an interrupt event (interrupt input) coming into the 90-70 PLC causing the operating system to invoke a block. Beginning in Rel 6.00, calls can be made from an LD block to other blocks (LD, C block, or C FBK). Note that calling a C block or C FBK terminates the call chain.

The example in Figure 3-8 shows that both the Normal Execution path ***AND*** the Interrupt Execution path calling (through a chain of different LD blocks) the same block. In order for this example to work correctly, this block ***MUST*** *be a Rel 6.00 vintage C FBK*. Furthermore, a set of guidelines MUST be followed by the C FBK developer so that execution of the block can be guaranteed.

All versions of 90-70 CPUs that support blocks (release 4 and later) and all versions of the C Programmer's Toolkit (IC641SWP709 and IC641SWP719) for Series 90-70 PLC have clearly stated that the same C block cannot be invoked by way of normal and interrupt execution paths. This restriction exists because the startup code in C blocks and code in Microsoft runtime libraries is not re-entrant.

A C FBK should be used because a C FBK has re-entrant startup code and permits NO CALLS to runtime library functions. This allows for a minimal amount of overhead and code size. C FBKs are also designed to allow the user to create re-entrant code if certain guidelines are followed. These guidelines are described in the following section.

A C FBK developer should use the following guidelines to ensure the success of a situation such as the one illustrated in Figure 3-9.

1. Only code written by the developer should be part of the C FBK (no PLCC_* calls and no runtime library calls). This is the standard definition of a C FBK.
2. All variables used by the C FBK should be stack-based (automatic) variables.
3. If there is any portion of the C FBK which operates on PLC global memories (%R, %P, ...etc.), the C FBK must contain additional code to handle some sort of hand-shaking between normal executions and interrupt executions to prevent data incoherency. The hand-shaking could be accomplished by declaring a global flag (variable) in the C FBK (or using an application-reserved location in PLC global memory) that the C FBK sets just prior to writing to the PLC global memories and then clears when the update is complete. Execution of the block (regardless of normal or interrupt) should read the global flag before changing the PLC global memory. If the flag is set, the C FBK should not change the PLC global memory.

Standalone C Programs Scheduled as Timed or Triggered Interrupts (Series 90-70 Only)

Standalone C programs may be scheduled in the PLC as timed or triggered interrupts with the following qualifications:

1. A standalone C program scheduled as an interrupt may have I/O specifications.
2. A standalone C program scheduled as an interrupt may be be assigned a priority.
3. Standalone C programs may be interrupted by programs of a higher priority and by any timed or I/O interrupt block.

Program Scheduling Mode (Series 90-70 Only)

It is important to understand the scheduling mode for programs because the scheduling mode of the program will affect the operation of the application. Please refer to Chapter 2 of the *Series 90-70 Reference Manual* (GFK-0265G) or Chapter 2 of the *Series 90-70 System Manual* (GFK-1192) for further discussion.

Scan Impact

The following table provides the scan impact of executing a single CALL external block function block. The times listed are the sum of the entry overhead time and the exit overhead time. The entry overhead time is measured from the start of the ladder logic CALL function block to the first instruction to be executed in the C program main( ). The exit overhead time is measured from the first instruction after returning from the C program main( ) to the next ladder logic instruction.

The first execution times reflect the scan impact the first time the block is called. The normal execution times are for all calls **except** the first call.

Table 3-8. Scan Impact of Executing a Call to an External Block *

CALL External Block - Scan Impact				
			Entry Time + Exit Time (microseconds)	
CPU Model	CPU Clock Speed	Floating Point	First Execution	Normal Execution
73x/77x	12 MHz	None	318	289
		Alternate Math	379	295
		Coprocessor	2088	446
78x	16 MHz	None	315	270
		Alternate Math	447	377
		Coprocessor	2328	453
91X	32 MHz	None	97	86
		Alternate Math	140	122
		Coprocessor	618	133
92X	64 MHz	None	67	51
		Alternate Math	94	84
		Coprocessor	339	82
90-30† 351	25 MHz	None	366	312
		Alternate Math	Not Supported	Not Supported
		Coprocessor	N/A	N/A
90-30† 352	25 MHz	None	366	312
		Alternate Math	Not Supported	Not Supported
		Coprocessor	2358	442

* The times given in this table are for C blocks that have no parameters. For those blocks which do have parameters, add 14 microseconds/parameter pair for a 73x/77x CPU, 6 microseconds/parameter pair for a 78x CPU, 2 microseconds/parameter pair for a 914/915 CPU, and 1 microsecond/parameter pair for a 924/925 CPU.

† The times given for 90-30 CPUs are for C subroutine blocks. For C Main programs, subtract 12 microseconds.

Note

The scan impact for calling an external (C) block on an IC697CPU731N or earlier or an IC697CPU771L or earlier will be somewhat slower than those listed in the previous table.

The following table provides the overhead time for a standalone C program. The times listed include the call overhead and overhead for each I/O specification.

Table 3-9. Overhead Time for a Standalone C Program (Series 90-70 Only)

Item Impacting Sweep	Sweep Impact (all times in microseconds)		
	781/782 (16MHz)	914/915 (32MHz)	924/925 (64MHz)
No Floating Point Math			
Call overhead	317	116	74
Overhead for first local input spec (1 byte)	3	3	2
Overhead for each additional local input spec (1 byte)	6	2	2
Overhead for first local output spec (1 byte)	3	3	2
Overhead for each additional local output spec (1 byte)	6	2	2
Overhead for each additional byte in a local I/O spec	1	1	1
Floating Point Math			
First scan call overhead for Alt FP	600	200	100
First scan call overhead for 80x87 coproc FP math	1900	500	300
Non-first scan call overhead for Alt FP	340	118	74
Non-first scan call overhead for 80x87 coproc FP math	345	182	125

Note

All local I/O specifications above use %I as a local input specifications and %Q as local output specifications. Using %M, %T, and %S will result in the same execution times. Should %R, %AI, or %AQ be used, the resulting sweep impact (per byte of local I/O spec) will be less than listed. The use of %P and %L is not permitted with standalone programs.

Section 7: Testing C Applications in the MS-DOS Environment

It is highly recommended that all C applications be tested prior to execution in a Series 90 PLC. This is best accomplished by testing the application within the MS-DOS environment.

If you compile and link a C application to form a DOS-executable **.EXE** file and if you invoke the resulting **.EXE** file from the MS-DOS prompt, MS-DOS will execute the **EXE** file once. When **main( )** in the C application returns, MS-DOS will exit back to the MS-DOS prompt. Since repeating these steps over and over in order to test different input conditions to a C application is very cumbersome, the C Programmer's Toolkit provides three test harnesses to aid in debugging the C application.

Note

In order to facilitate debugging with **printf( )**, runtime libraries have been enabled for C FBKs. All runtime library calls (both explicitly placed in the code by the user and implicitly placed by the compiler) are not allowed for C FBKs under the PLC, but may not be detected during the MS-DOS debugging.

Test Harnesses

Each test harness consists of one file that is compiled and linked with the user's C application. The harness provides the PLC sweep mechanism-a very basic PLC sweep-while operating under MS-DOS. The harness may be copied and modified to suit each C application that is to be tested.

The only requirements for modifying a harness are as follows:

1. The name of the test harness file must always be **BLKHARN.C, FBKHARN.C,** or **SAPHARN.C** (for C blocks, C FBKs, or standalone C programs respectively).
2. The name of the function declared in the harness must be **plc_sweep( )**.

Each version of a block harness that is created may be tailored to send the correct type, ordering, and number of parameters to the **main( )** function in the block being tested.

The harness also provides for the allocation of all PLC reference memory tables. So even though the C application is operating under MS-DOS and not in the Series 90-70 PLC, there is still a %R table, %I table, %Q table, etc... , all of which can be referenced from the harness or from the C application. The size of each reference table may also be user-selected by modifying the #define statements at the top of the harness.

The versions of harnesses provided in the C Programmer's Toolkit directory **\s9070c** or **\s9030c** are generic test harnesses. The block versions call the function **main( )** passing seven input and seven output parameters. Also in these test harnesses is control of **FST_SCN** and **FST_EXE** so that if the C application uses **FST_EXE** and/or **FST_SCN**, the values of the two %S references will be correct. The standalone C program version copies the input specification (taken from the user), calls the function **main( )**, and then copies the output specification. The harness also control the **FST_SCN** for the standalone C program.

Also provided as part of the C Programmer's Toolkit are five example programs. Two of these example programs have a modified version of **BLKHARN.C**. The modifications to each of the example **BLKHARN.C** files are designed to enable testing of the target C application in different ways. Example 1 is interactive in its receiving and displaying of application information. Example 2 uses an input data file for values to pass to the C application and places the results returned from the C application in an output data file. The other three examples cover C FBKs, standalone C programs, and multiple sources.

BLDVARS File

The Toolkit uses the file BLDVARS to define the type of application being built and whether debug is included. If this file does not exist in the directory with the source files, the build procedure will copy the **BLDVARS** file from the **\S9070C** (Series 90-70) or **\S9030C** (Series 90-30) directory. The default **BLDVARS** file is shown below:

```
#########################################################################
#
#  BLDVARS
#  Copyright (c) 1994, GE Fanuc Automation North America, Inc.
#                  All rights reserved.
#
#  This file defines build parameters.
#
#########################################################################

# debug information on or off
# ON include debug information in the build
#  for the debugger
# OFF do not include the debug information in
#  the build
DEBUG=OFF
# type of build (C exe block, C fbk, standalone C program)
# BLOCK      C exe block
# FBK        C fbk
# STANDALONE  standalone C program
TYPE=BLOCK
```

The default **BLDVARS** file designates a C block build with no debug information. To allow for debug information (which is required for the debugger under the PLC), change the line in **BLDVARS** that is currently **DEBUG=OFF** to **DEBUG=ON** (note that ON must be in all capital letters). To change the application type that is built, change the type field from **TYPE=BLOCK** to either **TYPE=FBK** or **TYPE=STANDALONE** (once again in all capital letters). If you wish to change these variable strictly for the files in one directory make the changes to the **BLDVARS** file in that directory.

Note

(Series 90-30 only) For the Series 90-30, DEBUG must always be set to OFF; TYPE must always be set to BLOCK. They are the only ones supported for the 90-30.

Building for MS-DOS Execution (Series 90-70)

Once the application is written, a copy of the appropriate harness has been customized for the C application, and **BLDVARS** has been modified, it is time to build the executable file (**.EXE**). As part of the C Programmer's Toolkit, several MS-DOS batch files (.BAT) are installed in the **\s9070c** directory. The **MKDOS.BAT** and **MKDOS7.BAT** files are used to build a C application for execution under MS-DOS. Both of these **.BAT** files will create an MS-DOS executable from a C application source file and harness. The difference between **MKDOS7** and **MKDOS** is in the implementation of floating point arithmetic. If the MS-DOS machine that will be used to test this C application contains a math coprocessor and the eventual target PLC for this C application will also contain a math coprocessor, then **MKDOS7.BAT** should be used to build the executable. If either the MS-DOS machine used for testing or the eventual target PLC will **not** contain a math coprocessor, then use **MKDOS.BAT** to build the executable.

Note

The harness file must reside in the same MS-DOS directory as your C application program source file. If the harness is not in the same directory as your C application source file, then an error message stating that the harness could not be found will be displayed and no executable will be created.

The MS-DOS syntax for using **MKDOS** is:

```
c:\apps> MKDOS  <application filename>
```

Similarly, the syntax for **MKDOS7** is:

```
c:\apps> MKDOS7 <application filename>
```

In the following example, assume the C source code is in the file **MATH.C**:

```
/* MATH :
 *     This function has two input parameters and two output
 *     parameters.
 *     Y1 = X1 + X2;
 *     Y2 = X1 - X2;
 */

main( int *x1, int *x2, int *y1, int *y2) {
    if (((x1 != NULL) && (y1 != NULL)) &&
    ((x2 != NULL) && (y2 != NULL))) {
       *y1 = *x1 + *x2;
       *y2 = *x1 - *x2;
       return(OK);
    }
    else return (ERROR);
}
```

To build an **.EXE** file, for execution under MS-DOS, invoke **MKDOS.BAT** specifying the application filename MATH as a parameter:

```
c:\apps> MKDOS MATH
```

If the example file **MATH.C** contained source code which used C types like **float** or **double** AND the target MS-DOS machine and target PLC both had a math coprocessor, then MATH.C could be built using **MKDOS7**:

```
c:\apps> MKDOS7 MATH
```

Invoking either **MKDOS** or **MKDOS7** and specifying the application source code filename will cause the following:

1. The creation of a subdirectory named DOS under the current directory
2. The compilation of the application source file creating an output object file (**.OBJ**) in the DOS subdirectory
3. The compilation of the harness creating an output object file (**.OBJ**) in the DOS subdirectory
4. The linking of these two modules with the required runtime libraries to create an MS-DOS-executable output file (**.EXE**) in the DOS subdirectory

The MS-DOS executable file will have the same name as the C application source file. In the previous example, using **MATH.C**, the MS-DOS executable would be **MATH.EXE**.

Note

When the MS-DOS command file **MKDOS7.BAT** is used to create the MS-DOS executable file, a coprocessor is required at execution time.

The MS-DOS batch files **MKDOS.BAT** and **MKDOS7.BAT** and the makefile (**PLCC9070.MAK**) invoked by each of these batch files are designed to support either a single application source file or multiple sources files. To learn more about multiple sources files refer to Chapter 6, "C Application Development Using Multiple C Source Files."

Note

The build procedures for the Toolkit are conditional builds. This means that the application will be rebuilt only if the source files (**.C**) are newer than the object files (**.OBJ**). Therefore, if you are switching from a build that requires a coprocessor to one that doesn't, the object files (**.OBJ**) need to be deleted.

Debugging under MS-DOS

If MS-DOS testing of a C application uncovers bugs in the C application source file, use one of the following approaches to debug the file:

1. **`printf( )`** debugging
2. Microsoft **`CodeView`** debugging

printf( ) Debugging

Using **`printf( )`** debugging involves inserting C **`printf( )`** statements at key points within the C application. As the application executes under MS-DOS, each of the **`printf( )`** statements encountered will cause its text string to be printed to the screen. By following the printed text string, you can determine the path taken through the software. The number or placement of any **`printf( )`** statement is completely up to the developer.

However, there are a few drawbacks to using **`printf( )`**:

1. If the C application is not a simple program, the number of **`printf( )`** statements required to follow the execution path may be quite large.
2. If there are a large number of **`printf( )`** statements inserted into the C application, the compile time and amount of generated code will be greatly increased.
3. With a large number of **`printf( )`** statements, the corresponding number of text strings displayed to the screen may make following the program's execution path difficult.
4. Debugging using **printf( )** is available to C FBKs when operating in the MS-DOS environment.

Because of these drawbacks to using **`printf( )`** debugging, Microsoft **`CodeView`** debugging is preferred for all but the simplest C applications.

Microsoft CodeView Debugging

The **`.EXE`** file created when either **`MKDOS.BAT`** or **`MKDOS7.BAT`** is used contains the symbol, line number, and debug information necessary for using the Microsoft **`CodeView`** debugger. Within **`CodeView`**, variables may be examined, variables may be modified, breakpoints can be set, input can be entered, and output can be observed − all without adding numerous **`printf( )`** statements.

While in **CodeView**, the MS-DOS versions of PLC reference memory tables (sized and allocated through the harness) can be accessed using the names described in the following table:

Table 3-10. MS-DOS Versions of 90-70 PLC Reference Memories

Memory Type	PLC Reference	CodeView Symbol (case sensitive)
Registers	%R	_R_TBL
Analog Inputs	%AI	_AI_TBL
Analog Outputs	%AQ	_AQ_TBL
Inputs	%I	_I_TBL
Outputs	%Q	_Q_TBL
Local Registers	†%L †%P	_L_TBL _P_TBL
Temporary Coils	%T	_T_TBL
Internal Coils	%M	_M_TBL
System Status	%S %SA %SB %SC	_S_TBL _SA_TBL _SB_TBL _SC_TBL
Genius	%G %GA %GB %GC %GD %GE	_G_TBL _GA_TBL _GB_TBL _GC_TBL _GD_TBL _GE_TBL
Transition Memory		_I_TRANS_TBL _Q_TRANS_TBL _T_TRANS_TBL _M_TRANS_TBL _S_TRANS_TBL _SA_TRANS_TBL _SB_TRANS_TBL _SC_TRANS_TBL _G_TRANS_TBL _GA_TRANS_TBL _GB_TRANS_TBL _GC_TRANS_TBL _GD_TRANS_TBL _GE_TRANS_TBL
Diagnostic Memory		_I_DIAG_TBL _Q_DIAG_TBL _AI_DIAG_TBL _AQ_DIAG_TBL
Fault Memory		_RSB_TBL

† not available for standalone C programs

Using the symbol names in the previous table, any of the MS-DOS versions of the 90-70 PLC reference table may be initialized, viewed, or modified during execution.

For more information on Microsoft **CodeView**, please consult the documentation provided with your Microsoft C Compiler.

Building for MS-DOS Execution (Series 90-30)

Once the application is written, a copy of the appropriate harness has been customized for the C application, and **BLDVARS** has been modified, it is time to build the executable file (**.EXE**). As part of the C Programmer's Toolkit, several MS-DOS batch files (.BAT) are installed in the **\s9030c** directory. The **MK3DOS.BAT** and **MK3DOS7.BAT** files are used to build a C application for execution under MS-DOS. Both of these **.BAT** files will create an MS-DOS executable from a C application source file and harness. The difference between **MK3DOS7** and **MK3DOS** is in the implementation of floating point arithmetic. If the MS-DOS machine that will be used to test this C application contains a math coprocessor and the eventual target PLC for this C application will also contain a math coprocessor, then **MK3DOS7.BAT** should be used to build the executable. If either the MS-DOS machine used for testing or the eventual target PLC will **not** contain a math coprocessor, then use **MK3DOS.BAT** to build the executable.

Note

The harness file must reside in the same MS-DOS directory as your C application program source file. If the harness is not in the same directory as your C application source file, then an error message stating that the harness could not be found will be displayed and no executable will be created.

The MS-DOS syntax for using **MK3DOS** is:

```
c:\apps> MK3DOS  <application filename>
```

Similarly, the syntax for **MK3DOS7** is:

```
c:\apps> MK3DOS7 <application filename>
```

In the following example, assume the C source code is in the file **MATH.C**:

```
#include "plcc9030.h"

EXE_stack_size = 2048;

/* * MATH :
 * This function has two input parameters and two ouput parameters.
 * Input parameter one two will be located in %R0001 and %R0002
 * respectfully. The output parameters will be located in
 * %AQ0001 and %AQ0002.
 *
 * %AQ0001 = %R0001 + %R0002;
 * %AQ0002 = %R0001 - %R0002;
 *
 */main()
{
 AQI(1) = RI(1) + RI(2);
 AQI(2) = RI(1) - RI(2);
 return(OK);
}
```

To build an **.EXE** file, for execution under MS-DOS, invoke **MK3DOS.BAT** specifying the application filename MATH as a parameter:

```
c:\apps> MK3DOS MATH
```

If the example file **MATH.C** contained source code which used C types like **float** or **double** AND the target MS-DOS machine and target PLC both had a math coprocessor, then MATH.C could be built using **MK3DOS7**:

```
c:\apps> MK3DOS7 MATH
```

Invoking either **MK3DOS** or **MK3DOS7** and specifying the application source code filename will cause the following:

1. The creation of a subdirectory named DOS under the current directory
2. The compilation of the application source file creating an output object file (**.OBJ**) in the DOS subdirectory
3. The compilation of the harness creating an output object file (**.OBJ**) in the DOS subdirectory
4. The linking of these two modules with the required runtime libraries to create an MS-DOS-executable output file (**.EXE**) in the DOS subdirectory

The MS-DOS executable file will have the same name as the C application source file. In the previous example, using **MATH.C**, the MS-DOS executable would be **MATH.EXE**.

Note

When the MS-DOS command file **MK3DOS7.BAT** is used to create the MS-DOS executable file, a coprocessor is required at execution time.

The MS-DOS batch files **MK3DOS.BAT** and **MK3DOS7.BAT** and the makefile (**PLCC9030.MAK**) invoked by each of these batch files are designed to support either a single application source file or multiple sources files. To learn more about multiple sources files refer to Chapter 6, "C Application Development Using Multiple C Source Files."

Note

The build procedures for the Toolkit are conditional builds. This means that the application will be rebuilt only if the source files (**.C**) are newer than the object files (**.OBJ**). Therefore, if you are switching from a build that requires a coprocessor to one that doesn't, the object files (**.OBJ**) need to be deleted.

Debugging Under MS-DOS

If MS-DOS testing of a C application uncovers bugs in the C application source file, use one of the following approaches to debug the file:

1. **`printf( )`** debugging
2. Microsoft **`CodeView`** debugging

printf( ) Debugging

Using **`printf( )`** debugging involves inserting **`printf( )`** statements at key points within the C application. As the application executes under MS-DOS, each of the **`printf( )`** statements encountered will cause its text string to be printed to the screen. By following the printed text string, you can determine the path taken through the software. The number or placement of any **`printf( )`** statement is completely up to the developer.

However, there are a few drawbacks to using **`printf( )`**:

1. If the C application is not a simple program, the number of **`printf( )`** statements required to follow the execution path may be quite large.
2. If there are a large number of **`printf( )`** statements inserted into the C application, the compile time and amount of generated code will be greatly increased.
3. With a large number of **`printf( )`** statements, the corresponding number of text strings displayed to the screen may make following the program's execution path difficult.
4. Debugging using **printf( )** is available to C FBKs when operating in the MS-DOS environment.

Because of these drawbacks to using **`printf( )`** debugging, Microsoft **`CodeView`** debugging is preferred for all but the simplest C applications.

Microsoft CodeView Debugging

The **`.EXE`** file created when either **`MK3DOS.BAT`** or **`MK3DOS7.BAT`** is used contains the symbol, line number, and debug information necessary for using the Microsoft **`CodeView`** debugger. Within **`CodeView`**, variables may be examined, variables may be modified, breakpoints can be set, input can be entered, and output can be observed − all without adding numerous **`printf( )`** statements.

While in **CodeView**, the MS-DOS versions of PLC reference memory tables (sized and allocated through the harness) can be accessed using the names described in the following table:

Table 3-11. MS-DOS Versions of 90-30 PLC Reference Memories

Memory Type	PLC Reference	CodeView Symbol (case sensitive)
Registers	%R	___R_TBL
Analog Inputs	%AI	___AI_TBL
Analog Outputs	%AQ	___AQ_TBL
Inputs	%I	___I_TBL
Outputs	%Q	___Q_TBL
Temporary Coils	%T	___T_TBL
Internal Coils	%M	___M_TBL
System Status	%S %SA %SB %SC	___S_TBL ___SA_TBL ___SB_TBL ___SC_TBL
Genius	%G	___G_TBL
Transition Memory		___I_TRANS_TBL ___Q_TRANS_TBL ___T_TRANS_TBL ___M_TRANS_TBL ___S_TRANS_TBL ___SA_TRANS_TBL ___SB_TRANS_TBL ___SC_TRANS_TBL ___G_TRANS_TBL

Using the symbol names in the previous table, any of the MS-DOS versions of the 90-30 PLC reference table may be initialized, viewed, or modified during execution.

For more information on Microsoft **CodeView**, please consult the documentation provided with your Microsoft C Compiler.

Section 8: C Applications in the Series 90 PLC Environment

Once the C application has been tested under MS-DOS and all execution errors have been corrected, the next step is to rebuild the application for execution in the Series 90 PLC.

BLDVARS File

The Toolkit uses a file, BLDVARS, to define the type of application being built and determine whether debug information is included. If this file does not exist in the directory with the source files, the build procedure will copy the **BLDVARS** file from the **\S9070C** or **\S9030C** directory. The default **BLDVARS** file is shown below:

```
#########################################################################
#
#  BLDVARS
#  Copyright (c) 1994, GE Fanuc Automation North America, Inc.
#                  All rights reserved.
#
#  This file defines build parameters.
#
#########################################################################

# debug information on or off
# ON include debug information in the build
#  for the debugger
# OFF do not include the debug information in
#  the build
DEBUG=OFF
# type of build (C exe block, C fbk, standalone C program)
# BLOCK      C exe block
# FBK        C fbk
# STANDALONE  standalone C program
TYPE=BLOCK
# C application Import type
# LM90    compatible for LM90 import
# CC90    compatible for import into the Windows-based programming software
# Import=LM90
```

The default **BLDVARS** file designates a C block build with no debug information. To allow for debug information (which is required for running the C Application Debugger under the PLC), change the line in **BLDVARS** that is currently **DEBUG=OFF** to **DEBUG=ON** (note that ON must be in all capital letters). To change the application type that is built, change the type field from **TYPE=BLOCK** to either **TYPE=FBK** or **TYPE=STANDALONE** (once again in all capital letters). If you wish to change these variables strictly for the files in one directory, make the changes to the **BLDVARS** file in that directory.

Note

The Series 90-30 PLC does ***not*** support DEBUG = ON. Also, TYPE must be set to BLOCK.

Creating a Folder for a Standalone C Program (Series 90-70 Only)

When creating a standalone C program, a Logicmaster 90-70 folder MUST be created BEFORE the build procedure is started. The build procedure will not create a folder. Refer to *Logicmaster 90-70 User's Manual* (GFK-0263) for information on how to create a program folder.

Building for 90-70 PLC Execution

To build a PLC-executable **.EXE** file from a C application source file, use one of the two DOS command files **MKPLC.BAT** or **MKPLC7.BAT**. (Both of these are provided in the **\s9070c** directory as part of the C Programmer's Toolkit.) The use of and differences between **MKPLC** and **MKPLC7** are analogous to **MKDOS** and **MKDOS7** for building DOS executables. **MKPLC** will build a PLC executable which does not require a coprocessor and will not use a coprocessor, even if one is present. **MKPLC7** does the opposite: it will build a PLC executable which requires a coprocessor and will not execute without a coprocessor.

If no floating point is required or if floating point is required but no coprocessor will be available in the target PLC, use **MKPLC.BAT**:

```
c:\apps> MKPLC  <application filename>
```

If, however, floating point is required and the target PLC is guaranteed to have a math coprocessor, use **MKPLC7.BAT**:

```
c:\apps> MKPLC7  <application filename>
```

Note

When using MKPLC and the Microsoft C v7.0 or v8.0 compiler to create a C application, the following messages will be displayed. This messages are expected and no action is required:

```
Command line warning D4023 : option '/FPa' forces use of optimizing compiler
```

```
LINK warning L4021 : no stack segment
```

Invoking either **MKPLC** or **MKPLC7** and specifying the application source code file name will cause the following:

1. The creation of a subdirectory named PLC under the current directory
2. The compilation of the application source file creating an output object file (**.OBJ**) in the PLC subdirectory
3. The linking of the object module with the required runtime libraries to create an output file (**.PPP**) in the PLC subdirectory

4. The processing of the linked file (**.PPP**) in the PLC subdirectory into a PLC-executable file (**.EXE** for C blocks and **.STA** for standalone C programs) in the PLC subdirectory
5. In the case of a standalone C program build, the user will be prompted for a target folder name, and the PLC executable file (**.STA**) will be imported into the folder and deleted from the PLC subdirectory. For more detail see the section below *Importing Standalone C Programs into Folders*.

The PLC-executable file as well as the standalone C program will have the same name as the application C program source file. (If the C application source file is **QWERTY.C**, the PLC executable file to be added to the Logicmaster 90-70 Librarian would be **QWERTY.EXE** and the standalone C program name would be **QWERTY**).

Note

PLC-executable files may be invoked under MS-DOS without causing the MS-DOS machine to crash. All PLC-executable files created with the C Programmer's Toolkit verify that the target hardware is a Series 90-70 PLC at execution startup. If any PLC executable detects that it is not running in a Series 90-70 PLC, an error message is displayed and the program immediately returns to MS-DOS.

The MS-DOS batch files **MKPLC.BAT** and **MKPLC7.BAT** and the makefiles invoked by each of these batch files are designed to support a single application source file or multiple sources files. Refer to Chapter 6, "C Application Development Using Multiple C Source Files," for more information about multiple source files.

Note

The build procedures for the Toolkit are conditional builds. This means that the application will be rebuilt only if the source files (**.C**) are newer than the object files (**.OBJ**). Therefore, if you are switching from a build that requires a coprocessor to one that doesn't, the object files (**.OBJ**) need to be deleted.

Building for 90-30 PLC Execution

To build a PLC-executable **.EXE** file from a C application source file, use one of the two DOS command files **MK3PLC.BAT** or **MK3PLC7.BAT**. (Both of these are provided in the **\s9030c** directory as part of the C Programmer's Toolkit.) The use of and differences between **MK3PLC** and **MK3PLC7** are analogous to **MK3DOS** and **MK3DOS7** for building DOS executables. **MK3PLC** will build a PLC executable which does not require a coprocessor and will not use a coprocessor, even if one is present. **MK3PLC7** does the opposite: it will build a PLC executable which requires a coprocessor and will not execute without a coprocessor.

If no floating point is required or if floating point is required, use **MK3PLC.BAT**:

```
c:\apps> MK3PLC  <application filename>
```

If, however, floating point is required and the target PLC is guaranteed to have a math coprocessor, use **MK3PLC7.BAT**:

```
c:\apps> MK3PLC7  <application filename>
```

Note

When using MKPLC and the Microsoft C v8.0 compiler to create a C application, the following messages will be displayed. This messages are expected and no action is required:

```
Command line warning D4023 : option '/FPa' forces use of optimizing compiler
```

```
LINK warning L4021 : no stack segment
```

Invoking either **MK3PLC** or **MK3PLC7** and specifying the application source code file name will cause the following:

1. The creation of a subdirectory named PLC under the current directory
2. The compilation of the application source file creating an output object file (**.OBJ**) in the PLC subdirectory
3. The linking of the object module with the required runtime libraries to create an output file (**.PPP**) in the PLC subdirectory
4. The processing of the linked file (**.PPP**) in the PLC subdirectory into a PLC-executable file (**.EXE** for C blocks and **.STA** for standalone C programs) in the PLC subdirectory

The PLC-executable file will have the same name as the application C program source file.

Note

PLC-executable files may be invoked under MS-DOS without causing the MS-DOS machine to crash. All PLC-executable files created with the C Programmer's Toolkit verify that the target hardware is a Series 90 PLC at execution startup. If any PLC executable detects that it is not running in a Series 90 PLC, an error message is displayed and the program immediately returns to MS-DOS.

The MS-DOS batch files **MK3PLC.BAT** and **MK3PLC7.BAT** and the makefiles invoked by each of these batch files are designed to support a single application source file or multiple sources files. Refer to Chapter 6, "C Application Development Using Multiple C Source Files," for more information about multiple source files.

Note

The build procedures for the Toolkit are conditional builds. This means that the application will be rebuilt only if the source files (**.C**) are newer than the object files (**.OBJ**). Therefore, if you are switching from a build that requires a coprocessor to one that doesn't, the object files (**.OBJ**) need to be deleted.

Importing Standalone C Programs into Folders (Series 90-70 Only)

Importing C Programs with Logicmaster

During the MKPLC process, a list of subdirectories of the current working directory will appear on the screen as follows:

```
Subdirectories:
PLC
Please Enter Folder Name:
```

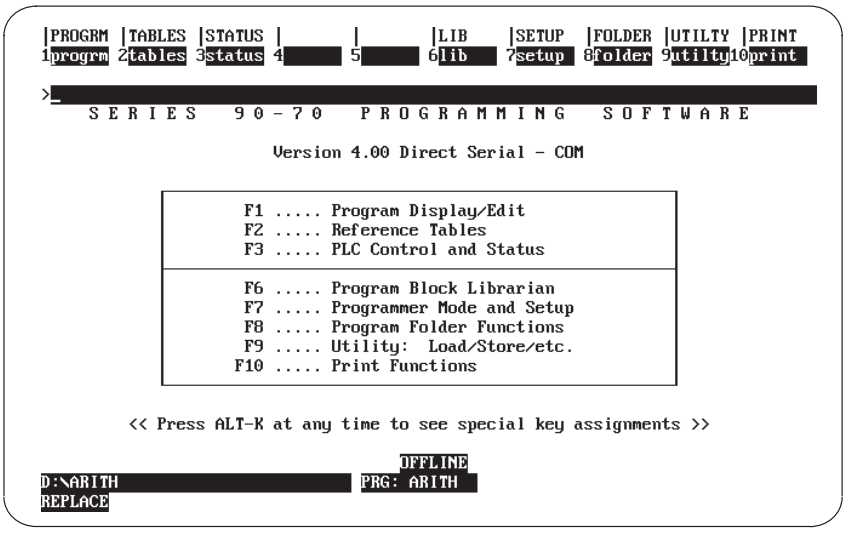
If one of these subdirectories listed is the desired folder, then you can enter the name of the subdirectory, and the build procedure will import the standalone C program. If you know the path to the folder, you can also enter the path with the folder name, and the build procedure will import the standalone C program. If you don't know the location of the folder, change directories by typing in the path and ending with a \, (this will change the current working directory to that path and show the subdirectories). For example:

```
Subdirectories:
PLC
Please Enter Folder Name: \FOLDERS\

Subdirectories:
ARITH
Please Enter Folder Name:
```

Adding Blocks Through the Logicmaster 90 Librarian (Series 90-70 Only)

Before importing the block into Logicmaster 90, the C application source file must be compiled and linked to successfully create the PLC-executable version of the C application.

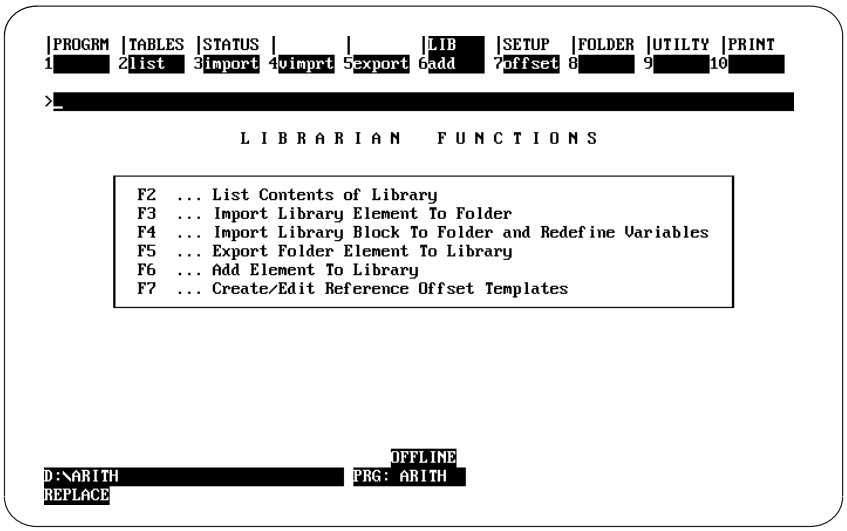
Once the PLC-executable version of a C application source file is created, the executable needs to be loaded into Logicmaster 90. Use the Logicmaster 90 Librarian to add a block into the Logicmaster 90 library. From the main menu of the Logicmaster 90 programming software, press the Librarian (F6) softkey.



Note

In the Windows-based programming software, use the New Block method of adding blocks (C or other types of blocks) to the Equipment Folder. For directions on performing this task, refer to page 3-98 and following. This information also exists in Chapter 6 in GFK-1295 and in the online help that comes with the Windows-based programming software.

After pressing F6, the following Librarian Functions menu is displayed:

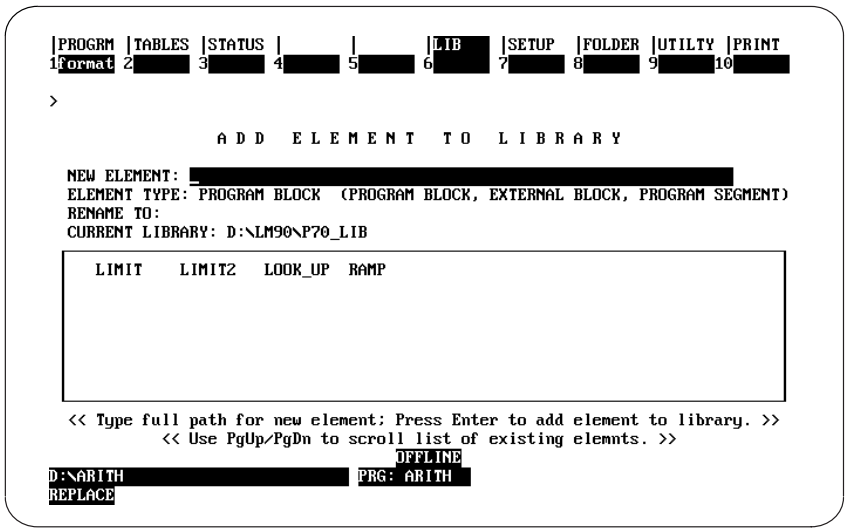


The two functions you will use to add a block to a folder are:

1. Add Element to Library (F6).
2. Import Library Element to Folder (F3).

An external block, such as a block, cannot be directly added to a Logicmaster 90 folder. The block must first be added to the Logicmaster 90 library. Once in the library, it can then be added to a folder.

To add an external block to the Logicmaster 90 library, press the F6 key (Add Element to Library) from the Librarian Functions menu. The Add Element screen is displayed:

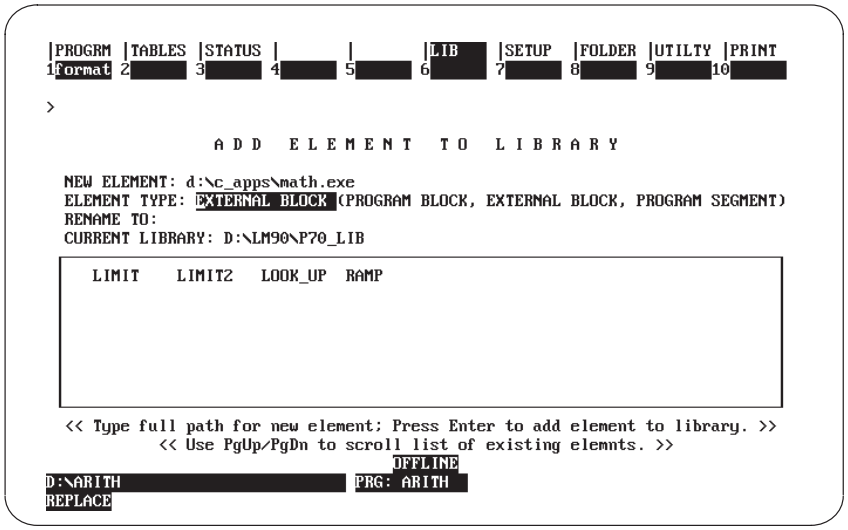


The add element function requires the complete path name and file name of the element to be added, and the type of element (program block, program segment, or external block). An optional ***Rename To*** field may be used to rename a block as it is inserted into the Logicmaster 90 library.

Note

New element file names are limited to seven (7) characters. Do not use file names greater than seven characters in length.

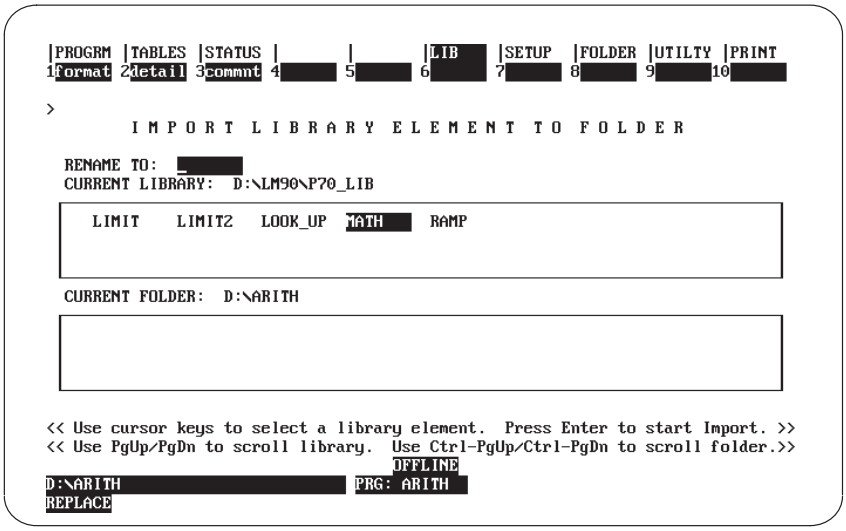
To add a block to the library, enter the complete path name and file name in the ***New Element*** field, then press the Down arrow key to move to the ***Element Type*** field. Use the Tab key to select EXTERNAL BLOCK as the element type. When both the element type and new element fields are correctly filled in, press the Enter key.



After pressing the Enter key, the Logicmaster 90 software will prompt you for the number of input/output parameter pairs. This is the number of input/output parameter pairs specified in the block declaration of **`main( )`**. At the prompt, enter a number from 0 to 7 corresponding to the number of parameter pairs the block expects to have passed to **`main( )`**. The Logicmaster 90 software will ask you to confirm the name of the element to be added and the number of parameter pairs after the prompt "Add element *NEWNAME* (*X* pairs) to library? (Y/N)." Press **Y** (Yes) to proceed. When the add element function is completed successfully, the message "*NEWNAME* added to library" is displayed.

With the block added to the Logicmaster library, the next step is to import the library element (the newly added block) from the library into a program folder. To perform the import, first return to the Librarian Functions menu by pressing the Escape key. Then press the F3 softkey (Import Library Element To Folder).

The following screen is displayed:



The import element function will import the library element into the currently selected program folder. The library element may be selected by using the cursor key to highlight the desired element. Once the library element has been selected, press the Enter key. At import time, the copy of the library element that will be copied into the current program folder may be renamed by entering the new name into the ***Rename To*** field before pressing the Enter key.

After pressing the Enter key, the Librarian will prompt you for confirmation to import the selected (highlighted) library element to the current program folder. Press **Y** (Yes) to confirm the prompt. The library element will be imported into the current program folder.

After the library element is successfully imported to the current program folder, the ladder program may be edited to insert ladder calls to the block by using the CALL EXTERNAL function block. (A declaration for the imported block is automatically added to the ladder logic program.) Recall that during the add element operation, the Librarian prompted you for the number of input/output parameter pairs for the external block being added. Since the Logicmaster 90 programmer knows how many parameter pairs are required for each block in its library of program blocks, the Logicmaster 90 software will provide the correct CALL function block format (number of parameters) when the CALL and target block name are provided.

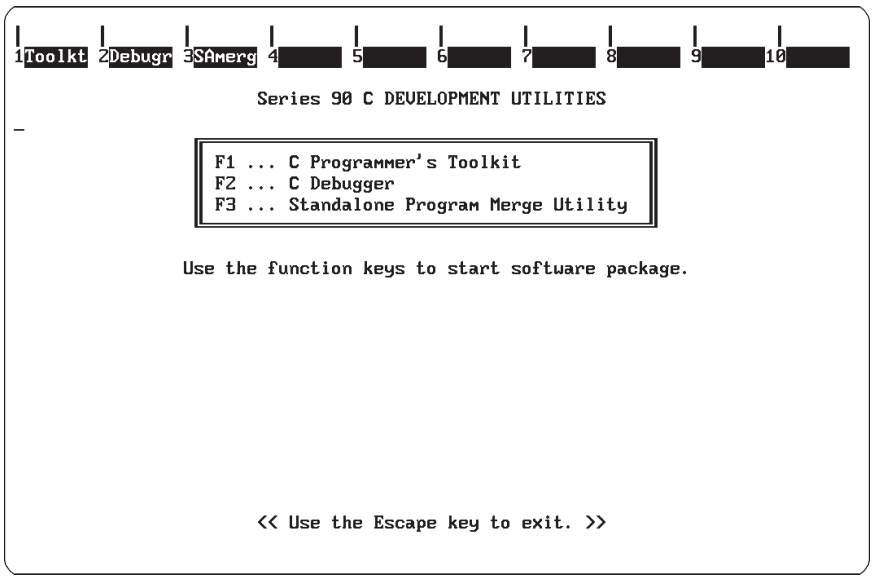
Note

Each time a change to the block occurs, the file must be re-compiled using **MKPLC** (or **MKPLC7**), re-added to the Logicmaster 90 library, and then re-imported into the Logicmaster folder. Failure to perform these steps will result in the software using a previous version of the block.

Standalone C Program Merge Option

An alternate way of bringing a C program into your folder is to use the SA Merge option (especially useful if you have a C program but do not have the Toolkit on your PC or if you are putting the standalone program into more than one folder). To use this option, first press (F6) from the main menu in Logicmaster.

The following screen will be displayed:



Press (F3) to begin the Standalone Program Merge process. The following screen will be displayed:

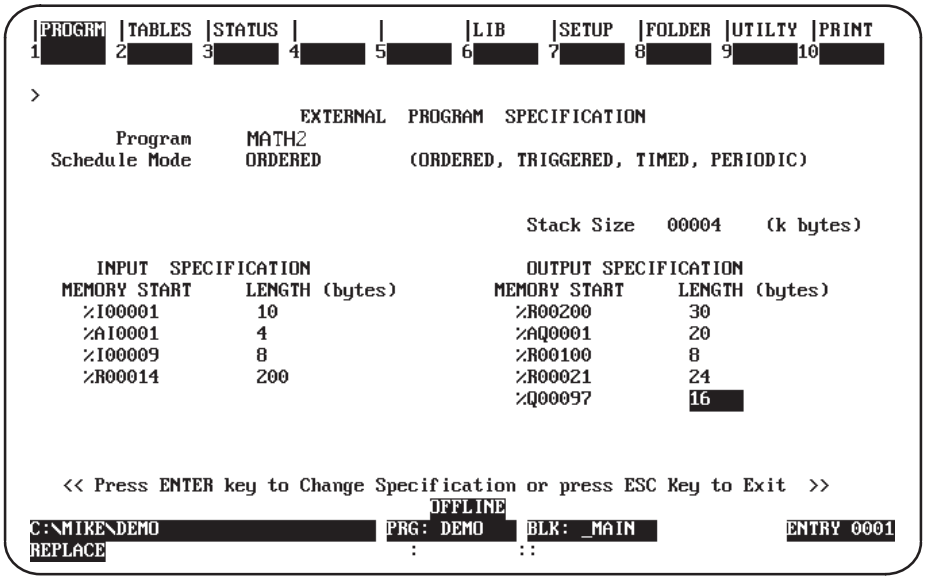
```
STANDALONE MERGE UTILITY version 1.4

PLEASE ENTER THE NAME OF THE FILE YOU WISH TO COPY
NOTE: ending the string with \ will move to that directory

   Current Subdirectories:
[.]           [..]          [C70S]        [P70S]
[LESSON]      [EM7]         [HSC]         [FIP]
[C70]         [P70]         [FIELD]       [FLDCTRL]
[P70_LIB]     [LESSON1]     [SAINTS]      [MULTI]
[C30S]        [P30S]        [TCPIP]       [MICRO]
[LESSON3]     [TCPIP2]      [LESSON2]     [UPGRADE]
[FIELD1]      [NEW]         [LESSON4]     [NEW1]
[LESSON5]     [FIPNEW]      [CTOOL]       [VME]
[BLKMVR]      [R7CNVT]      [X_PLC]       [E_PLC]
[REAL]        [LESSON6]     [LESSON7]     [REALLY]
[DEC2_96]     [LAB2]        [PROB_1]      [LESSON9]
[COFFEE]      [SNPTEST]     [LESSON8]

   Standalone .STA files in the current directory:
BUBBLE.STA    LIMIT.STA

File to copy: C:\LM90\
```

Type the name of the standalone C program you wish to merge into the folder. If the standalone C program is not located in the default directory, type the path and name of the program.

Scheduling Standalone C Programs through the Logicmaster 90-70 Scheduler

Perform the following steps to schedule a standalone C program:

1. Go to the Program Specification screen under Logicmaster 90-70 (F1).
2. Press (F1) and enter the name of the standalone C program.
3. If the ladder logic is still desired, press (F1) and enter the name of the folder.
4. Zoom into the programs (F9).



5. Select the schedule mode for the program(s) (ordered, triggered, timed, or periodic) and any other options desired or required (priority, disable bit, etc.). Refer to Logicmaster 90-70 User's Manual, GFK-0263, for more information.
6. Fill in the I/O specifications (make sure they match those in the program; these will not be checked by Logicmaster).

Note

For directions on performing these tasks in the Windows-based programming software, refer to the "Controlling Block and Program Execution" section in Chapter 6 in GFK-1295. The same topic also exists in the online help that comes with the Windows-based programming software.

Working with C Programs and Blocks in the Windows-based Programming Software

For Series 90-70 PLCs, the tasks performed in the Windows-based programming software are very similar to the tasks performed in Logicmaster, but the interface is different. For 90-30 PLCs, the Windows-based programming software is the only programming interface that accommodates the 90-30 C (Main) programs and 90-30 C subroutine blocks.

Adding C Programs to Your Equipment Folder (Series 90-70 Only)

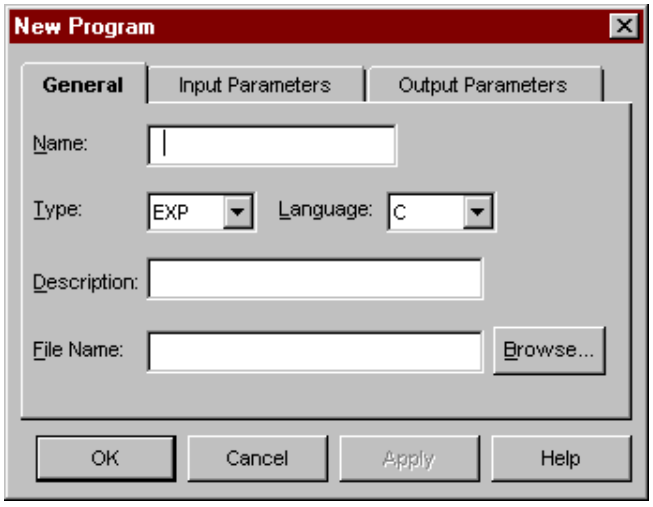
Once your standalone C program is developed, follow these steps to incorporate the program into the the Windows-based programming software Equipment Folder:

1. From the Equipment Folder Browser, click Resource. (If the Resource is not displaying, click the plus (+) sign next to SW Config in the left side of the Equipment Folder window—if SW Config is not displaying, click the plus sign next to the folder name.)

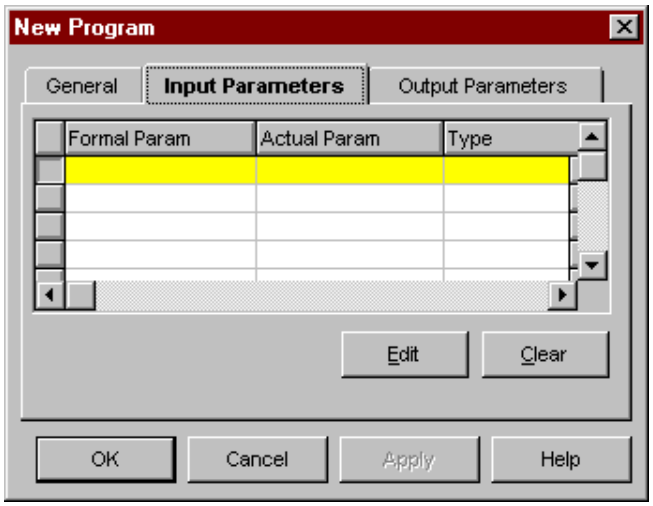
2. Click the right mouse button and choose Edit Resource. The Resource Window will appear.

3. Select the Program menu and choose New Program.

4. In the New Program dialog box, set the type field to EXP for external C program. Type the name you want to call this program within your Equipment Folder in the Name field. This name does not need to match the name you already have given your C program.



5. Type the name and location of the C program or use the Browse button to locate the C program. To specify any input or output parameters, click the appropriate tab (click the Edit button to create or edit parameters):



6. Click the OK button. If you are importing more than one program, click the Apply button instead of OK. This allows you to add another program without closing the dialog box.

Importing Revised C Programs

If you need to make changes to an external C program that has already been added to an Equipment Folder, you must update the program using the C Programmer's Toolkit before importing the revised program into your Equipment Folder.

You can import the revised C external programs from the Equipment Folder Browser or the Resource window. After editing and compiling your C program, perform the following steps to import the program into your Equipment Folder:

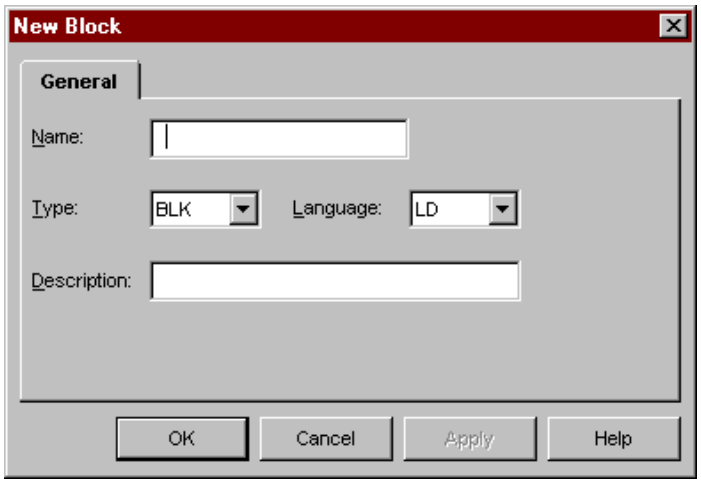
1. Edit and compile your C program using the C Programmer's Toolkit.
2. From the Equipment Folder Browser or the Resource, click the C program you want to update.
3. Click the right mouse button and select Import.
4. In the dialog box, type the path and name of the revised C program and click the OK button or click the Browse button to locate and specify the path and file name.

Creating or Adding Blocks

Follow these steps to add new blocks at the Program Level:

1. In the Equipment Folder Browser window, select Program (by clicking once on it).

2. Click the right mouse button and choose New Block. The New Block dialog box will appear:



3. In the New dialog box, type the name of the block in the Name field and select the block type from the drop-down list in the Type field and the language (that is, C) from the drop-down list in the Language field.

4. Specify the file name and path in the File Name field, or locate the file using the Browse button.

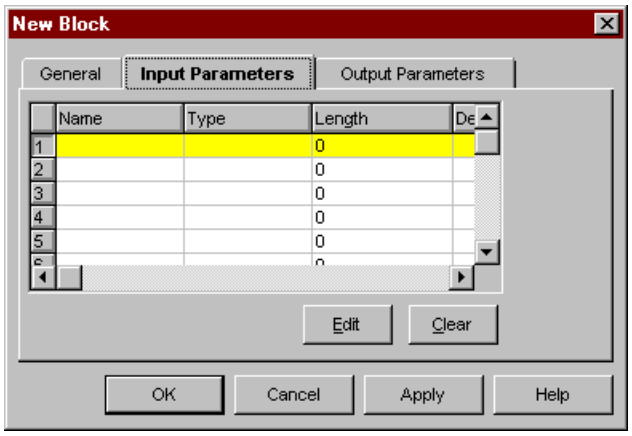
5. When adding a 90-70 C block, you probably need to fill in the information in the Input Parameters and Output Parameters tabs (see below). This is not done with 90-30 C blocks because they do not have parameters.

6. Click OK. After doing the preceding steps for each block you plan to use, you can call the block from within _MAIN.

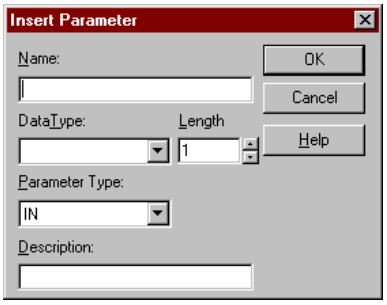
Specifying Parameters for 90-70 C Blocks

When you are adding a C block to your 90-70 Equipment Folder, you may also need to specify input and output parameters. Use the following procedures to define these parameters:

1. Once you have completed the necessary information in the General tab of the New Block dialog box, click the Input Parameter or Output Parameter tab.



2. Each Parameter tab contains a table. To add entries to the table, click the Edit button. The Insert Parameter dialog box will appear:



3. In the Insert Parameter dialog box, type the formal parameter name (which can then be used in the block to reference the parameter), click the drop−down arrow to choose the data type, click the up or down arrows to specify length, and type in a description (optional).

4. After you have defined the necessary input and output parameters, click the OK button.

Editing C Blocks

To edit an existing external C block, follow these steps:

1. Edit and compile the C block using the C Programmer's Toolkit.
2. If the block is not displayed in the Equipment Folder Browser below the program, select Program, click the right mouse button, and choose Refresh Node.
3. Click the C block, click the right mouse button, and select Import.
4. In the dialog box, type the path and name of the revised C block and press the OK button or click the Browse button to locate and specify the path and file name.

Adding C Main Programs (Series 90-30 Only)

Series 90-30 CPUs do not allow standalone C programs with scheduling, event-triggered, and other specifications; however, you can have a C program as the main program by following these steps:

1. Create a new 90-30 Equipment Folder.
2. In the left side of the Equipment Folder window (the Browser), expand the selections (by clicking on the plus signs) until the Resource (under SW Config) is displaying.
3. Double-click Resource to open the Resource editor.
4. Select the default main program (by clicking once on it).
5. Press the Delete key on your keyboard to delete the default main program.
6. Select the Program menu and choose New Program.
7. In the dialog box that appears, change the Type field to EXP and the language to C. In the Name field, type in the name of the Resource (usually the same as the name

of the Equipment Folder). In the File Name field, type the path and name of the C program or use the Browse button to navigate to the C program. Then press OK.

This information is also available in the online help and in GFK-1295. For related information, refer to the "Adding C Programs to Your Equipment Folder" and "Importing Revised C Programs" sections of Chapter 6 in GFK-1295 or read those same topics in the online help.

Debugging in the PLC

There are several ways to debug the C application operating in the PLC, including **`printf( )`** debugging, reference table monitoring, single-sweep mode debugging,. The C Debugger is also available for debugging applications. Refer to Chapter 7, "The C Application Debugger for Series 90-70 PLCs," for more information.

Note

The only way to view internal C application data without the C Debugger is to copy the data from its C variable location to an unused PLC reference table location.

Printf( ) Debugging (Series 90-70 Only)

The use of **`printf( )`** to debug a C application running in a Series 90-70 PLC is very similar to using **`printf( )`** to debug the same C application under MS-DOS. The C source code must be modified to contain **`printf( )`** statements. The **`printf( )`** statements should be placed in the source code to provide a road map of the execution path and to display the value of any key data items.

Note

In order for **`printf( )`** to work, the CPU's serial port must be configured for **`message generation (MSG)`** mode. If the CPUs serial port is not configured for **`MSG`** mode and **`printf( )`** is called, no characters are placed into the print queue, the return value from **`printf( )`** is -1, and an External Block Run Time error will be logged in the PLC fault table.

Reference Table Monitoring

As with **`printf( )`** debugging, the execution path and key data items may be determined by modifying a C application to place this information into unused areas of the global PLC reference tables (%R, %M, %T, %P,... etc.) and then viewing the saved execution road map and key data items through the Logicmaster 90 programmer's online reference display(s).

Single-Sweep Debug (Series 90-70 Only)

Release 4.0 Series 90-70 CPUs, when used in conjunction with Release 4.0 Logicmaster 90-70 software, support a single-sweep program debug command. When this command is issued, the PLC will execute one sweep. Use of single sweep debug may prove helpful in debugging C applications in a Series 90-70 PLC. For details on using the single-sweep debug with the debugger refer to Chapter 7, "The C Application Debugger for Series 90-70 PLCs."

In order for single sweep debug in the PLC to aid in the debugging of C applications, it is necessary that the C application provide some information back to the user. This information may be generated by **`printf( )`** statements to the serial port or by copying key data values into an unused portion of the PLC reference table(s).

To use single sweep debug, the Logicmaster 90 programmer must be online with the PLC and the PLC must be in **STOP** mode. By pressing ALT-G, the CPU will be commanded to execute a single sweep. You may specify a first scan sweep by entering **0** on the command line and then pressing ALT-G. Subsequent sweeps may be singly executed by pressing ALT-G with no value on the command line. If another first scan sweep is desired at any time, enter **0** on the command line again and press ALT-G.

The most convenient way to accomplish a single sweep debug is as follows:

1. Ensure that the C application is copying any key data into unused PLC reference table areas (for example, %R memory) so that the Logicmaster 90 software displays this data.
2. If all of the data to be viewed is in one PLC reference memory, select that reference table for display using the Logicmaster 90 reference table display function. If multiple reference tables are being used to store the C application data which is to be viewed, a mixed reference table may be created and viewed using the reference table display function.
3. While displaying the selected reference table(s), initialize any necessary reference locations, such as those locations in which the C application will expect data to be but which the ladder logic program will not initialize prior to calling the C application.
4. While still viewing the selected reference table(s), enter **0** on the command line and press ALT-G to command the PLC to execute a single, first scan sweep.
5. Check the data values in the displayed reference table(s) against desired C application operation.
6. If there are no errors at this point, the debug process may be complete. If the problem was related to first scan, no further action is required. If, however, the problem is not a first scan problem, press ALT-G again for another sweep of PLC ladder execution. Go to step 5.
7. If there are errors, inspect the C source code and the received data to isolate the problem. If the problem can be isolated and corrected, rebuild and re-import the C application. Go back to step 2.
8. If the problem cannot be isolated, it may be necessary to have the C application copy more pieces of information into the reference table(s) it is using. Modify the C source code to copy more information, then rebuild the C block, add it to the Logicmaster 90 library, re-import the C application into the folder, and store the updated folder to the CPU. Go back to step 2.

Chapter 4 Example C Series 90-70 Application Development

Section 1: Installed Sample Blocks

In the C development software directory, **\s9070c**, there are two subdirectories which have examples of blocks: **example1** and **example2**. Each subdirectory contains the block source file **LIMIT.C**. **LIMIT.C** provides an illustration of a function which could be written in ladder logic, but is, instead, written in C.

LIMIT is an application program which range checks a current value against a low limit and a high limit and returns, if necessary, a corrected value which is within the range. If the current value is lower than the low limit or higher than the high limit, then **LIMIT** will return low limit or high limit, respectively. **LIMIT** takes three input parameters and returns two output parameters. The three input parameters correspond to a current value, a low limit value, and a high limit value. The two output parameters are the current value, corrected if necessary, and two alarm indication bits (low and high alarms which are bits 2 and 1, respectively, in output reference number two). A ladder logic call to this external block could look like the following:

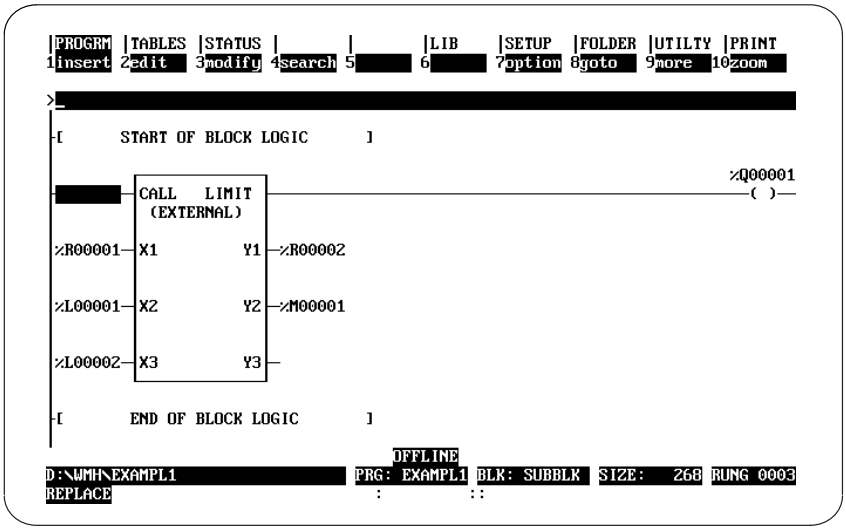


Figure 4-1. Ladder Logic Call to Example Block

In the sample ladder logic rung above, %R00001 contains the current value, %L00001 contains the high limit value, and %L00002 contains the low limit value. LIMIT.C is shown below:

```
#include "plcc9070.h"

#define   HI_ALM_MSK    0x01
#define   LO_ALM_MSK    0x02

main( short int  *input_value,     /* parameter X1 */
      short int  *high_limit,      /* parameter X2 */
      short int  *low_limit,       /* parameter X3 */
      short int  *output_value,    /* parameter Y1  */
      short int  *alarm_bits,      /* parameter Y2  */
      void       *dummy )          /* parameter Y3  */

{
/*  check for error condition high_limit < low_limit
 *  if error, set no power flow and return, don't change any outputs
 */
    if (*high_limit < *low_limit)
       return(ERROR);
    /*
     *  check for in out range
     *  if in exceeds either limit
     * use the exceeded limit value for the output
     *  else use the input value for the output
     *  always write both high and low alarm bits appropriately
     */
    if ((*input_value) > (*high_limit)){
       *output_value = *high_limit;   /* use high_limit for output */
       *alarm_bits |= HI_ALM_MSK;     /* turn high alarm bit on */
       *alarm_bits &= ~LO_ALM_MSK;    /* turn low alarm bit off */
    } else if ((*input_value) < (*low_limit)) {
       *output_value = *low_limit;    /* use low_limit for output */
       *alarm_bits &= ~HI_ALM_MSK;    /* turn high alarm bit off */
       *alarm_bits |= LO_ALM_MSK;     /* turn low alarm bit on */
    } else {
       *output_value = *input_value;  /* use in for output */
       *alarm_bits &= ~HI_ALM_MSK;    /* turn high alarm bit off */
       *alarm_bits &= ~LO_ALM_MSK;    /* turn low alarm bit off */
    }

    return(OK);
}
```

The return parameters from the sample ladder logic call to the external block LIMIT are the range corrected current value in %R00002, an indication of the input current value being over the high limit in %M00001, and an indication of the input current value being under the low limit in %M00002 (bit 2 of the 16 bit value starting with %M0001).

The following two sections will describe how to interactively test a block (example 1) and how to batch test a block (example 2). The differences between the two methods (interactive and batch) will be shown in the test harness file, **BLKHARN.C**.

Example 1: Interactive LIMIT

As stated at the beginning of this chapter, **LIMIT.C** is the same in both this example and in **example2**. The difference between the two examples is the manner in which testing under MS-DOS may be accomplished. In this example, you will use interactive methods to request user input and display data values.

The interactive commands which get user input to pass to the block and which display data upon return of the block are all placed into the file **BLKHARN.C**; if a C FBK is to be built then the file needs to be renamed to **FBKHARN.C**. Recall that **BLKHARN.C** is the MS-DOS test harness which is linked with the application (in this case, **LIMIT**) to create an MS-DOS executable and which provides the basic PLC sweep mechanism. The intent of **BLKHARN.C** is not only to provide the basic sweep mechanism, but also to contain any and all debugging code.

The code contained in **BLKHARN.C** for **example1** is shown below:

```
#include <stdio.h>
#include "plcc9070.h"

/*****************************************************************************
 *      GE Fanuc 90-70 PLC Memory Size Declarations                          *
 *****************************************************************************/
#define       L_MEM_WORDS       1024
#define       P_MEM_WORDS       1024
#define       R_MEM_WORDS       1024
#define       AI_MEM_WORDS        64
#define       AQ_MEM_WORDS        64
#define       I_MEM_BYTES        256
#define       Q_MEM_BYTES        256
#define       T_MEM_BYTES         32
#define       M_MEM_BYTES        512
#define       SA_MEM_BYTES        16
#define       SB_MEM_BYTES        16
#define       SC_MEM_BYTES        16
#define       S_MEM_BYTES         16
#define       G_MEM_BYTES        160
#define       GA_MEM_BYTES       160
#define       GB_MEM_BYTES       160
#define       GC_MEM_BYTES       160
#define       GD_MEM_BYTES       160
#define       GE_MEM_BYTES       160
#define       I_DIAG_BYTES       256
#define       Q_DIAG_BYTES       256
#define       I_TRANS_BYTES      256
#define       Q_TRANS_BYTES      256
#define       T_TRANS_BYTES       32
#define       M_TRANS_BYTES      512
#define       SA_TRANS_BYTES      16
#define       SB_TRANS_BYTES      16
#define       SC_TRANS_BYTES      16
#define       S_TRANS_BYTES       16
#define       G_TRANS_BYTES      160
#define       GA_TRANS_BYTES     160
#define       GB_TRANS_BYTES     160
#define       GC_TRANS_BYTES     160
#define       GD_TRANS_BYTES     160
#define       GE_TRANS_BYTES     160
#define       AI_DIAG_BYTES       64
#define       AQ_DIAG_BYTES       64
#define       RSB_MEM_BYTES      688
```

```
#include  “tharndat.inc”

static short int   X1 = 2;            /* input 1 parameter */
static short int   X2 = 3;            /* input 2 parameter */
static short int   X3 = 1;            /* input 3 parameter */
static short int   Y1;                /* output 1 parameter */
static short int   Y2;                /* output 2 parameter */
static short int   Y3;                /* output 3 parameter */

void plc_sweep(int argc,char *argv, char *envp)
{
    int  ok;  /* used to store returned power flow -- the OK parameter */

/*
 * Modify stdout to use NON-buffered i/o
 */

    setbuf( stdout, NULL );


/*
 * To mimic PLC operation, initialize the FIRST EXE and FIRST SCAN
 * special status bits.
 */

(*sb_mem)[0]       |= 0x01;           /* FST_SCN */
(*sb_mem)[15]      |= 0x01;           /* FST_EXE */
(*sb_mem)[0]        = 0x40            /* ALW_ON  */
(*sb_mem)[0]       &= ~x80            /* ALW_OFF */

/*
 *  Loop forever
 *  Prompt user for IN, HIGH_LIMIT, LOW_LIMIT.
 *  Call main to execute block.
 *  Print out results.
 *  End loop
 */
```

```
   while (1)  {

      printf("INPUT VALUE: ");
      scanf("%d",&X1);
      printf("HIGH_LIMIT: ");
      scanf("%d",&X2);
      printf("LOW_LIMIT: ");
      scanf("%d",&X3);

      /* X1 - X3, Y1, & Y2 are all used.  Y3 is unused by  */
      /* main() in LIMIT.C but is required to appear in    */
      /* the call to main() to preserve parameter location */
      /* on the stack.                                     */
      ok = main(&X1, &X2, &X3, &Y1, &Y2, &Y3);

      /* From description of LIMIT in top of file LIMIT.C  */
      /* the alarm bits returned from LIMIT are the 2      */
      /* least significant bits in the Y2 reference.       */
      printf("OK: %d, OUT: %d, HIGH ALARM: %d, LOW_ALARM: %d.\n\n",
         (int)(ok==OK), Y1, (Y2 & 0x01), ((Y2 & 0x02) >> 1) );

      /*
       * To further mimic PLC operation, now
       * clear the FIRST EXE and FIRST SCAN
       * special status bits.
       */
      (*sb_mem)[0]  &= 0xFE;
      (*sb_mem)[15] &= 0xFE;
   }
}
```

This version of **BLKHARN.C** allows for the interactive testing of the **LIMIT** application program. The interactive nature of this version of **BLKHARN** is seen in the prompting for input values at the beginning of the **while( )** loop (**printf( )** and **scanf( )** pairs) and in the single **printf( )** statement which appears inside the **while( )** loop immediately after the call to **main( )**. The **printf( )** and **scanf( )** statements in this version of **BLKHARN.C** are written specifically for testing the example program **LIMIT.C**. The user is prompted to provide values for the current input, the high limit, and the low limit. Once each of these values is entered, **main( )** is called passing in the user-supplied values. Upon return from **main( )** the range-corrected current value and the alarm bits are displayed to the screen. By providing different combinations of current value, high limit, and low limit, and by verifying the returned range-corrected current value and the alarm bits, the application **LIMIT** may be verified to operate correctly-all before it is ever placed into the Series 90-70 PLC.

Example 2: Batch Mode LIMIT

Example 2 uses batch methods for testing the application LIMIT. Rather than interactively requesting the input data and then printing the returned output data to the screen, this example will read all input values from an input data file. Likewise, example 2 will write all returned data to an output file. There is one additional difference to example 2's version of **BLKHARN.C**: the operation of the section of ladder logic which controls the enabling of the C block is duplicated. The duplicated ladder logic in **BLKHARN.C** mimics the enable on the CALL LIMIT function block. The source for this version of **BLKHARN.C** is shown below:

```
#include <stdio.h>
#include "plcc9070.h"

/*****************************************************************************
 *      GE Fanuc 90-70 PLC Memory Size Declarations                          *
 *****************************************************************************/
#define      L_MEM_WORDS        1024
#define      P_MEM_WORDS        1024
#define      R_MEM_WORDS        1024
#define      AI_MEM_WORDS         64
#define      AQ_MEM_WORDS         64
#define      I_MEM_BYTES         256
#define      Q_MEM_BYTES         256
#define      T_MEM_BYTES          32
#define      M_MEM_BYTES         512
#define      SA_MEM_BYTES         16
#define      SB_MEM_BYTES         16
#define      SC_MEM_BYTES         16
#define      S_MEM_BYTES          16
#define      G_MEM_BYTES         160
#define      GA_MEM_BYTES        160
#define      GB_MEM_BYTES        160
#define      GC_MEM_BYTES        160
#define      GD_MEM_BYTES        160
#define      GE_MEM_BYTES        160
#define      I_DIAG_BYTES        256
#define      Q_DIAG_BYTES        256
#define      I_TRANS_BYTES       256
#define      Q_TRANS_BYTES       256
#define      T_TRANS_BYTES        32
#define      M_TRANS_BYTES       512
#define      SA_TRANS_BYTES       16
#define      SB_TRANS_BYTES       16
#define      SC_TRANS_BYTES       16
#define      S_TRANS_BYTES        16
#define      G_TRANS_BYTES       160
#define      GA_TRANS_BYTES      160
#define      GB_TRANS_BYTES      160
#define      GC_TRANS_BYTES      160
#define      GD_TRANS_BYTES      160
#define      GE_TRANS_BYTES      160
#define      AI_DIAG_BYTES        64
#define      AQ_DIAG_BYTES        64
#define      RSB_MEM_BYTES       688
```

```
#include  “tharndat.inc”

static char *inname = “limit.dat”;    /* data input file */
FILE *infile;                         /* streamer for input file */
static char *outname = “limit.out”;   /* output file */
FILE *outfile;                        /* streamer for output file */

static short int    X1;               /* input 1 parameter */
static short int    X2;               /* input 2 parameter */
static short int    X3;               /* input 3 parameter */
static short int    Y1;               /* output 1 parameter */
static short int    Y2;               /* output 2 parameter */
static short int    Y3;               /* output 3 parameter */

void plc_sweep(void)
{
    int  ok;         /* used to store returned power flow -- the OK parameter */
    int  fstat;      /* returned status of file operations. */
    int  enabled;    /* used to compute enable status of function. */
    word temp[5];    /* used to temporarily hold input status information */
    int  loop;       /* local loop counter */
    char line[133];  /* used in skipping input data comment line */

/*
 *  open files:
 *     data input file
 *     log output file
 */
    infile = fopen(inname, “r”);      /* open input file */
    if (infile == NULL) {
       fprintf(stderr, “can’t open %s\n”,inname);
       exit(1);
    }

    outfile = fopen(outname, “w”);    /* open output file */
    if (outfile == NULL) {
       fprintf(stderr, “can’t open %s\n”, outname);
       exit(1);
    }

/*
 *  while not EOF
 *  read values to pass to limit function:
 *     HIGH_LIMIT, IN, LOW_LIMIT
 *  call limit using that set of values
 *  write input values, plus values returned from limit function:
 *     HIGH_LIMIT, IN, LOW_LIMIT, OUT, ALARM_BITS, power
 *  end
 */
    fprintf(outfile,“ ENABLE  HIGH_LIMIT    IN    LOW_LIMIT    OUT    HIGH_ALARM
  LOW_ALARM    OK\n\n”);
    if (fgets(line, 132, infile) == NULL) return; /* skip the comment line */
```

```
/*
 * To mimic PLC operation, initialize the FIRST EXE, FIRST SCAN,
 * ALW_ON, and ALW_OFF special status bits.
 */
       (*sb_mem)[0]  |= 0x01;        /* FST_SCN */
       (*sb_mem)[15] |= 0x01;        /* FST_EXE */
       (*sb_mem)[0]  |= 0x40;        /* ALW_ON  */
       (*sb_mem)[0]  &= ~0x80;       /* ALW_OFF */
       for(;;) {
              fstat = fscanf(infile, "%u %u %u %u %u ", &temp[0], &temp[1], &temp[2],
&temp[3], &temp[4]);
              if (fstat != 5) break;
              fstat = fscanf(infile, "%u %u %u\n", &X2, &X1, &X3);
              if (fstat != 3) break;
              for (loop = 0; loop < 5; loop++) {
                     if (temp[loop] == 0)
                            BIT_CLR_I(loop+1);
                     else
                            BIT_SET_I(loop+1);
              }
              /*   %I1   %I2  %I3   %I5        +---------------+
               * --| |---| |---| |---| |-------| CALL LIMIT_I  |
               *             |     |           |  (EXTERNAL)   |
               *             | %I4 |           |               |
               *             +-| |-+           | {parameters}  |
               *                               |    .          |
               *                               |    .          |
               *                               |    .          |
               *                               +---------------+
               */
              enabled = ( BIT_TST_I(1) & BIT_TST_I(2) &
                        (BIT_TST_I(3) | BIT_TST_I(4)) & BIT_TST_I(5) );
              if (enabled) {
                     ok = main(&X1, &X2, &X3, &Y1, &Y2, &Y3);
                     fprintf(outfile,"%4u    %6d    %6d    %6d     %6d   %6u       %6u
%6u\n",
                              enabled, X2, X1, X3, Y1, (Y2 & 0x01), ((Y2 & 0x02) >> 1),
(int)(ok==OK));
              } else {
                     fprintf(outfile,"%4u  \n", enabled);
              }
              /*
               * To further mimic PLC operation, now
               * clear the FIRST EXE and FIRST SCAN
               * special status bits.
               */
              (*sb_mem)[0]  &= 0xFE;
              (*sb_mem)[15] &= 0xFE;
       }
}
```

The input file used in **example2**, **LIMIT.DAT**, contains values for %I1 −> %I5, high limit, low limit, and the current value. **LIMIT.DAT** is provided as part of **example2**. The contents of **LIMIT.DAT** are shown below:

```
%I1 %I2 %I3 %I4 %I5 high_limit input_value low_limit expected_result

 1   1   1   0   1       30          20         10
 1   1   0   0   1       30          20         10
 1   0   1   1   1       30          20         10
 1   1   1   0   0       30          20         10
 1   1   0   1   1       30          20         10
 0   1   1   0   1       30          20         10
 1   1   0   1   1       30           5         10
 1   1   1   1   1       30          35         10
 1   1   1   1   1       30          30         10
 1   1   1   1   1       30          10         10
 1   1   1   1   1       10          20         30
 1   1   1   1   1        0           0          0
 1   1   1   1   1      -10         -20        -30
 1   1   1   1   1      -10          -5        -30
 1   1   1   1   1      -10         -35        -30
 1   1   1   1   1      -10         -10        -30
 1   1   1   1   1      -10         -30        -30
 1   1   1   1   1      -30         -20        -10
```

The output file, **LIMIT.OUT**, is created when **LIMIT.EXE** is executed. When the **LIMIT.DAT** file provided with **example2** is used as the input file and **LIMIT.EXE** created from **example2** is executed, **LIMIT.OUT** will contain the following:

```
 ENABLE  HIGH LIMIT  IN  LOW_LIMIT  OUT  HIGH_ALARM  LOW_ALARM  OK

   1        30       20     10       20      0          0        1
   0
   0
   0
   1        30       20     10       20      0          0        1
   0
   1        30        5     10       10      0          1        1
   1        30       35     10       30      1          0        1
   1        30       30     10       30      0          0        1
   1        30       10     10       10      0          0        1
   1        10       20     30       10      0          0        0
   1         0        0      0        0      0          0        1
   1       -10      -20    -30      -20      0          0        1
   1       -10       -5    -30      -10      1          0        1
   1       -10      -35    -30      -30      0          1        1
   1       -10      -10    -30      -10      0          0        1
   1       -10      -30    -30      -30      0          0        1
   1       -30      -20    -10      -30      0          0        0
```

The batch file commands which get user input to pass to the block and which display data upon return of the block are all placed into the file **BLKHARN.C**. Because the code contained in **BLKHARN.C** will never appear in the PLC, **BLKHARN.C** is not limited to the list of PLC supported C functions, as listed in appendix A, *Standard C Library Functions Supported in the Series 90-70 PLC*. Therefore, the use of **fopen( )**, **fclose( )**, **fscanf( )**, and **fprintf( )** in **BLKHARN.C** is allowed. Note that **BLKHARN.C** makes use of the **BIT_TST_I( )** macro when evaluating the simulated boolean ladder logic which is the enable to **LIMIT**.

Batch file testing provides several advantages over interactive testing:

1. If the test/debug/fix cycle must be repeated several times, the use of an input data file saves retyping all of the test cases. Also, the use of an input data file ensures that the same data cases will be attempted from test session to test session.
2. The input data file may be reviewed at any time to examine what cases are/were tested.
3. The output data file may also be viewed any time after the test has been run to examine what data values were tested and the corresponding results.

Section 2: Step-by-Step Example Session For Blocks

In this section, the process of building **LIMIT.C** and testing it (under MS-DOS), both interactively and in **BATCH** mode, are described in detail. Also covered are building **LIMIT.C** for execution in the PLC and testing **LIMIT** in the PLC. This section finishes with a discussion of when to use C blocks and when to use C FBKs.

Building and Debugging LIMIT under MS-DOS

Interactive Limit

To build **LIMIT** as in **example1**, make **\s9070c\example1** the active MS-DOS directory. To build a C block, invoke the C development software command file **MKDOS.BAT** specifying the application filename to be built:

```
c:\s9070c\example1>  mkdos limit
```

MKDOS will invoke the Microsoft C compiler and linker to compile both **LIMIT.C** and **BLKHARN.C** and then link the two resulting object files into one DOS-executable file **LIMIT.EXE**. **LIMIT.C** and **BLKHARN.C** are located in the **example1** subdirectory. When **MKDOS** executes, it creates a subdirectory under **example1** named **DOS**. The object files (***.OBJ**) created from the C compiler are both placed in the directory **\s9070c\example1\dos**. When the linker creates the executable file **LIMIT.EXE**, this file is also placed in the **\s9070c\example1\dos** subdirectory. The DOS subdirectory under **EXAMPLE1** is used to group together all of the files specific to the DOS-executable version of the **LIMIT** application.

To build a C FBK, copy **BLDVARS** from **\s9070c** into **\s9070c\example1**. Edit **BLDVARS** and change the line **TYPE=BLOCK** to **TYPE=FBK**. Then follow the procedure for C blocks. The same files exist in the same directories.

Once the executable **LIMIT.EXE** is created (all compiles and the link is complete without error), change the active MS-DOS directory to **s9070c\example1\dos** (for both C blocks and C FBKs). To begin testing **LIMIT**, type **limit** at the MS-DOS prompt:

```
c:\s9070c\example1\dos>  limit
```

LIMIT.EXE will proceed to prompt for the current input value, high limit, and low limit. After retrieving the input values, **LIMIT.EXE** will call **main( )** to execute the application and will then display the results from the call to **main( )**. A sample of one such session could be:

```
INPUT VALUE:  20
HIGH_LIMIT:  30
LOW_LIMIT:  10
OK: 1, OUT: 20, HIGH ALARM: 0, LOW ALARM: 0.

INPUT VALUE: _
```

The above sequence would be repeated until all required test cases had been attempted. To stop **LIMIT.EXE** and return to MS-DOS, press CTRL-C.

Batch Mode Limit

The process for building **LIMIT** in **example2** is the same as for **example1** for both C blocks and C FBKs. Recall that the difference between the two examples is actually in the code contained in the file **BLKHARN.C**. First, make **\s9070c\example2** the active MS-DOS directory. Next, alter the **BLDVARS** file as in example 1 if building a C FBK. Then invoke the C development software command file **MKDOS.BAT** specifying the application file name to be built:

```
c:\s9070c\example2>  mkdos limit
```

Once **LIMIT.EXE** is created (all compiles and the link complete without error), ensure that the active MS-DOS directory is **\s9070c\example2**. To begin testing **LIMIT** type **dos\limit** at the MS-DOS prompt:

```
c:\s9070c\example2>  dos\limit
```

Unlike **example1**, **LIMIT** in **example2** will not prompt the user for any information. In **example2**, **LIMIT** will open the input file **LIMIT.DAT**, open the output file **LIMIT.OUT**, and then repeatedly read a line from the input file, pass the read values to **main( )** to execute the application, and then write the results from the call to **main( )** to the output file. No messages are displayed to the screen unless a file operation error occurs. The contents of **LIMIT.OUT** after processing are shown below:

```
ENABLE HIGH_LIMIT  IN  LOW_LIMIT  OUT  HIGH_ALARM  LOW_ALARM  OK

  1       30       20     10       20       0          0        1
  0
  0
  0
  1       30       20     10       20       0          0        1
  0
  1       30        5     10       10       0          1        1
  1       30       35     10       30       1          0        1
  1       30       30     10       30       0          0        1
  1       30       10     10       10       0          0        1
  1       10       20     30       10       0          0        0
  1        0        0      0        0       0          0        1
  1      -10      -20    -30      -20       0          0        1
  1      -10       -5    -30      -10       1          0        1
  1      -10      -35    -30      -30       0          1        1
  1      -10      -10    -30      -10       0          0        1
  1      -10      -30    -30      -30       0          0        1
  1      -30      -20    -10      -30       0          0        0
```

Building and Debugging LIMIT for the PLC

When an application C block or C FBK is built for execution in the Series 90-70 PLC, the harness file is not included as part of the link process. The harness is not required because the actual target PLC will provide the PLC reference table allocations and will also provide the PLC sweep mechanism.

To illustrate building the example application program **LIMIT.C**, make **\s9070c\example1** the active MS-DOS directory.

Note

Since the file **LIMIT.C** is the same for both **example1** and for **example2** and the file **BLKHARN.C** is not needed, either the **example1** or **example2** subdirectories may be used to build the example program for execution in the Series 90-70 PLC.

To build the sample program **LIMIT.C** for execution in the Series 90-70 PLC, use the following procedures:

1. Invoke the MS-DOS batch file **MKPLC.BAT** located in the **\s9070c** directory. **MKPLC** requires the name of the application as a parameter:

```
c:\s9070c\example1>  mkplc  limit
```

2. **MKPLC** will invoke a Microsoft **NMAKE** makefile to compile **LIMIT.C** and then, in the case of the C block, link the compiled **LIMIT** with the PLC-specific runtime libraries. If the compile and subsequent link operations complete successfully (no error messages displayed on the screen), the file **LIMIT.EXE** will reside in the **\s9070c\example1\plc** subdirectory. The **LIMIT.OBJ** (the compiler output file), **LIMIT.PPP** (the linker output), and **LIMIT.MAP** (the linker output file) files produced in the build process will also reside in that subdirectory.
3. In the Logicmaster 90-70 software, create a new program folder. (Make sure to use a name other than **LIMIT** as the program folder name.)
4. Using the Logicmaster 90 librarian, add **\s9070c\example1\plc\limit.exe** to the library.
5. When adding to the library, you will be prompted for the number of input/output pairs **LIMIT** has. Respond with **3**.
6. With **LIMIT.EXE** in the library, proceed to import **LIMIT.EXE** into your newly created folder. With **LIMIT.EXE** successfully imported into your folder, you are now ready to create a ladder logic program to exercise **LIMIT**.
7. Within the Logicmaster 90-70 software, leave the librarian and enter the program editor. Since this is a new program folder, there will be nothing in the ladder logic program.

8. Insert the following rungs of ladder logic:

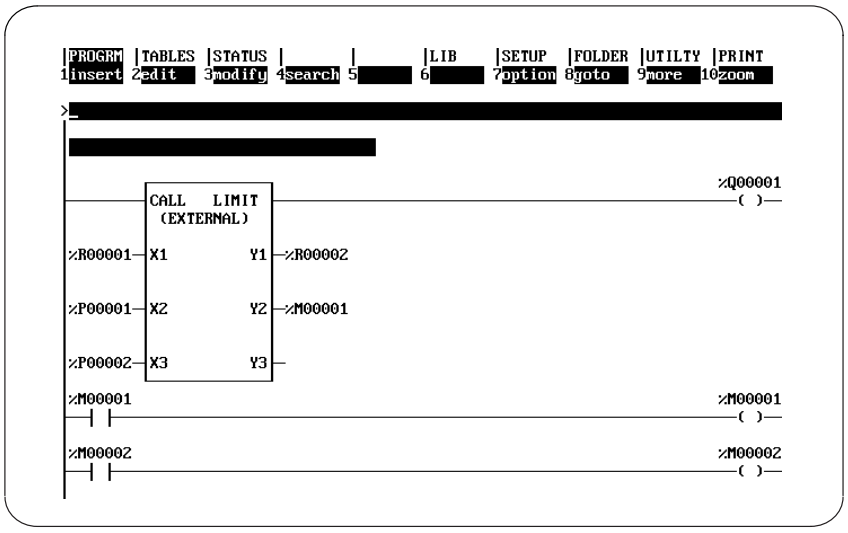


Figure 4-2. Ladder Logic for Testing Example Block LIMIT

9. Store the folder to the PLC using the Logicmaster 90 store function.
10. Once the program folder is successfully stored to the PLC, return to the program editor display.
11. With the Logicmaster 90 programmer online and equal to the PLC, place the PLC in **RUN** mode.

The execution of the C block **LIMIT** can be tested by placing different values into %R00001, %P00001, and %P00002. Recall that %R00001 is the current value, %P00001 is the high limit, and %P00002 is the low limit. As changes are made to %R00001, %P00001, and/or %P00002 watch as the values for %R00002, %M00001, and %M00002 change. Try entering different values for %R00001, %P00001, and %P00002.

Table 4-1. Test Values for LIMIT.EXE in the Series 90-70 PLC

Case	Input Parameters			Output Parameters		
	%R00001	%P00001	%P00002	%R00002	%M00001	%M00002
1	5	10	0	5	0	0
2	15	10	0	10	1	0
3	−1	10	0	0	0	1
4	0	10	0	0	0	0

C Blocks Versus C FBKs

The examples in this section could be built as either a C block and C FBK. There is no advantage to having a C block if the source can be built into a C FBK. The example, LIMIT, would be best built as a C FBK. However, if the source code parameters were changed to floating point, the compiler would insert library calls as shown in the following examples:

```
#include "plcc9070.h"

#define   HI_ALM_MSK    0x01
#define   LO_ALM_MSK    0x02

main( float      *input_value,     /* parameter X1 */
      float      *high_limit,      /* parameter X2 */
      float      *low_limit,       /* parameter X3 */
      float      *output_value,    /* parameter Y1  */
      short int  *alarm_bits,      /* parameter Y2  */
      void       *dummy )          /* parameter Y3  */

/*  check that all parameters have been provided
*/
    if ((input_value == NULL)  ||
        (high_limit == NULL)   ||
        (low_limit == NULL)    ||
        (output_value == NULL) ||
        (alarm_bits == NULL))
       return(ERROR);
{
/*  check for error condition high_limit < low_limit
 *  if error, set no power flow and return, don't change any outputs
 */
    if (*high_limit < *low_limit)
       return(ERROR);
    /*
     *  check for in out range
     *  if in exceeds either limit
     * use the exceeded limit value for the output
     *  else use the input value for the output
     *  always write both high and low alarm bits appropriately
     */
    if ((*input_value) > (*high_limit)){
       *output_value = *high_limit;   /* use high_limit for output */
       *alarm_bits |= HI_ALM_MSK;     /* turn high alarm bit on */
       *alarm_bits &= ~LO_ALM_MSK;    /* turn low alarm bit off */
    } else if ((*input_value) < (*low_limit)) {
       *output_value = *low_limit;    /* use low_limit for output */
       *alarm_bits &= ~HI_ALM_MSK;    /* turn high alarm bit off */
       *alarm_bits |= LO_ALM_MSK;     /* turn low alarm bit on */
    } else {
       *output_value = *input_value;  /* use in for output */
       *alarm_bits &= ~HI_ALM_MSK;    /* turn high alarm bit off */
       *alarm_bits &= ~LO_ALM_MSK;    /* turn low alarm bit off */
    }

    return(OK);
}
```

The compiler inserts runtime library calls to handle the floating point variables. This is not a problem for C FBKs under MS-DOS, because the runtime libraries are allowed. In the PLC build, however, these calls show up as unresolved externals. In this case the runtime library call cannot be eliminated and **LIMIT** should be built into a C block.

Eliminating Runtime Library Calls

It may be possible to eliminate runtime library calls. To find the calls, edit the **plcc9070.mkd** file in the **\s9070c** directory as follows:

1. In **plcc9070.mkd** there is a comment: "to provide a combined source and assembly file, uncomment the following line." The next line looks like this:

   ```
   #CFLAGS=-Fc
   ```

   Remove the # from the start of the file.

2. Delete **limit.obj** in the **plc** subdirectory, and rebuild **limit**.

3. There should be a file, **limit.cod**, in the same directory as **limit.c**. Edit this file and look for the unresolved external function calls. A function call will be a line with the word "call" followed by the function name.

4. Once this line is located, the line of code from the source file above the call is the most likely source of the runtime library call. Try to rewrite the line to not invoke the runtime library call.

Section 3: Installed Sample C FBK

An example for C FBK development is in the **\s9070c\exfbk** subdirectory. The harness developed for this example is a simple interactive interface. A batch mode version can be made using **example2** as a guideline. The harness FBKHARN.C is shown below:

```
#define _TESTHARN_
#include <stdio.h>
#include "plcc9070.h"
/*************************************************************
 *          GEFanuc 90-70 PLC Memory Size Declarations       *
 *************************************************************/
#define      L_MEM_WORDS           1024
#define      P_MEM_WORDS           1024
#define      R_MEM_WORDS           1024
#define      AI_MEM_WORDS            64
#define      AQ_MEM_WORDS            64
#define      I_MEM_BYTES            256
#define      Q_MEM_BYTES            256
#define      T_MEM_BYTES             32
#define      M_MEM_BYTES            512
#define      SA_MEM_BYTES            16
#define      SB_MEM_BYTES            16
#define      SC_MEM_BYTES            16
#define      S_MEM_BYTES             16
#define      G_MEM_BYTES            160
#define      GA_MEM_BYTES           160
#define      GB_MEM_BYTES           160
#define      GC_MEM_BYTES           160
#define      GD_MEM_BYTES           160
#define      GE_MEM_BYTES           160
#define      I_DIAG_BYTES           256
#define      Q_DIAG_BYTES           256
#define      I_TRANS_BYTES          256
#define      Q_TRANS_BYTES          256
#define      T_TRANS_BYTES           32
#define      M_TRANS_BYTES          512
#define      SA_TRANS_BYTES          16
#define      SB_TRANS_BYTES          16
#define      SC_TRANS_BYTES          16
#define      S_TRANS_BYTES           16
#define      G_TRANS_BYTES          160
#define      GA_TRANS_BYTES         160
#define      GB_TRANS_BYTES         160
#define      GC_TRANS_BYTES         160
#define      GD_TRANS_BYTES         160
#define      GE_TRANS_BYTES         160
#define      AI_DIAG_BYTES           64
#define      AQ_DIAG_BYTES           64
#define      RSB_MEM_BYTES          688
```

```
#include “tharndat.inc”

static short int                    X1;/* input 1 parameter */
static short int                    Y1;/* output 1 parameter */
void plc_sweep(void)
{
             for(;;) {
              printf(“Enter number : ”);
                       scanf(“%ud”,&X1);
                         main(&X1, &Y1);
              printf(“%u!: %u\n”,X1,Y1);
             }
}
```

The program the harness tests is **FACT.C**, shown below:

```
/*
 * This will calculate the factorial of a number X! where:
 *
 * 0! = 1
 * X! = (X-1)!*X
 */

#include “plcc9070.h”

int main(short int *X1,short int *Y1) {
       int I,J;
       short int xy;

       /*
        * 0! = 1
        */
       (*Y1) = 1;
       for(I=1;I<=(*X1);I++){

              /*
               * I! = (I-1)! * I
               */
              xy = 0;

              for(J=0;J<I;J++) xy += (*Y1);

              (*Y1) = xy;
       }
       return OK;
}
```

The example has a BLDVARS file included that defines this directory as a C FBK build directory. To build and debug this application follow the instructions for **example1**.

Section 4: Installed Sample Standalone C Program

An example for standalone C program development is in the \s9070c\exsap subdirectory. The harness developed for this example is a simple interactive interface. A batch mode version can be made using example2 as a guideline. The harness SAPHARN.C is shown below:

```
#define _TESTHARN_
#include <stdio.h>
#include "plcc9070.h"

/*****************************************************************
 *          GEFanuc 90-70 PLC Memory Size Declarations           *
 *****************************************************************/
#define      L_MEM_WORDS            1024
#define      P_MEM_WORDS            1024
#define      R_MEM_WORDS            1024
#define      AI_MEM_WORDS             64
#define      AQ_MEM_WORDS             64
#define      I_MEM_BYTES             256
#define      Q_MEM_BYTES             256
#define      T_MEM_BYTES              32
#define      M_MEM_BYTES             512
#define      SA_MEM_BYTES             16
#define      SB_MEM_BYTES             16
#define      SC_MEM_BYTES             16
#define      S_MEM_BYTES              16
#define      G_MEM_BYTES             160
#define      GA_MEM_BYTES            160
#define      GB_MEM_BYTES            160
#define      GC_MEM_BYTES            160
#define      GD_MEM_BYTES            160
#define      GE_MEM_BYTES            160
#define      I_DIAG_BYTES            256
#define      Q_DIAG_BYTES            256
#define      I_TRANS_BYTES           256
#define      Q_TRANS_BYTES           256
#define      T_TRANS_BYTES            32
#define      M_TRANS_BYTES           512
#define      SA_TRANS_BYTES           16
#define      SB_TRANS_BYTES           16
#define      SC_TRANS_BYTES           16
#define      S_TRANS_BYTES            16
#define      G_TRANS_BYTES           160
#define      GA_TRANS_BYTES          160
#define      GB_TRANS_BYTES          160
#define      GC_TRANS_BYTES          160
#define      GD_TRANS_BYTES          160
#define      GE_TRANS_BYTES          160
#define      AI_DIAG_BYTES            64
#define      AQ_DIAG_BYTES            64
#define      RSB_MEM_BYTES           688
```

```
#include “tharndat.inc”
typedef struct _io_spec_rec *_io_spec_rec_ptrs;
extern _io_spec_rec_ptrs spec_data_ptrs[16];

void plc_sweep(int argc,char *argv, char *envp)
{
         int I,J;
         unsigned int input;
         word *word_array;
         byte *byte_array;

/*
 *  Loop forever
 *      Prompt user for input spec values.
 *      Call main to execute block.
 *      Print out results.
 *  End loop
 */
         while (1)  {
                  /*
                   * Get input specs
                   */
                  for(I=0;I<8;I++) {

                           /* IF current input spec exists */
                           if(spec_data_ptrs[I]->data != NULL) {
                                    printf(“INPUT SPEC %d ”,I+1);

                                    /* This will get the input spec in bytes or words  */
                                    /* depending on the type.                          */
                                    if(spec_data_ptrs[I]->spec_type == BYTE_IO_SPEC_TYPE)
                                    {
                                             printf(“(in bytes): ”);
                                             byte_array = spec_data_ptrs[I]->data;
                                             for(J=0;J<(spec_data_ptrs[I]->byte_len);J++) {
                                                      scanf(“%d”,&input);
                                                      byte_array[J] = (byte)input;
                                             }
                                    }
                                    else {
                                             printf(“(in words): ”);
                                             word_array = (word *)(spec_data_ptrs[I]->data);
                                             for(J=0;J<(spec_data_ptrs[I]->byte_len/2);J++) {
                                                      scanf(“%d”,&input);
                                                      word_array[J] = input;
                                             }
                                    }
                           }/* end IF current input spec exists */
                  }

                  /*
                   * End of Get input specs
                   */

                  /* in1 is bubble sorted to out2              */
                  main();

                  /*
                   * Print output specs
                   */
                  for(I=0;I<8;I++) {
                           /* IF current output spec exists */
                           if(spec_data_ptrs[I+8]->data != NULL) {
                                    printf(“OUTPUT SPEC %d ”,I+1);

                                    /* This will get the input spec in bytes or words  */
                                    /* depending on the type.                          */
                                    if(spec_data_ptrs[I+8]->spec_type == BYTE_IO_SPEC_TYPE) {
                                             printf(“(in bytes): ”);
                                             byte_array = spec_data_ptrs[I+8]->data;
                                             for(J=0;J<spec_data_ptrs[I+8]->byte_len;J++) {
                                                      printf(“%d ”,byte_array[J]);
                                             }
                                    }
                                    else {
                                             printf(“(in words): ”);
                                             word_array = (word *)(spec_data_ptrs[I+8]->data);
                                             for(J=0;J<(spec_data_ptrs[I+8]->byte_len/2);J++)
                                                      printf(“%d ”,word_array[J]);
                                    }
                           }/* end IF current input spec exists */
                  }



                  printf(“\n”);
```

```
                /*
                 * End of Print output specs
                 */

                /*
                 * To further mimic PLC operation, now
                 * clear the FIRST SCAN status bit.
                 */
                __fst_scn = 0;
        }
}
```

The program the harness tests is **BUBBLE.C** shown below:

```
#include “plcc9070.h”
IN1_B(batch,10);
OUT2_B(queue,10);

int main() {
       int I,J,LOWVALUE;
       int i;
       byte temp;



       for (I=0;I < 9;I++) {
              LOWVALUE = I;
              for(J=I+1;J < 10;J++) {
                     if(batch[LOWVALUE] > batch[J]) LOWVALUE = J;
              }
              if(I != LOWVALUE) {
                     /* Swap the values */
                     temp = batch[LOWVALUE];
                     batch[LOWVALUE] = batch[I];
                     batch[I] = temp;
              }
       }
       for (I=0;I < 10;I++) {
              queue[I] = batch[I];
       }
}/* end bubble_sort */
```

The example has a BLDVARS file included that defines this directory as a standalone C program build directory. To build this application follow the instructions in the next section.

Section 5: Step-by-Step Example Session For Standalone C Program

Building and Debugging BUBBLE under MS-DOS

To build **BUBBLE**, make **EXSAP** the active directory. Once in the example directory, invoke the C development software command file **MKDOS.BAT** specifying the application to be built:

```
c:\s9070c\exsap> mkdos bubble
```

MKDOS will invoke the Microsoft C compiler and linker to compile both **BUBBLE.C** and **SAPHARN.C** and then link the two resulting object files into one DOS-executable file **BUBBLE.EXE**. **BUBBLE.C** and **SAPHARN.C** are located in the exsap subdirectory. When MKDOS executes, it creates a subdirectory under **EXSAP** named **DOS**. The object files (***.OBJ**) created from the C compile are both placed in the directory **\s9070c\exsap\dos**. When the linker created the executable file **BUBBLE.EXE**, this file is also placed in the **\s9070c\exsap\dos** subdirectory. The DOS subdirectory under **EXSAP** is used to group together all of the files specific to the DOS-executable version of the BUBBLE application.

Once the executable **BUBBLE.EXE** is created (all compiles and the link complete without error) C blocks and standalone C programs work the same way, change the active MS-DOS directory to **\s9070c\exsap\dos**. To begin testing **BUBBLE**, type bubble at the MS-DOS prompt:

```
c:\s9070c\exsap\dos> bubble
```

BUBBLE.EXE will proceed to prompt for the size of the list and the elements of that list. After getting the input values, **BUBBLE.EXE** will call main( ) to execute the application and will then display the results from the call to main( ). A sample of such a session could be:

```
INPUT SPEC 1 (in bytes): 4
3
2
1
5
6
7
8
0
9
OUTPUT SPEC 2 (in bytes):
0 1 2 3 4 5 6 7 8 9
INPUT SPEC 1 (in bytes): _
```

The above sequence would be repeated until all required test cases had been attempted. To stop **BUBBLE.EXE** and return to MS-DOS, press CTRL-C (press the Control key and C at the same time).

Building and Debugging BUBBLE for the PLC

When a standalone C program is built for execution in the Series 90-70 PLC, the file **SAPHARN.C** is not included as part of the link process. **SAPHARN** is not required because the actual target PLC will provide the PLC reference table allocations and will also provide the PLC sweep mechanism.

To illustrate building the example application program **BUBBLE.C**, make **\s9070c\exsap** the active MS-DOS directory.

To build the sample program **BUBBLE.C** for execution in the Series 90-70 PLC, create a folder in the Logicmaster 90-70 software (make sure to use a name other than **BUBBLE** as the folder name). Once the folder has been created, invoke the MS-DOS batch file **MKPLC.BAT** located in the **\s9070c** directory. **MKPLC** requires the name of the application as a parameter:

```
c:\s9070c\exsap>  mkplc  bubble
```

MKPLC will invoke a Microsoft **NMAKE** makefile to compile **BUBBLE.C** and then link the compiled **BUBBLE** with the PLC-specific runtime libraries. If the compile and subsequent link operations complete successfully (no error messages displayed on the screen), the build process will prompt for a folder name. Enter the folder name (use the full MS-DOS path if the folder is not a subdirectory of **\s9070c\exsap**). This will import the standalone C program into the folder created by Logicmaster. In the PLC subdirectory will be **BUBBLE.OBJ** (the compiler output file), **BUBBLE.PPP** (the linker output file), and **BUBBLE.MAP** (the linker mapfile). In the **\s9070c\exsap** directory there will be **BUBBLE.DBG**. In the folder there will be **BUBBLE.STA**, using this file the standalone C program can be moved to other folders using Logicmaster (from Logicmaster Main Menu F6 C utilities, F3 standalone merge). Do not try doing a DOS copy.

In the Logicmaster 90-70 Programming software, enter the Program Declaration Screen (press F1). Next, hit F1 to insert the standalone C program into the Program Declaration Screen. Then hit ESC to accept the program name. The highlight should remain on the standalone C program. Go to the control screen (press F9) to specify the program options. Go to the input specifications. On the first line there should already be a size of 10, enter %R1 for the memory start (since our parameter is an array of ten bytes). There should already be a size of 10 for the second output specification. Enter %R6. Hit ESC to accept the data.

Next, store the folder to the PLC using the Logicmaster 90 store function. Once the program folder is successfully stored to the PLC, return to the program editor display. With the Logicmaster 90 programmer online and equal to the PLC, place the PLC in **RUN** mode.

The execution of the standalone C program **BUBBLE** can be tested by placing different values into %R00001−%R00005. Recall that the ten bytes in %R00001−%R00005 is the batch list, and the ten bytes in %R00006−%R00010 is the sorted list. Try entering the different values for %R00001−%R00005.

Table 4-2. Test Values

Case	Input List	Output List
1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
2	9, 8, 7, 6, 5, 4, 3, 2, 1, 0	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
3	5, 4, 3, 2, 1, 0, 6, 7, 8, 9	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
4	9, 8, 7, 6, 5, 0, 1, 2, 3 ,4	0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Chapter 5 Example C Series 90-30 Application Development

Section 1: Installed Sample Blocks

In the C development software directory, **\s9030c**, there are two subdirectories which have examples of simple blocks: **example1** and **example2**. Each subdirectory contains the block source file **LIMIT.C**. **LIMIT.C** provides an illustration of a function which could be written in ladder logic, but is, instead, written in C.

LIMIT is an application program which range checks a current value against a low limit and a high limit and returns, if necessary, a corrected value which is within the range. If the current value is lower than the low limit or higher than the high limit, then **LIMIT** will return low limit or high limit, respectively. A ladder logic call to this external block could look like the following:

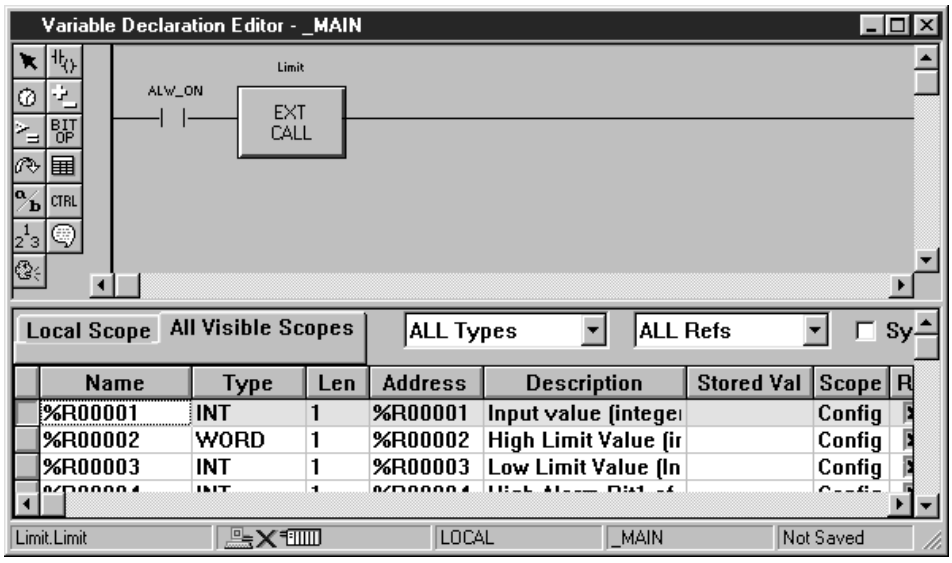


Figure 5-1. Ladder Logic Call to Example Block

The C block LIMIT is shown below:

```
#include "plcc9030.h"

#define HI_ALM_MSK      0x01
#define LO_ALM_MSK      0x02

EXE_stack_size = 2048;

main(void)
{

   short int *input_value    = (short int *)__ri_mem;
   short int *high_limit     = (short int *)__ri_mem+1;
   short int *low_limit      = (short int *)__ri_mem+2;
   short int *output_value   = (short int *)__ri_mem+3;
   unsigned char *alarm_bits = (unsigned char *)__mb_mem;

/*  check that all parameters have been provided
*/
    if ((input_value == NULL)  ||
        (high_limit == NULL)   ||
        (low_limit == NULL)    ||
        (output_value == NULL) ||
        (alarm_bits == NULL))
       return(ERROR);

/*  check for error condition high_limit < low_limit
 *  if error, set no power flow and return, don't change any outputs
 */
    if (*high_limit < *low_limit)
       return(ERROR);
/*
 *  check for in out range
 *  if in exceeds either limit
 *      use the exceeded limit value for the output
 *  else use the input value for the output
 *  always write both high and low alarm bits appropriately
 */
    if ((*input_value) > (*high_limit)){
       *output_value = *high_limit;     /* use high_limit for output */
       *alarm_bits |= HI_ALM_MSK;       /* turn high alarm bit on */
       *alarm_bits &= ~LO_ALM_MSK;      /* turn low alarm bit off */
    } else if ((*input_value) < (*low_limit)) {
       *output_value = *low_limit;      /* use low_limit for output */
       *alarm_bits &= ~HI_ALM_MSK;      /* turn high alarm bit off */
       *alarm_bits |= LO_ALM_MSK;       /* turn low alarm bit on */
    } else {
       *output_value = *input_value;    /* use in for output */
       *alarm_bits &= ~HI_ALM_MSK;      /* turn high alarm bit off */
       *alarm_bits &= ~LO_ALM_MSK;      /* turn low alarm bit off */
    }

    return(OK);
}
```

In the C subroutine block LIMIT above, %R00001 contains the current value, %R00002 containg the high limit value, and %R00003 contains the low limit value. The return parameters from the C subroutine block LIMIT above are the range corrected current value in %R00004, an indication of the input current value being over the high limit in %M00001, and an indication of the input current value being under the low limit in %M00002 (bit 2 of the 16 bit value starting with %M00001).

The following two sections will describe how to interactively test a block (example 1) and how to batch test a block (example 2). The differences between the two methods (interactive and batch) will be shown in the test harness file, **BLKHARN.C**.

Example 1: Interactive LIMIT

As stated at the beginning of this chapter, **LIMIT.C** is the same in both this example and in **example2**. The difference between the two examples is the manner in which testing under MS-DOS may be accomplished. In this example, you will use interactive methods to request user input and display data values.

The interactive commands which get user input to pass to the block and which display data upon return of the block are all placed into the file **BLKHARN.C**. Recall that **BLKHARN.C** is the MS-DOS test harness which is linked with the application (in this case, **LIMIT**) to create an MS-DOS executable and which provides the basic PLC sweep mechanism. The intent of **BLKHARN.C** is not only to provide the basic sweep mechanism, but also to contain any and all debugging code.

The code contained in **BLKHARN.C** for **example1** is shown below:

```
#define _TESTHARN_
#include <stdio.h>
#include "plcc9030.h"

/*****************************************************************
 *          GEFanuc 90-30 PLC Memory Size Declarations           *
 *****************************************************************/
#define         R_MEM_WORDS             9999
#define         AI_MEM_WORDS            2048
#define         AQ_MEM_WORDS             512
#define         I_MEM_BYTES              256
#define         Q_MEM_BYTES              256
#define         T_MEM_BYTES               32
#define         M_MEM_BYTES              512
#define         SA_MEM_BYTES              16
#define         SB_MEM_BYTES              16
#define         SC_MEM_BYTES              16
#define         S_MEM_BYTES               16
#define         G_MEM_BYTES              160
#define         I_TRANS_BYTES            256
#define         Q_TRANS_BYTES            256
#define         T_TRANS_BYTES             32
#define         M_TRANS_BYTES            512
#define         SA_TRANS_BYTES            16
#define         SB_TRANS_BYTES            16
#define         SC_TRANS_BYTES            16
#define         S_TRANS_BYTES             16
#define         G_TRANS_BYTES            160
```

```
#include        "tharndat.inc"

void plc_sweep(int argc,char *argv, char *envp)
{
    int  ok;         /* used to store returned power flow -- the OK parameter */

/*
 * Modify stdout to use NON-buffered i/o
 */

    setbuf( stdout, NULL );


/*
 * To mimic PLC operation, initialize the FIRST SCAN, LAST SCAN,
 * ALW_ON, and ALW_OFF special status bits.
 */

  SB(1)  |= 0x01;          /* FST_SCN */
  SB(1)  |= 0x02;          /* LST_SCN */
  SB(1)  |= 0x40;          /* ALW_ON  */
  SB(1)  &= ~0x80;         /* ALW_OFF */

/*
 *  Loop forever
 *      Prompt user for IN (%R0001), HIGH_LIMIT (%R0002), LOW_LIMIT (%R0003).
 *      Call main to execute block.
 *      Print out results.
 *  End loop
 */



    while (1)  {

       printf("INPUT VALUE: ");
       scanf("%d",(short int *)__ri_mem);   /* %R0001 */
       printf("HIGH_LIMIT: ");
       scanf("%d",(short int *)__ri_mem+1); /* %R0002 */
       printf("LOW_LIMIT: ");
       scanf("%d",(short int *)__ri_mem+2); /* %R0003 */

       ok = main();

       /* From description of LIMIT in top of file LIMIT.C  */
       /* the alarm bits returned from LIMIT are in %M0001  */
       /* and %M0002 reference.                             */
       printf("OK: %d, OUT: %d, HIGH ALARM: %d, LOW_ALARM: %d.\n\n",
          (int)(ok==OK),
          RW(4),
          (MB(1) & 0x01),
          ((MB(1) & 0x02) >> 1) );

       /*
        * To further mimic PLC operation, now
        * clear the FIRST SCAN special status bits.
        */
       SB(1) &= 0xFE;
    }
}
```

This version of **BLKHARN.C** allows for the interactive testing of the **LIMIT** application program. The interactive nature of this version of **BLKHARN** is seen in the prompting for input values at the beginning of the **while( )** loop (**printf( )** and **scanf( )** pairs) and in the single **printf( )** statement which appears inside the **while( )** loop immediately after the call to **main( )**. The **printf( )** and **scanf( )** statements in this version of **BLKHARN.C** are written specifically for testing the example program **LIMIT.C**. The user is prompted to provide values for the current input, the high limit, and the low limit. Once each of these values is entered, **main( )** is called passing in the user-supplied values. Upon return from **main( )** the range-corrected current value and the alarm bits are displayed to the screen. By providing different combinations of current value, high limit, and low limit, and by verifying the returned range-corrected current value and the alarm bits, the application **LIMIT** may be verified to operate correctly-all before it is ever placed into the Series 90-30 PLC.

Example 2: Batch Mode LIMIT

Example 2 uses batch methods for testing the application LIMIT. Rather than interactively requesting the input data and then printing the returned output data to the screen, this example will read all input values from an input data file. Likewise, example 2 will write all returned data to an output file. There is one additional difference to example 2's version of **BLKHARN.C**: the operation of the section of ladder logic which controls the enabling of the C block is duplicated. The duplicated ladder logic in **BLKHARN.C** mimics the enable on the CALL LIMIT function block. The source for this version of **BLKHARN.C** is shown below:

```
#define   _TESTHARN_
#include <stdio.h>
#include "plcc9030.h"

/*****************************************************************
    GEFanuc 90-30 PLC Memory Size Declarations
*****************************************************************/
#define      R_MEM_WORDS        9999
#define      AI_MEM_WORDS       2048
#define      AQ_MEM_WORDS        512
#define      I_MEM_BYTES         256
#define      Q_MEM_BYTES         256
#define      T_MEM_BYTES          32
#define      M_MEM_BYTES         512
#define      SA_MEM_BYTES         16
#define      SB_MEM_BYTES         16
#define      SC_MEM_BYTES         16
#define      S_MEM_BYTES          16
#define      G_MEM_BYTES         160
#define      I_TRANS_BYTES       256
#define      Q_TRANS_BYTES       256
#define      T_TRANS_BYTES        32
#define      M_TRANS_BYTES       512
#define      SA_TRANS_BYTES       16
#define      SB_TRANS_BYTES       16
#define      SC_TRANS_BYTES       16
#define      S_TRANS_BYTES        16
#define      G_TRANS_BYTES       160

#include  "tharndat.inc"

static char *inname = "limit.dat";      /* data input file */
FILE *infile;                           /* streamer for input file */
static char *outname = "limit.out";     /* output file */
FILE *outfile;                          /* streamer for output file */

/*
 *  %R0001 (__ri_mem)   = Input value (integer)
 *  %R0002 (__ri_mem+1) = High limit value (integer)
 *  %R0003 (__ri_mem+2) = Low limit value (integer)
 *  %R0004 (__ri_mem+3) = Output value (integer)
 *  %M0001 (__mb_mem)   = High alarm (bit1 of reference)
 *  %M0002 (__mb_mem)   = Low alarm (bit2 of reference)
 *
 */
```

```
#include  "tharndat.inc"

static char *inname = "limit.dat";      /* data input file */
FILE *infile;                           /* streamer for input file */
static char *outname = "limit.out";     /* output file */
FILE *outfile;                          /* streamer for output file */

/*
 *  %R0001 (__ri_mem)   = Input value (integer)
 *  %R0002 (__ri_mem+1) = High limit value (integer)
 *  %R0003 (__ri_mem+2) = Low limit value (integer)
 *  %R0004 (__ri_mem+3) = Output value (integer)
 *  %M0001 (__mb_mem)   = High alarm (bit1 of reference)
 *  %M0002 (__mb_mem)   = Low alarm (bit2 of reference)
 *
 */

void plc_sweep(void)
{
    int  ok; /* used to store returned power flow -- the OK parameter */
    int  fstat;      /* returned status of file operations. */
    int  enabled;    /* used to compute enable status of function. */
    word temp[5];    /* used to temporarily hold input status information */
    int  loop;       /* local loop counter */
    char line[133]; /* used in skipping input data comment line */

/*
 *  open files:
 *     data input file
 *     log output file
 */
    infile = fopen(inname, "r");    /* open input file */
    if (infile == NULL) {
       fprintf(stderr, "can't open %s\n",inname);
       exit(1);
    }

    outfile = fopen(outname, "w");  /* open output file */
    if (outfile == NULL) {
       fprintf(stderr, "can't open %s\n", outname);
       exit(1);
    }
/*
 *  while not EOF
 *  read values to pass to limit function:
 *     HIGH_LIMIT, IN, LOW_LIMIT
 *  call limit using that set of values
 *  write input values, plus values returned from limit function:
 *     HIGH_LIMIT, IN, LOW_LIMIT, OUT, ALARM_BITS, power
 *  end
```

```
*/
        fprintf(outfile," ENABLE HIGH_LIMIT IN LOW_LIMIT OUT HIGH_ALARM LOW_ALARM OK\n\n");
        if (fgets(line, 132, infile) == NULL) return;        /* skip the comment line */

/*
 * To mimic PLC operation, initialize the FIRST EXE, FIRST SCAN,
 * ALW_ON, and ALW_OFF special status bits.
 */

        SB(1) |= 0x01;          /* FST_SCN */
        SB(1) |= 0x02;          /* LST_SCN */
        SB(1) |= 0x40;          /* ALW_ON  */
        SB(1) &= ~0x80;         /* ALW_OFF */

        for(;;) {
                fstat = fscanf(infile, "%u %u %u %u %u ", &temp[0], &temp[1], &temp[2],
&temp[3], &temp[4]);
                if (fstat != 5) break;

                fstat = fscanf(infile, "%u %u %u\n",
                                (short int *)__ri_mem+1 /* %R0002 */,
                                (short int *)__ri_mem   /* %R0001 */,
                                (short int *)__ri_mem+2 /* %R0003 */);
                if (fstat != 3) break;

                for (loop = 0; loop < 5; loop++) {
                        if (temp[loop] == 0)
                                BIT_CLR_I(loop+1);
                        else
                                BIT_SET_I(loop+1);
                }

                enabled = ( BIT_TST_I(1) & BIT_TST_I(2) &
                          (BIT_TST_I(3) | BIT_TST_I(4)) & BIT_TST_I(5) );
                if (enabled) {
                ok = main();
                fprintf(outfile,"%4u    %6d    %6d    %6d     %6d   %6u       %6u    %6u\n",
                 enabled, RI(2), RI(1), RI(3), RI(4), (MB(1) & 0x01), ((MB(1) & 0x02) >> 1),
                (int)(ok==OK));
                } else {
                        fprintf(outfile,"%4u  \n", enabled);
                }

                /*
                 * To further mimic PLC operation, now
                 * clear the FIRST SCAN special status bits. */
                SB(1) &= 0xFE;
        }
}
```

The input file used in **example2**, **LIMIT.DAT**, contains values for %I1 −> %I5, high limit, low limit, and the current value. **LIMIT.DAT** is provided as part of **example2**. The contents of **LIMIT.DAT** are shown below:

```
%I1 %I2 %I3 %I4 %I5 high_limit input_value low_limit expected_result

 1   1   1   0   1       30          20         10
 1   1   0   0   1       30          20         10
 1   0   1   1   1       30          20         10
 1   1   1   0   0       30          20         10
 1   1   0   1   1       30          20         10
 0   1   1   0   1       30          20         10
 1   1   0   1   1       30           5         10
 1   1   1   1   1       30          35         10
 1   1   1   1   1       30          30         10
 1   1   1   1   1       30          10         10
 1   1   1   1   1       10          20         30
 1   1   1   1   1        0           0          0
 1   1   1   1   1      -10         -20        -30
 1   1   1   1   1      -10          -5        -30
 1   1   1   1   1      -10         -35        -30
 1   1   1   1   1      -10         -10        -30
 1   1   1   1   1      -10         -30        -30
 1   1   1   1   1      -30         -20        -10
```

The output file, **LIMIT.OUT**, is created when **LIMIT.EXE** is executed. When the **LIMIT.DAT** file provided with **example2** is used as the input file and **LIMIT.EXE** created from **example2** is executed, **LIMIT.OUT** will contain the following:

```
 ENABLE  HIGH LIMIT  IN  LOW_LIMIT  OUT  HIGH_ALARM  LOW_ALARM  OK

   1        30       20     10       20      0          0        1
   0
   0
   0
   1        30       20     10       20      0          0        1
   0
   1        30        5     10       10      0          1        1
   1        30       35     10       30      1          0        1
   1        30       30     10       30      0          0        1
   1        30       10     10       10      0          0        1
   1        10       20     30       10      0          0        0
   1         0        0      0        0      0          0        1
   1       -10      -20    -30      -20      0          0        1
   1       -10       -5    -30      -10      1          0        1
   1       -10      -35    -30      -30      0          1        1
   1       -10      -10    -30      -10      0          0        1
   1       -10      -30    -30      -30      0          0        1
   1       -30      -20    -10      -30      0          0        0
```

The batch file commands which get user input to pass to the block and which display data upon return of the block are all placed into the file **BLKHARN.C**. Because the code contained in **BLKHARN.C** will never appear in the PLC, **BLKHARN.C** is not limited to the list of PLC supported C functions, as listed in Appendix A, "Standard C Library Functions Supported in the Series 90 PLC." Therefore, the use of **fopen( )**, **fclose( )**, **fscanf( )**, and **fprintf( )** in **BLKHARN.C** is allowed. Note that **BLKHARN.C** makes use of the **BIT_TST_I( )** macro when evaluating the simulated boolean ladder logic which is the enable to **LIMIT**.

Batch file testing provides several advantages over interactive testing:

1. If the test/debug/fix cycle must be repeated several times, the use of an input data file saves retyping all of the test cases. Also, the use of an input data file ensures that the same data cases will be attempted from test session to test session.
2. The input data file may be reviewed at any time to examine what cases are/were tested.
3. The output data file may also be viewed any time after the test has been run to examine what data values were tested and the corresponding results.

Section 2: Step-by-Step Example Session For Blocks

In this section, the process of building **LIMIT.C** and testing it (under MS-DOS), both interactively and in **BATCH** mode, are described in detail. Also covered are building **LIMIT.C** for execution in the PLC and testing **LIMIT** in the PLC.

Building and Debugging LIMIT under MS-DOS

Interactive Limit

To build **LIMIT** as in **example1**, make **\s9030c\example1** the active MS-DOS directory. To build a C block, invoke the C development software command file **MK3DOS.BAT** specifying the application filename to be built:

```
c:\s9030c\example1>  mk3Dos limit
```

MK3DOS will invoke the Microsoft C compiler and linker to compile both **LIMIT.C** and **BLKHARN.C** and then link the two resulting object files into one DOS-executable file **LIMIT.EXE**. **LIMIT.C** and **BLKHARN.C** are located in the **example1** subdirectory. When **MK3DOS** executes, it creates a subdirectory under **example1** named **DOS**. The object files (***.OBJ**) created from the C compiler are both placed in the directory **\s9030c\example1\dos**. When the linker creates the executable file **LIMIT.EXE**, this file is also placed in the **\s9030c\example1\dos** subdirectory. The DOS subdirectory under **EXAMPLE1** is used to group together all of the files specific to the DOS-executable version of the **LIMIT** application.

Once the executable **LIMIT.EXE** is created (all compiles and the link is complete without error), change the active MS-DOS directory to **s9030c\example1\dos** (for both C blocks). To begin testing **LIMIT**, type **limit** at the MS-DOS prompt:

```
c:\s9030c\example1\dos>  limit
```

LIMIT.EXE will proceed to prompt for the current input value, high limit, and low limit. After retrieving the input values, **LIMIT.EXE** will call **main( )** to execute the application and will then display the results from the call to **main( )**. A sample of one such session could be:

```
INPUT VALUE:  20
HIGH_LIMIT:  30
LOW_LIMIT:  10
OK: 1, OUT: 20, HIGH ALARM: 0, LOW ALARM: 0.

INPUT VALUE: _
```

The above sequence would be repeated until all required test cases had been attempted. To stop **LIMIT.EXE** and return to MS-DOS, press CTRL-C.

Batch Mode Limit

The process for building **LIMIT** in **example2** is the same as for **example1**. Recall that the difference between the two examples is actually in the code contained in the file **BLKHARN.C**. First, make **\s9030c\example2** the active MS-DOS directory. Then invoke the C development software command file **MK3DOS.BAT** specifying the application file name to be built:

```
c:\s9030c\example2>  mk3Dos limit
```

Once **LIMIT.EXE** is created (all compiles and the link complete without error), ensure that the active MS-DOS directory is **\s9030c\example2**. To begin testing **LIMIT** type **dos\limit** at the MS-DOS prompt:

```
c:\s9030c\example2>  dos\limit
```

Unlike **example1**, **LIMIT** in **example2** will not prompt the user for any information. In **example2**, **LIMIT** will open the input file **LIMIT.DAT**, open the output file **LIMIT.OUT**, and then repeatedly read a line from the input file, pass the read values to **main( )** to execute the application, and then write the results from the call to **main( )** to the output file. No messages are displayed to the screen unless a file operation error occurs. The contents of **LIMIT.OUT** after processing are shown below:

```
ENABLE HIGH_LIMIT  IN  LOW_LIMIT  OUT  HIGH_ALARM  LOW_ALARM  OK

  1       30       20     10       20       0          0        1
  0
  0
  0
  1       30       20     10       20       0          0        1
  0
  1       30        5     10       10       0          1        1
  1       30       35     10       30       1          0        1
  1       30       30     10       30       0          0        1
  1       30       10     10       10       0          0        1
  1       10       20     30       10       0          0        0
  1        0        0      0        0       0          0        1
  1      -10      -20    -30      -20       0          0        1
  1      -10       -5    -30      -10       1          0        1
  1      -10      -35    -30      -30       0          1        1
  1      -10      -10    -30      -10       0          0        1
  1      -10      -30    -30      -30       0          0        1
  1      -30      -20    -10      -30       0          0        0
```

Building and Debugging LIMIT for the PLC

When an application C subroutine block is built for execution in the Series 90-30 PLC, the harness file is not included as part of the link process. The harness is not required because the actual target PLC will provide the PLC reference table allocations and will also provide the PLC sweep mechanism.

To illustrate building the example application program **LIMIT.C**, make **\s9030c\example1** the active MS-DOS directory.

Note

Since the file **LIMIT.C** is the same for both **example1** and for **example2** and the file **BLKHARN.C** is not needed, either the **example1** or **example2** subdirectories may be used to build the example program for execution in the Series 90-30 PLC.

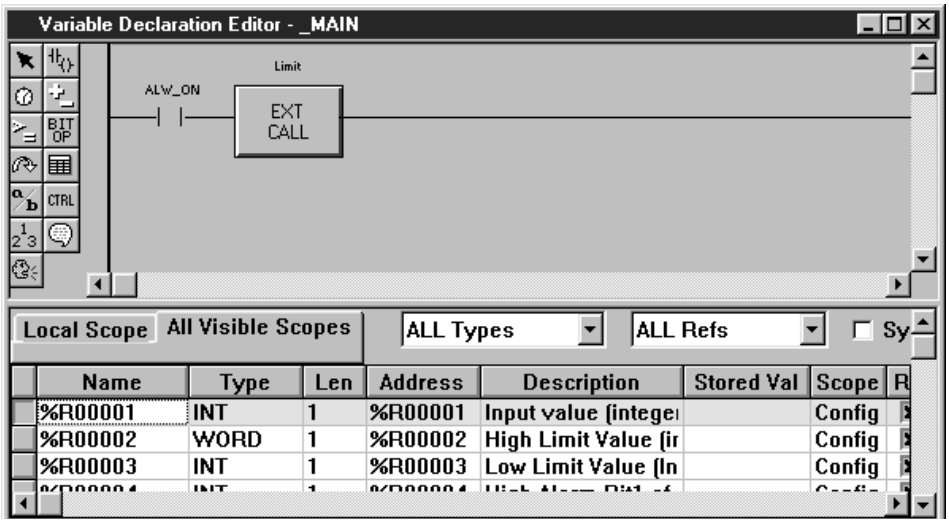
To build the sample program **LIMIT.C** for execution in the Series 90-30 PLC, use the following procedures:

1. Invoke the MS-DOS batch file **MK3PLC.BAT** located in the **\s9030c** directory. **MK3PLC** requires the name of the application as a parameter:

```
c:\s9030c\example1>  mk3plc  limit
```

2. **MK3PLC** will invoke a Microsoft **NMAKE** makefile to compile **LIMIT.C** and then, in the case of the C block, link the compiled **LIMIT** with the PLC-specific runtime libraries. If the compile and subsequent link operations complete successfully (no error messages displayed on the screen), the file **LIMIT.EXE** will reside in the **\s9030c\example1\plc** subdirectory. The **LIMIT.OBJ** (the compiler output file), **LIMIT.PPP** (the linker output), **LIMIT.LST** (the compiler source listing) and **LIMIT.MAP** (the linker output file) files produced in the build process will also reside in that subdirectory.

3. Use the Windows-based programming software to insert the C block or C Main program. (Refer to the "Working with C Programs and Blocks in the Windows-based Programming Software" section on page 3-97 and following for detailed instructions on placing C blocks and C MAIN programs in your the Windows-based programming software Equipment Folder. Also, refer to online help if you do not know how to use the Call function shown below.)



4. Store the folder to the PLC using the the Windows-based programming software store function.
5. Once the program folder is successfully stored to the PLC, return to the program editor display.
6. With the Windows-based programming software online and equal to the PLC, place the PLC in **RUN** mode.

The execution of the C block **LIMIT** can be tested by placing different values into %R0001, %R0002, and %R0003. Recall that %R0001 is the current value, %R0002 is the high limit, and %R0003 is the low limit. As changes are made to %R0001, %R0002, and/or %R0003 watch as the values for %R0004, %M00001, and %M00002 change. Try entering different values for %R0001, %R0002, and %R0003.

Table 5-1. Test Values for LIMIT.EXE in the Series 90-30 PLC

Case	Input Parameters			Output Parameters		
	%R0001	%R0002	%R0003	%R0004	%M00001	%M00002
1	5	10	0	5	0	0
2	15	10	0	10	1	0
3	−1	10	0	0	0	1
4	0	10	0	0	0	0

Chapter 6

C Application Development Using Multiple C Source Files

The C Programmer's Toolkit is organized to create C applications from a single C source file or multiple C source files. For most applications, a single C source file is sufficient to contain all required logic. However, some applications may require the use of multiple C source files. This may be especially true if more than one software developer is creating the C application source code. To aid in the development of C applications which are made from multiple C source files, an option for multiple source files has been added to this version of the toolkit. The subdirectory **`\s9070c\multisrc`** (for a 90-70 application) or **`\s9030c\multisrc`** (for a 90-30 application) and the files contained therein are provided as part of the C Programmer's Toolkit to demonstrate this new feature.

Overview

The **`sources`** file is used by the build procedure to compile and link multiple source files into one C application. The **`sources`** file specifies the files required to build the application, and identifies the source file that contains main(). The names and numbers of files can be expanded or reduced by editing the **`sources`** file.

Once the **`sources`** file is edited to reflect the structure of your multi-source application, the application is built by running **`MKPLC.BAT`** (90-70), **`MK3PLC.BAT`** (90-30), **`MKPLC7.BAT`** (90-70), **`MK3PLC7.BAT`** (90-30), **`MKDOS.BAT`** (90-70), **`MK3DOS.BAT`** (90-30), or **`MKDOS7.BAT`** (90-70), **`MK3DOS7.BAT`** (90-30) depending on your application. **`MKPLC.BAT`** and **`MKDOS.BAT`** create C blocks and standalone C programs that do not require a floating point coprocessor. **`MKPLC7.BAT`** and **`MKDOS7.BAT`** are used for applications that take advantage of the math coprocessor.

The **`BLDVARS`** file specifies the type of application that will be built: C block, C FBK, or standalone C program. To change the application type that is built, change the type field from TYPE=BLOCK to either TYPE=FBK or TYPE=STANDALONE (note all capital letters are used).

Creating a Multiple C Source Application SOURCES File

The build procedure will need to know the names of files used by the application and the name of the file that contains the main function. This information is contained in the **`sources`** file in the application directory. An example of the **`sources`** file is shown below. The **`MAINFILE`** line identifies the file that contains the main function (main.c) in the example. The **`FILENAMES`** line lists the files required for the application (main.c,

SRC1.C, SRC2.C, SRC3.C, SRC4.C, SRC5.C) in the example. The **.C** endings are necessary for the build procedure to manipulate the filename during the procedure. The harness filenames do **not** need to be in the **FILENAMES** list; the build procedure will include the appropriate harness file automatically on an MS-DOS build. If the filenames are too long to fit on one line, add the continuation character at the end of the line as shown in the example below.

```
MAINFILE=main.c
FILENAMES=main.c src1.c src2.c sr3.c src4.c\
   src5.c
```

The continuation character is a backslash followed by a carriage return, no spaces.

Note

All source files must be in the same directory. Do not designate a path for some of them.

Invoking a Multiple C Source Application Build

To invoke the multiple C source application build, use the **MKDOS.BAT** (90-70), **MK3DOS.BAT** (90-30), or **MKDOS7.BAT** (90-70), **MK3DOS7.BAT** (90-30), **MKPLC.BAT** (90-70), **MK3PLC.BAT** (90-30), **MKPLC7.BAT** (90-70), or **MK3PLC7.BAT** (90-30) programs, but do not specify a source filename on the command line.

I/O Specifications in Standalone C Programs (Series 90-70 Only)

I/O specifications may be used in any of the source files. The specification macro for declaring a particular I/O specification must be used in only one source file. If an I/O specification is to be used in multiple source files, the I/O specification macro should be used in one source file and the I/O specification array should be referenced as an **extern** in the other source files.

To illustrate, **a Series 90-70 example** of an external reference to an I/O specification is shown in error.c: (Note that the files shown below are not part of the MULTISRC example on the disk.)

```
/*
 * limit.c
*/

#include “plcc9070.h”

IN1_W(input_value,1);
IN2_W(high_limit,1);
IN3_W(low_limit,1);
OUT1_W(output_value,1);
OUT2_B(alarm_bits,1);

int check_error;

main( )

{
    /*  check for error condition high_limit < low_limit
     *  if error, don’t change any outputs
     */
    if(check_error() == ERROR) return(ERROR);

{
/*  check that all parameters have been provided
*/
       if ((input_value == NULL)  ||
           (high_limit == NULL) ||
           (low_limit == NULL) ||
           (output_value == NULL)  ||
           (alarm_bits == NULL)) ||
                 return (ERROR);

    /*
     *  check for in out range
     *  if in exceeds either limit
     * use the exceeded limit value for the output
     *  else use the input value for the output
     *  always write both high and low alarm bits appropriately
     */
    if ((*input_value) > (*high_limit)){
       *output_value = *high_limit;   /* use high_limit for output */
       BIT_SET(alarm_bits,1);         /* turn high alarm bit on */
       BIT_CLR(alarm_bits,2);         /* turn low alarm bit off */
    } else if ((*input_value) < (*low_limit)) {
       *output_value = *low_limit;    /* use low_limit for output */
       BIT_CLR(alarm_bits,1);         /* turn high alarm bit off */
       BIT_SET(alarm_bits,2);         /* turn low alarm bit on */
    } else {
       *output_value = *input_value;  /* use in for output */
       BIT_CLR(alarm_bits,1);         /* turn high alarm bit off */
       BIT_CLR(alarm_bits,2);         /* turn low alarm bit off */
    }

    return(OK);
}
```

```
/*
 * error.c
*/

#include “plcc9070.h”

extern word high_limit[];
extern word low_limit[];

int check_error()
{
    if(high_limit < low_limit) return(ERROR);

    return(OK);
}
```

Chapter 7 The C Application Debugger for Series 90-70 PLCs

The C Application Debugger is available with the PLC C Toolkit Professional (IC641SWP719). The PLC C Toolkit Professional includes an enhanced version of the C Programmer's Toolkit, the C Application Debugger, and the Soft-Scope® user interface (a product of Concurrent Sciences, Inc.).

The C Application Debugger for the Series 90-70 PLC offers programmers advanced debugging capabilities. This chapter contains information on installing the Debugger, starting a debugging session, and controlling and troubleshooting the debugging process.

Chapter 7 contains the following sections:

Caution

The C Application Debugger is a powerful tool that allows programmers to modify application code and data within the PLC CPU and significantly alter the behavior of the application being debugged. Care should be exercised in using the Debugger; indiscriminate use of its functionality may cause the application to malfunction.

Section 1: Installing the C Debugger

Installing the Toolkit

The first step in installing the C Debugger is to install the C Programmer's Toolkit software. See chapter 2 for a detailed description of this procedure.

Note

In addition to the directories described in chapter 2, the installation program will create the following subdirectory on the specified hard drive if it does not already exist:

```
<drive>:\s9070c\cdbs
```

This directory contains support files for the C debugger. See appendix B for a list of files copied to this subdirectory.

Editing the AUTOEXEC.BAT file

You must make two additional changes to the **autoexec.bat** file:

1. Add the **\s9070c\cdbs** directory to the **PATH** definition immediately following the **\s9070c** directory.
2. Add the command **LH <drive>:\s9070c\cdbs\snp.exe** to the end of the **autoexec.bat** file. This command installs the SNP serial driver at boot time to allow communication with the Debugger.

Note

If using a serial mouse on comm port 1 or 2 (not a dedicated mouse port), then the **snp.exe** line must appear before the line that installs the mouse driver.

Editing the CONFIG.SYS file

Add the command **DEVICEHIGH=<drive>:\s9070c\cdbs\ssrs232c.exe** to the end of the **config.sys** file. This command installs the GE Fanuc Debugger driver into high memory at boot time.

Note

The **ssrs232c.exe** driver included with the C Programmer's Toolkit is a replacement for an identically named driver included with CSI's Soft-Scope. **The C Debugger will not function properly with CSI's version of this driver.** After installing Soft-Scope (see below) you should verify that the **config.sys** file contains only the command to load the GE Fanuc driver as described above. A command installing CSI's version should **not** be added to **config.sys.**

Editing the GEF_CFG.INI file

You will need to edit the **gef_cfg.ini** file only in the following instances:

If you need to change any of the default serial port settings (port number, baud rate−19200, parity−odd, stop bits−1, modem turnaround time−0)

If you have a multidrop configuration and need to be able to communicate with different PLCs.

The **GEF_CFG.INI** file is located in \S9070C\CDBS and will have the same or very similar information as the following printout of a sample GEF_CFG.INI file:

```
; GEF_CFG.INI file for Series 9070 C debugger

; COPYRIGHT (c) 1995   GE FANUC AUTOMATION NORTH AMERICA INC.
; Published in only a limited, copyright sense and all rights, including
; trade secret rights, are reserved. Unauthorized use of the information
; or program is strictly prohibited.

; The following line specifies the name of the PLC as it will appear
; on the debugger host software's PLC select screen.

[PLC1]

; The following lines describe the serial port parameters.
; They must appear under the port name "DEFAULT" for proper operation.
; The line "TYPE = SNP_SERIAL" is required to specify SNP serial
; communications.
; The line "PORT = COM1" specifies the comm port; the allowable values
; are COM1 and COM2.

[DEFAULT]
TYPE = SNP_SERIAL
PORT = COM1
```

Installing Soft-Scope

The next step in the installation process is to install CSI's Soft-Scope debugger user interface and its associated support utilities as described in the Soft-Scope documentation. Note that this installation procedure also requires updating the **PATH** in the **autoexec.bat** file. Consult the Soft-Scope manual for more details.

Note

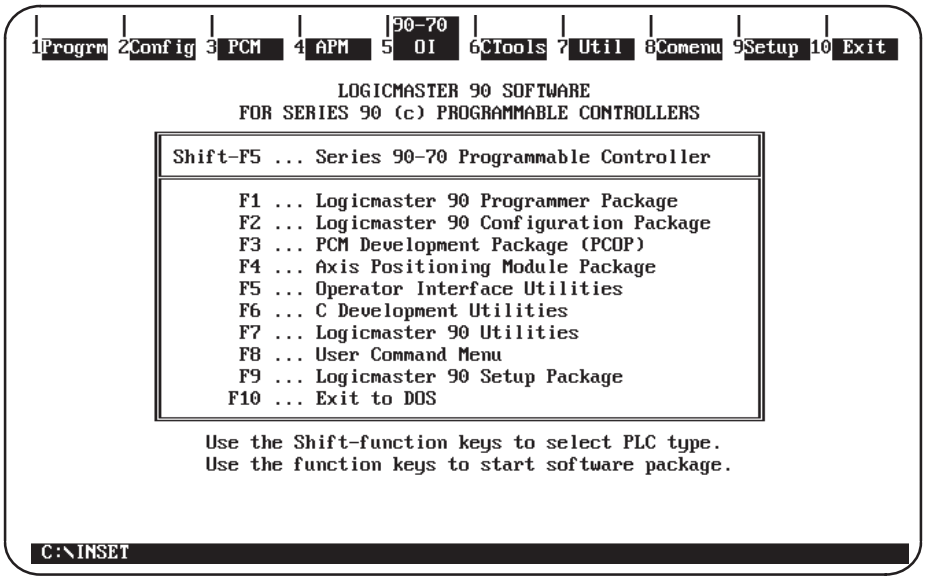
In addition to installing the Soft-Scope user interface and its associated support utilities, CSI's installation procedure will also install the CSiMon debug monitor sources and example files. These files are not necessary to run the C Debugger.

Note

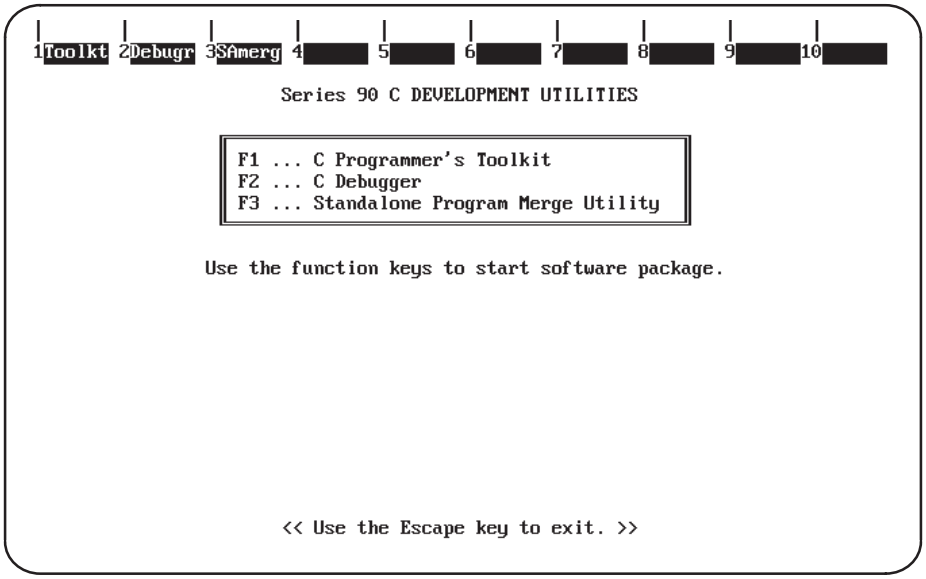
Remember to reboot your PC before attempting to use the Debugger so that the changes to the **autoexec.bat**, **config.sys**, and if applicable, **gef_cfg.ini**, files will take effect.

Section 2: Starting a Debugging Session

To access the C Debugger, choose the C Development Utilities menu (F6) on the Logicmaster Main Menu (Release 6.0 or later of Logicmaster 90 is required):



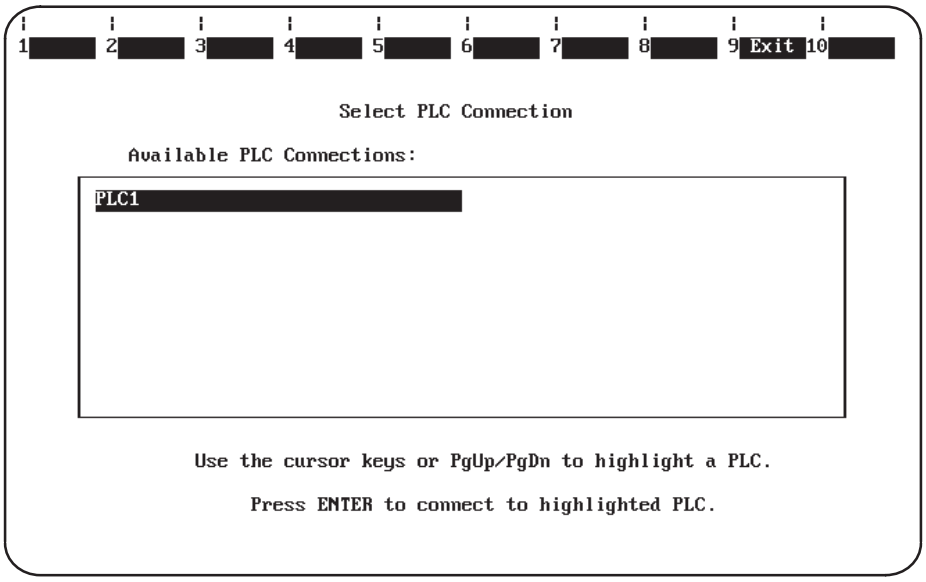
Choose C Debugger (F2) on the the Series 90 C Development Utilities menu screen:



The following table describes useful keystrokes and their functions for use within the C Debugger:

Keystroke	Function
F9	Return to the previous menu
ESC	Return to the previous menu
ALT-A	Abort current action
ALT-C	Clear the current active edit field
ALT-H	Help
ALT-K	Key help
ALT-L	List directory files

On the Series 90 C Development Utilities screen, select C Debugger (F2) to initiate the Select PLC Connection screen:



The Select PLC Connection screen provides a list of CPUs able to communicate with the Debugger.

Selecting a PLC

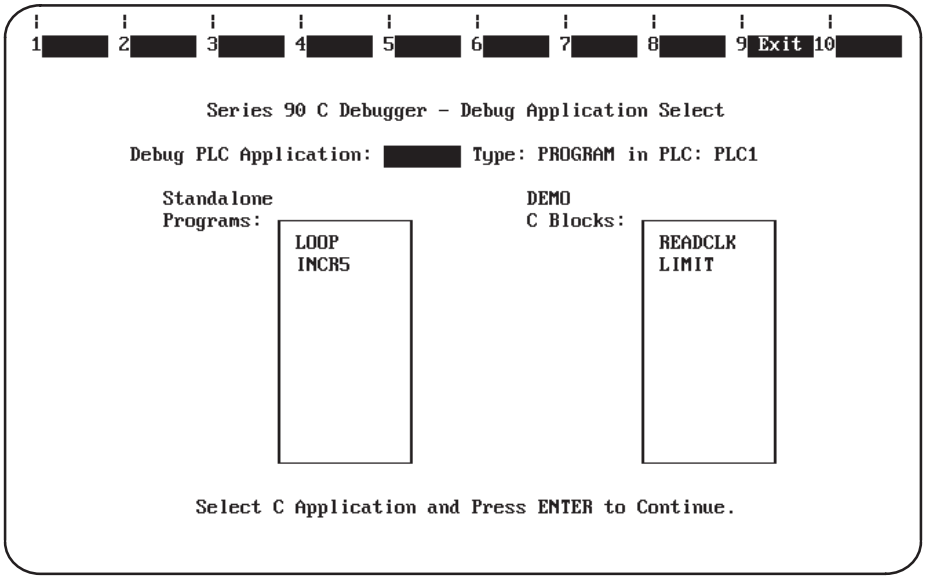
Use the PgUp/PgDn keys to view the next/previous page of available PLCs and the arrow keys to highlight the desired PLC.

Press the **ENTER** key to begin communicating with the highlighted PLC.

You will be prompted to confirm the selection with the message "Connect to PLC <*plc name*> (Y/N)?"

An "N" answer (or any non-"Y" answer) will reactivate the Select PLC Connection screen.

A "Y" answer will activate the Debug Application Select screen:



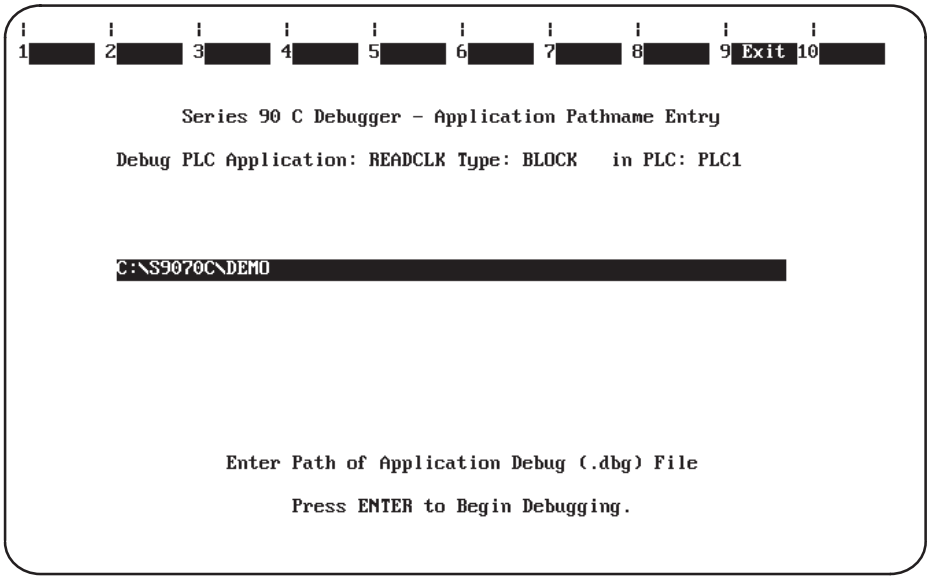
Specify the C application to be debugged using one of the following methods:

Using the arrow keys, move the cursor through the list of Standalone Programs or C Blocks to highlight the desired application. Press **ENTER** to activate the highlighted selection.

OR

Type the name of the C application into the "Debug PLC Application" field, use the **TAB**/**SHIFT-TAB** to select the application type (Program or Block) in the "Type" field. Press **ENTER** to activate the selection.

The Application Pathname Entry screen will appear:



The Application Pathname Entry screen is used to inform the Debugger software of the application's .DBG file location. This .DBG file was created by the Toolkit. Accessing the .DBG file allows the Debugger software to debug at the source code level and access all available debugging symbols.

Locating the .DBG file

The Debugger software searches the current directory for a .DBG file matching the name of the selected C application. If the file is found in the current directory, the pathname of the current directory will be displayed as default text in the pathname entry field.

If the file is not found, the Debugger software will try to determine the pathname of the last application built by the Toolkit and, if successful, will display the pathname as default text in the pathname entry field; otherwise, the root directory of the current drive will be displayed.

Note

If the application is a C Block, then it may have been renamed during export to/import from the LM90 library. In such a case, the name of the .DBG file on the disk will not agree with the name of the C block within the PLC. To debug the block you will need to specify in the application pathname entry field the name of the .DBG file together with the path to the subdirectory containing the .DBG file. Under all other circumstances, specification of the .DBG filename in the pathname entry field is optional.

Edit the default pathname as needed and press **ENTER**.

If the software cannot locate the .DBG file you specified, the message "<*appname*>.DBG not found" will appear and the pathname entry field will reactivate.

Note

If the .DBG file does not exist and the Toolkit is Version 3.0 or later, rebuild the project with **DEBUG=ON** in the **BLDVARS** file.

You must have Version 3.0 or later of the Toolkit installed to generate the .DBG file.

The software will prompt the user for final confirmation by displaying "Debug PLC Application:<*application name*> Type: <*type*> in PLC <*PLC name*> (Y/N)?"

An "N" answer (or any non-"Y" answer) will reactivate the Application Pathname Entry Field.

A "Y" answer will activate the Debugger on the selected application.

Section 3: Controlling the Debugging Process

This section outlines specific features or ways of controlling the debugging process:

- Optimizing performance of the user interface
- Controlling application execution
- Accessing CPU reference memories
- Printf() function
- Calculating background checksum
- Patching application code
- Terminating a debug session

For more detailed information about the debugging process and Concurrent Science's Soft-Scope user interface, consult the Soft-Scope manual.

Optimizing Performance of the User Interface

Soft-Scope communicates with the C Debugger through sequences of commands known as debugger primitives. A single operation performed with Soft-Scope may require that a sequence of many primitives be carried out. Since the C Debugger firmware processes each primitive at a rate of one primitive per sweep, CPU sweep time should be reduced as much as possible to achieve optimal performance of the Debugger user interface.

To reduce sweep time, try to eliminate all sources of unnecessary sweep overhead while debugging. For example, if the application consists of multiple programs within the CPU, try to run and debug each component individually whenever possible in order to reduce total logic execution time.

Note

Soft-Scope refreshes the contents of any open Execution, Register, Dump, and Watch window each time an application break occurs. Closing these windows when they are not needed will improve Soft-Scope's response time in reporting application breaks.

Controlling Application Execution

The Debugger allows you to start and stop the execution of your application with breakpoint, step, and other execution control features. These features affect whether the Debugger believes your application is running or stopped.

PLC-related conditions

In addition, certain PLC-related conditions independent of the Debugger must be satisfied in order for the application to run:

The PLC must be in RUN mode

If the application is a standalone program, its disable reference (if specified) must be clear

If the application is a C block, then any enabling logic must pass power to the associated CALL function block

The Debugger interface has no knowledge of these PLC-related conditions and may report the application status as running even if one or more of the necessary conditions is not met. However, the application will execute only when all these conditions are met AND the application is running from the Debugger's point of view.

Note

If the PLC state transitions from RUN to STOP to RUN, the current execution point of the application is reset to a startup value outside of the user code within the CPU's operating system. (See "Application Out of Context" below.) This reset takes place during the STOP to RUN transition of the PLC state. However, if such a transition of the PLC state occurs while the application is stopped from the Debugger's point of view, then the change in execution point will not be reflected in the Code window.

Note

Timed and event-triggered standalone programs may be debugged with the C Application Debugger. If the timer or I/O interrupt event associated with the program triggers while the program's execution is stopped by the Debugger, then that trigger will be ignored by the PLC (that is, the trigger event will not cause the program to rerun while an outstanding invocation of the program is under the control of the Debugger.)

Debugging timed or I/O interrupt-driven C blocks (or C blocks called directly or indirectly from other timed or interrupt blocks) is **NOT** supported. If a breakpoint is encountered within a such a C block, an informational "Breakpoint Encountered in Interrupt Block" fault will be logged in the PLC fault table. In this case, the breakpoint will be removed and the application will continue executing.

Functionality restrictions

Certain aspects of the Debugger's functionality are restricted while the application is running:

Soft-Scope's Register window is not updated during the execution of the application

The application's current execution point as indicated in the Code window is updated only when the application transitions from running to stopped from the

Debugger's point of view. Since the Code window is not updated while the application is running, the current execution point indicated in this window is valid only while the application is stopped. This information becomes "stale" when the application starts running, and it will be updated again when the application stops executing.

The Data, Watch, and Dump windows will not be updated during this period unless the configuration option **exec.refresh** has been set to a non-zero value. (See the Soft-Scope documentation for more information about **exec.refresh**.)

Breakpoints

Although a new breakpoint can be created with Soft-Scope at any time, breakpoints take effect only after the application has transitioned from the stopped state to the executing state. Thus, a breakpoint entered while the application is running will not become active until the application has transitioned from executing to stopped and back to executing again. The existence of such a breakpoint will be reflected in Soft-Scope's Code and Breakpoint windows immediately upon entry to indicate that the breakpoint was successfully created in the Debugger's internal database. Breakpoints entered while the application is stopped will become active immediately upon the application's next transition to the running state.

Accessing CPU Reference Memories

The Debugger has the ability to access the CPU's reference memories. A set of macros is provided to allow convenient access to these areas from Soft-Scope. The appropriate macro file is **`<drive>:\S9070C\CDBS\ACCESS.MAC`** where **`<drive>`** is the letter associated with the disk drive where the Toolkit has been installed. After this file has been loaded into Soft-Scope with the Macro/Load command, the macros are available for execution with the Macro/Display command.

The macros available from **`ACCESS.MAC`** are direct counterparts to the C macros available from the header file **`REFMEM.H`**. For example, invoking the ri macro with a parameter equal to 5 will display the contents of %R5 as a signed integer. As written, all the macros display their output in the Watch window.

Note

These macros rely on an internal table of data pointers to execute correctly. This table is initialized during a STOP to RUN transition by the CPU. Therefore, these macros will not return meaningful results until the first STOP to RUN transition occurs. Such a transition is necessary after every STOP mode store of logic in order to initialize the table properly.

Using the Printf() Function

Application programs may use the C language's standard I/O library function printf() to output data through the CPU's serial port with the following restrictions:

The serial port must be configured for message generation mode (MSG) instead of the default SNP mode in order to support the printf() function

Because the Debugger communicates via the serial port, sending printf() data through this port is not supported while an application is being debugged. No

output is performed by a call to printf() in this case; however, the return value of the printf() call (typically the number of characters generated) will be the same as if no debugging session were currently active.

Note

If the CPU's serial port is configured for MSG mode, it is necessary to place the PLC in STOP mode in order to establish communication between Soft-Scope and the CPU. However, once a debugging session has been established, Debugger communication will be maintained regardless of any STOP to RUN or RUN to STOP transitions by the CPU during the course of the session.

If the CPU's serial port is configured for SNP mode, then it is not necessary to place the PLC in STOP mode to start the Debugger.

Calculating Background Checksum

The CPU firmware ordinarily performs a periodic checksum calculation over all applications currently stored in memory in order to detect memory corruption. However, because the Debugger may modify the application's code image during the course of a debug session (while setting a software breakpoint, for example), this background checksum calculation will be disabled for the application being debugged. Background checksums continue as normal for all applications which are not being debugged, and checksums for an application being debugged are reenabled automatically when the debug session is terminated.

Patching Application Code

Using the Debugger's memory modification capabilities, it is possible to modify the code image, or "patch," an application being debugged. However, the expected checksum value for an application is not recomputed as a result of a patch. If the Debugger is used to patch an application, the checksum calculated over the modified code image will probably disagree with the expected value for the checksum (as calculated at program download time). However, this mismatch will not be detected until the current debug session is terminated and background checksums are reenabled for the application. When this condition is detected, a **fatal** User Patch Detected Fault will be logged in the PLC fault table.

Note

Patching application code with the Debugger is not recommended. Problems in application code should be corrected by rebuilding the application source code and downloading the modified application to the PLC.

Terminating a Debug Session

Under normal circumstances, a debug session is terminated using the Soft-Scope File/Quit menu selection. Upon exiting Soft-Scope, the PLC Select screen will be

redisplayed to allow the user to select a new PLC with which to establish communication. At this time the CPU firmware will clean up the current debug session in preparation for a new one. Several actions occur as a result of this cleanup process:

All current hardware and software breakpoints will be removed

Background checksum calculations will be reenabled for the application (see "Calculating Background Checksum and "Patching Application Code")

If application execution was halted by the Debugger, it will be restarted

Under certain special circumstances, the CPU firmware will automatically terminate the current debug session. These circumstances include:

loss of the Debugger's communication link

loss of CPU power (termination of debug session occurs during subsequent power up)

If the CPU firmware terminates a debug session, the normal cleanup actions outlined above will take place. However, Soft-Scope will be unaware that the debug session has been terminated and will continue to attempt to communicate with the CPU firmware. This will result in a communication timeout or an error in communicating with the SSDEV device driver (ssrs232c.exe).

If a communication timeout occurs, press **ENTER** to remove the dialog box then exit Soft-Scope using the File/Exit menu selection

If a read or write fault error occurs on device SSDEV, press the **A** key to abort the current operation, then exit Soft-Scope using the File/Exit menu selection.

An additional "System Timed Out Waiting for PLC to Respond" message may also appear on the screen. Press **ENTER** to acknowledge the message, then exit the Debugger software.

Note

It may take up to 65 seconds for the software to indicate a loss of communications after the physical link has been broken.

Section 4: Special Considerations

This section expands on information provided in the Soft-Scope user manual by describing special considerations to take into account involving the following:

Notes on Soft-Scope Functionality

Specifying memory addresses

Data breakpoints

System calls

Using Logicmaster during a debug session

Application out of context

Notes on Soft-Scope Functionality

This section outlines additional aspects of the Debugger's functionality which differ from the functionality described in the Soft-Scope documentation:

The Monitor Application Programming Interface functions described in the Soft-Scope documentation are not supported.

The Load and Symbol Load options within the File menu are not supported. The Load command is unnecessary since applications are downloaded to the PLC through the LogicMaster software independent of the Soft-Scope debugger. The Symbol Load command is also unnecessary since the application's symbol information is automatically downloaded whenever a debugging session initiates.

Performing port I/O operations with Soft-Scope's port() function is not supported. Any attempt to access a port with this function will generate an error message.

The TERMINAL command allows direct access to the Debugger at the monitor command level; however, using this interface is **not** recommended.

Caution

Indiscriminate use of the TERMINAL command may cause the PLC to malfunction.

Specifying Memory Addresses

Any program executing in the CPU (LD program, SFC program, or Standalone C Program) can access an address space one megabyte in length spanning the address range 00000 to FFFFF hexadecimal. These 20 bit addresses are specified in the microprocessor via 16 bit segment values in conjunction with a 16 bit offset value. The microprocessor resolves these two 16 bit values into a 20 bit address by shifting the segment value to the left by 4 bits and then adding the offset value. For example,

Segment 1234, Offset 5678 is resolved to address 12340 + 5678 = 179B8.

The Soft-Scope documentation mentions three alternative methods for directly specifying a memory address:

A "logical" address is specified in segment:offset form, for example, 1234:5678

A "linear" address is specified as a 20 bit number followed by L, for example, 179B8L

A "physical" address is specified as a 20 bit number followed by P, for example, 179B8P

From the Debugger's point of view, these three methods are equivalent. All three of the examples above specify the same address. The CPU's operating system and the microprocessor are responsible for managing the individual address spaces seen by each program and mapping them onto the "real" physical addresses used by the memory address decoding hardware. The C Debugger is unaware that this mapping takes place and therefore treats linear and physical addresses as equivalent. Addresses entered with either the L or P suffix will be displayed in Soft-Scope with the P suffix. Addresses entered in segment:offset form will be displayed in that form.

Note

Memory addresses corresponding to variables in a C program may be specified with a symbol name. Consult the Soft-Scope documentation for more information concerning the syntax of symbol names.

Data Breakpoints

The Debugger supports the use of the microprocessor's debug registers to monitor accesses to program variables. As explained in the Soft-Scope manual, two types of data breakpoints are available: access breakpoints and write breakpoints. An access breakpoint will trigger whenever the microprocessor reads or writes at the specified memory address, while a write breakpoint will trigger only when writes occur. Note that a single line of C code may read and/or write a particular variable multiple times, so a data breakpoint may trigger more than once on a single line of C code. When a data breakpoint triggers, execution stops on the assembly language instruction immediately **following** the instruction which performed the access.

Note

Care should be taken when using data breakpoints with stack-based variables (that is, variables which are local to a single routine or passed as parameters). Due to a limitation with Soft-Scope, data breakpoints on these variables may trigger unexpectedly. Because these variables are stack-based, their locations on the stack may be reused for other purposes as the application executes. A data breakpoint associated with a stack-based variable will trigger on any access of the associated memory address, even if the access is a reuse of memory not related to the variable. Such "false triggers" can be distinguished by noting the routine in which the current execution point is located at the time of the trigger. If this routine is not the same as the routine associated with the stack based variable, then the trigger is invalid.

System Calls

A system call is a special type of subroutine call used by a C application to request a service from the CPU's operating system. The support functions defined in the plcfnc.h

header file (PLCC_read_elapsed_clock(), PLCC_get_plc_version(), etc.) are system calls. They are implemented via assembly language software interrupt instructions (for example, INT 61h and INT 21h).

The Debugger treats certain execution control scenarios involving system calls as special cases. When stepping at the assembly language level, the Debugger will not step into the code invoked by a system call. Instead, execution will occur at full speed through the system call, and the Debugger will stop application program execution at the return point of the INT instruction. This holds true regardless of the type of step performed (step INTO or step OVER the called procedure). Similarly, a memory access which would ordinarily trigger a data breakpoint will not do so if the access occurs within a system call. Instead, the Debugger postpones reporting the breakpoint trigger until the system call is complete. Once again, the Debugger will stop executing the application at the return point of the INT instruction associated with the system call.

Note

Invocation of system calls is implemented by library routines which are linked with the application. GE Fanuc does not provide the source code for these routines or for internal initialization and cleanup routines which are also linked with the application. If application execution stops within any of these areas (as a result of a data breakpoint triggering within a system call, for example), Soft-Scope will be unable to locate the corresponding source file and will instead display a dialog box prompting the user to input its location. Press **ENTER** to allow Soft-Scope to continue without source code; the execution point will then be displayed in assembly language.

Using Logicmaster 90 During a Debug Session

The following operations are unavailable in Logicmaster 90 while a debug session is active:

clearing logic from the LM90 Clear Memory screen

implicitly clearing memory as a result of storing of logic while the PLC is in STOP mode

any store of logic in RUN mode, including updates to LD/SFC via ALT-S

Attempting to perform any of these operations will cause LM90 to display an "Insufficient Privilege" message. If any of the above operations are currently in progress while an attempt is made to activate a debug session, then the activation of the debug session will be refused.

Modifying PLC reference memories on any of LM90's reference memory screens or on the logic execution screen is permitted.

Note

The scenarios described in this section require communication with LM90 during an active debug session. This is only possible if LM90 is communicating through some means other than the CPU's serial port.

Application Out of Context

The Soft-Scope user interface automatically displays the code at the current execution point whenever the application is stopped from the Debugger's point of view. However, if the current execution point is within the CPU's operating system (rather than within the user application code), then it cannot be displayed meaningfully to the user. When the application is in this state, it is said to be "out of context." This state occurs most commonly when the application is stopped between invocations, that is, after the code in the C block or program has completed execution for a particular sweep but before main() has been called for its next execution.

Note

The Debugger automatically stops application execution at the start of a debug session. Under most circumstances this stoppage will occur between invocations of the application. Thus, it is normal for the application to be out of context immediately after starting a debug session. The application will also typically be out of context whenever it is stopped manually with Soft-Scope's Code/Stop command.

When the application is out of context, Soft-Scope's Code window will appear as shown below:

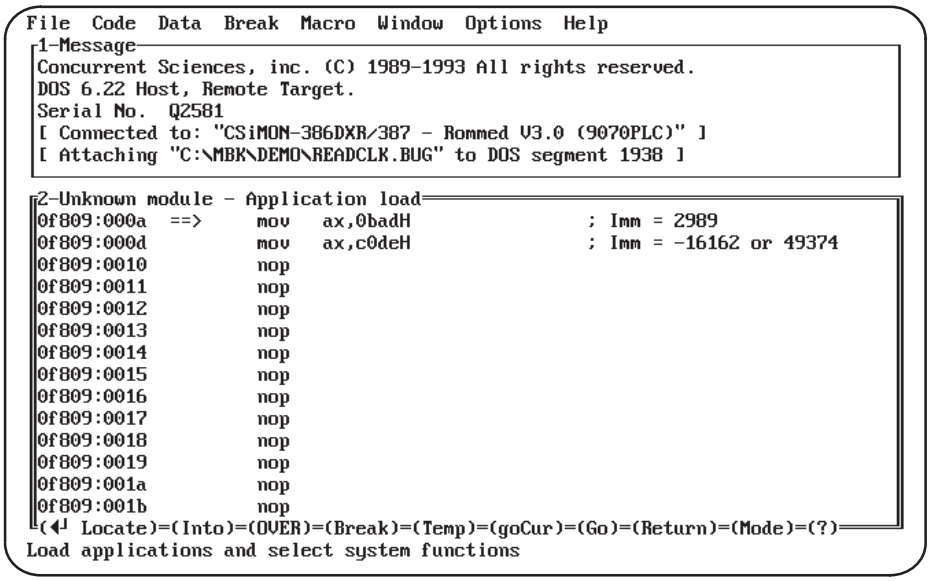


Figure 7-1. Application Out of Context

The displayed execution point is within a representative "dummy" code stream which indicates that the application is out of context (this is not the actual execution point, which is within the CPU's operating system). While the application is out of context, certain capabilities normally available with the Debugger are restricted. For example, all register values displayed in the Register window will be 0xFFFF or 0xFFFFFFFF, as appropriate, except for the CS and EIP registers, which will point to the dummy code stream. Attempts to modify any of the registers will be disallowed. However, memory addresses may be read or written as normal. Attempts to step execution will have no effect while the application is out of context. However, the GO command (in all of its variations) and the debugger's breakpointing functions will be available in this state.

The out of context state may be exited by issuing any form of the Soft-Scope GO command to restart application execution. The next time the application's execution is stopped, the execution point at the time of the stoppage will be reevaluated. If it is within the bounds of application code, it will be displayed as normal. Otherwise, it is within the CPU's operating system, and the application will again be out of context.

Note

A simple way to ensure that execution stops within the bounds of application code is to use the Code/Goto command to break at the beginning of main() or any other convenient point. Alternatively, any of the Debugger's breakpointing functions can be used in conjunction with the GO command. (Note that Code/Display command can be helpful in selecting breakpoints and is available even when the application is out of context.)

Section 5: Troubleshooting

This section describes page faults and error conditions that might be reported by the host Debugger software.

Page Faults

As explained earlier, the CPU's operating system and the microprocessor are responsible for mapping the addresses seen by an application program onto the physical addresses used by the memory hardware. The mechanism used to accomplish this mapping is known as paging. It is important to understand that not every byte in the program's one megabyte address space is necessarily mapped onto a physical memory address. Many addresses may be "unmapped" and therefore effectively inaccessible to the program. Any attempt to access one of these unmapped addresses will cause the microprocessor to signal a special error condition known as a page fault. The most common cause of page faults in a C application program is an attempt to dereference a NULL or uninitialized pointer.

The CPU's method of handling application page faults varies. If the application is not currently being debugged, a fatal exe block runtime error or standalone runtime error fault will be logged as appropriate in the PLC fault table. If the application is executing in a debug session when the page fault occurs, the Debugger will stop application execution at the assembly instruction which caused the page fault and will display "INT xx − Page Fault" in the Code window. Note that the execution point will occur **before** the instruction which attempted the illegal access. Thus, it may be possible to take corrective action (such as modifying an appropriate register or memory value) to allow program execution to continue with the GO command. If no such corrective action is taken, then restarting program execution will cause another page fault to occur. When a page fault occurs in an application being debugged, no runtime error fault is logged.

If a page fault occurs due to an illegal memory access by the Debugger itself (as opposed to an illegal access by the application program being debugged) then the fault will be signalled by Soft-Scope. The wording of the error depends on the particular operation attempted. For example, the NULL pointer is represented by memory address 0000:0000. Attempting to perform a memory dump at this address with the Data/Dump command will result in the error message "Bad Memory Read."

Error Conditions

Under some circumstances, an error condition may be reported by the host Debugger software. Error messages reported by Soft-Scope are detailed in the Soft-Scope documentation. The most common error conditions reported by the Debugger software are described below:

ERROR MESSAGE	EXPLANATION	CORRECTIVE ACTION
Debugging not supported in selected PLC	The PLC selected for debugging contains pre-release 6 firmware, which does not support the C Debugger.	The CPU must be upgraded. Call your distributor for an upgrade kit. Note: Not all CPUs can be upgraded to Release 6.
Application Checksum Mismatch	The .DBG file containing the application's symbol information does not match the application code downloaded to the PLC.	Choose the correct path to the appropriate .DBG file.
Insufficient Privilege Level	The Debugger software cannot access the selected PLC to initiate a debugging session. The most common cause of this error is the presence of password or OEM key protection within the PLC.	Remove the PLC's password or OEM key protection.
PLC Memory Allocation Error	An error occurred when attempting to dynamically allocate memory within the PLC.	Call PLC Technical Support
Host PC Memory Allocation Error	An error occurred when attempting to dynamically allocate memory within the Debugger host PC.	Remove unnecessary TSRs or device drivers currently installed on the host PC then reboot the system. If the error continues, call PLC Technical Support.
Could not spawn ss.exe *OR* **Could not spawn ssbug.exe**	The host Debugger software could not spawn the Soft-Scope user interface or the ssbug symbol extraction utility.	Make sure that these executable files are present on the host PC and that the subdirectory where they reside is included in the DOS PATH environment variable. These errors can also indicate a lack of sufficient free memory on the host PC.
System Timed Out Waiting for PLC to Respond *OR* **Host Not Connected or Sending**	A timeout or other communications error occurred while the host Debugger software was attempting to communicate with the selected PLC.	Check the serial cable connection. This error may occur if the CPU firmware detected an error condition and terminated the debug session. See "Terminating a Debug Session."
Unable to Connect to PLC *OR* **Could not attach to specified PLC**	An error occurred while the host Debugger software was attempting to establish communication with the selected PLC.	Check the serial cable connection. Also check the autoexec.bat and config.sys files for the appropriate initialization of the snpdrv.exe and ssrs232c.exe communication drivers and the C70_PATH environment variable. Check the location and contents of the GEF_CFG.INI file. For more information, see "Section 1: Installing the Debugger."
Unable to find system files in <drive:\pathname>	The host Debugger software could not locate one of its internal system files in the indicated subdirectory.	Check the autoexec.bat file for the appropriate definition of the C70_PATH environment variable.
Internal System Error <error_number>	An internal error has occurred in the host Debugger software.	Call PLC Technical Support.

Section 6: A Sample Debug Session

This section walks you through a simple debug session using the limit C block. This example assumes the following:

that limit.c has been built as a C block with debug information and downloaded into the PLC

that the PLC is in RUN mode and that all enabling logic to the call to this block is active

1. To activate a debug session, press **F6** (C Development Utilities) on the Release 6 Logicmaster 90 Main Menu screen, and press **F2** (C Debugger) on the Series 90 Development Utilities screen.
2. On the Select PLC Connection screen, select the PLC you wish to debug.
3. On the Debug Application Select screen, select the C Block LIMIT.
4. After the Debugger has finished its initialization, you should see the initial Soft-Scope screen. Note that the current execution point is at the dummy code stream indicating that the application is out of context, that is, suspended in between invocations.
5. This example will use the PLC reference memory macros to set up the parameters to the limit block from the Debugger. (This example assumes that the block's three input parameters—the input value, the high limit, and the low limit—are tied to %R700, %R701, and %R702 respectively.) Press **ALT-M, L** to load the macro file **<drive>:\s9070c\cdbs\access.mac**, where **<drive>** is the drive where the C Toolkit was installed.
6. After Soft-Scope finishes processing the file, select the ri macro to access %R memory in integer format. Type 700 as the macro parameter and press **ENTER**.
7. Note that a new entry is placed in the watch window corresponding to the current value of %R700. Place the cursor on this entry and press **ENTER** to modify the current value of %R700; set it equal to 2 and press **ENTER**.
8. In a similar manner, set %R701 equal to 3 and %R702 equal to 1.
9. Execute the application until the routine main() is reached. Press **ALT-C** then type **G** then type **MAIN** to command the Debugger to execute to main().
10. The application stops at this routine, and Soft-Scope displays the current execution point.
11. Note that the input value (2) falls between the high limit (3) and the low limit (1). Therefore, main() should set the output value of the C block equal to the input value on line 79. Verify this value with a write data breakpoint on the output value. Press **ALT-B, W** and type ***output_value** to set up the data breakpoint. Don't forget the leading *****, since **output_value** is a pointer to an integer, and you want to break when the value pointed to (i.e., ***output_value**) is written.
12. Return to the execution window by pressing **ALT-1**, and type **G** to restart application execution. Note that application execution stops on the instruction

immediately following the write to **`*output_value`**, in this case on line 80 as expected.

13. The next operation main() will perform is updating the alarm bits. Let's step through the C code to observe this process:

    First, add the expression **`*alarm_bits`** to the Watch window by pressing **ALT-D, W** and typing **`*alarm_bits`**.

    If the lower two bits of this value are not currently set, set them now by changing the value of this variable to 3.

    Press **ALT-2** to return to the Execution window.

    Press **O** to step over the next line of C code. Note that the value of **`*alarm_bits`** is now 2.

    Press **O** to step again and note that the value is now O.

14. This concludes the sample debug session. Press **Alt-F9** to exit

Appendix A

Standard C Library Functions Supported in the Series 90 PLC

The following Microsoft C runtime library functions are supported in the Series PLC. These functions are not valid for C FBKs

0 p2 3%PC1Q4 C	%PC1Q4 C
Input/Output	printf (Series 90-70 only) sprintf (Series 90-70 only)
Internationalization	strcoll strftime
Math	acos, acosl asin, asinl atan, atanl _cabs†, _cabsl† ceil, ceill cos, cosl cosh, coshl div exp, expl fabs, fabsl floor, floorl fmod, fmodl frexp, frexpl _hypot†, _hypotl † ldexp, ldexpl ldiv log, logl log10, log10l _lrotl, _lrotr max, min modf, modfl pow, powl rand _rotl, _rotr sin, sinl sinh, sinhl sqrt, sqrtl srand tan, tanl tanh, tanhl
Search and Sort	bsearch _lfind† _lsearch† qsort

† These functions are available in the Microsoft C Version 6.0 compiler without the preceding underscore.

0 p2 3%PC1Q4 C	%PC1Q4 C
String Manipulation	strcat, _fstrcat strchr, _fstrchr strcmp, _fstrcmp strcpy, _fstrcpy strcspn, _fstrcspn strerror, _strerror _stricmp†, _fstricmp strlen, _fstrlen _strlwr†, _fstrlwr strncat, _fstrncat strncmp, _fstrncmp strncpy, _fstrncpy _strnicmp†, _fstrnicmp _strnset†, _fstrnset strpbrk, _fstrpbrk strrchr, _fstrrchr _strrev†, _fstrrev _strset†, _fstrset strspn, _fstrspn strstr, _fstrstr strtok, _fstrtok _strupr†, _fstrupr
Time	asctime difftime _strdate strftime _strtime
Buffer Manipulation	_memccpy†, _fmemccpy memchr, _fmemchr memcmp, _fmemcmp memcpy, _fmemcpy _memicmp†, _fmemicmp memmove, _fmemmove memset, _fmemset _swab†
Character Classification and Conversion	isalnum isalpha isascii iscntrl isdigit isgraph islower isprint ispunct isspace isupper isxdigit toascii tolower _tolower toupper _toupper
Data Conversion	abs atof atoi atol _itoa† labs _ltoa† strtol strtoul _ultoa†

† These functions are available in the Microsoft C Version 6.0 compiler without the preceding underscore.

Appendix B C Programming Toolkit Files

Series 90-70 Programming Toolkit Files

When the C development software installation is complete, the following list of files and the corresponding directories should appear on the target hard disk.

0 3O1%POO	2341 C 1
\s9070c	AAREADME BLDVARS CCOPY.BAT PLCC9070.H PLCFNC.H PLCFLT.H PLCVME.H REFMEM.H STAND.H PLCC9070.MKD PLC.MAK PLC7.MAK DOS.MAK DOS7.MAK TESTHARN.C STUB .C MKPLC.BAT MKDOS .BAT MKPLC7.BAT MKDOS7.BAT MKEXE.BAT PLCINC.MKD DOSINC.MKD THARNDAT.INC RMFIXUP.OBJ MSCVER.BAT SAPDOS.LIB BLKDOS.LIB FBKDOS.LIB DOS2.LIB SAPPLC.LIB BLKPLC.LIB FBKPLC.LIB PLC2.LIB

0 3O1%POO	2341 C 1
\s9070c (cont'd)	_SAPDOS.LIB _BLKDOS.LIB _FBKDOS.LIB _DOS2.LIB _SAPPLC.LIB _BLKPLC.LIB _FBKPLC.LIB _PLC2.LIB GEFLIB.FAR GEFLIB7.FAR EXISTDIR.EXE POSTPROC.EXE AUTOEXEC.BAT CONFIG.SYS LLIBCA.FAR LLIBC7.FAR
\s9070c\multisrc	MAIN.C MULTISRC.DOC BLKHARN.C BLDVARS SRC1.C SRC2.C SRC3.C SRC4.C SRC5.C SRCS.H SOURCES
\s9070c\example1	LIMIT.C BLKHARN.C BLDVARS
\s9070c\example2	LIMIT.C BLKHARN.C LIMIT.DAT BLDVARS
\s9070c\exfbk	FBKHARN.C FACT.C BLDVARS
\s9070c\exsap	SAPHARN.C BUBBLE.C BLDVARS

0 3O1%POO	2341 C 1
\s9070c\cdbs Note: Additional files for C Toolkit Professional	PLCDEBUG.EXE SCREEN0.SCR SCREEN1.SCR SCREEN2.SCR SCREEN3.SCR SCREEN4.SCR SCREEN5.SCR SCREEN6.SCR SCREEN7.SCR SCREEN8.SCR SCREEN9.SCR GEF_CFG.INI MULTIDRP.INI FILE000.MSG ACCESS.MAC SSRS232C.EXE SNP.EXE

Series 90-30 Programming Toolkit Files

When the C development software installation is complete, the following list of files and the corresponding directories should appear on the target hard disk.

0 3O1%POO	2341 C 1
\s9030c	AAREADME BLDVARS CCOPY.BAT PLCC9030.H PLCFNC.H PLCFLT.H REFMEM.H PLCC9030.MKD PLC.MAK PLC7.MAK DOS.MAK DOS7.MAK TESTHARN.C STUB .C MK3PLC.BAT MK3DOS .BAT MK3PLC7.BAT MK3DOS7.BAT MKEXE.BAT PLCINC.MKD DOSINC.MKD THARNDAT.INC RMFIXUP.OBJ MSCVER.BAT

0 3O1%POO	2341 C 1
\s9030c (cont'd)	_BLKDOS.LIB _DOS2.LIB _BLKPLC.LIB _PLC2.LIB GEFLIB.FAR GEFLIB7.FAR EXISTDIR.EXE POSTPROC.EXE AUTOEXEC.BAT CONFIG.SYS LLIBCA.FAR LLIBC7.FAR
\s9030c\multisrc	MAIN.C MULTISRC.DOC BLKHARN.C BLDVARS SRC1.C SRC2.C SRC3.C SRC4.C SRC5.C SRCS.H SOURCES
\s9030c\example1	LIMIT.C BLKHARN.C BLDVARS
\s9030c\example2	LIMIT.C BLKHARN.C LIMIT.DAT BLDVARS

Appendix C

C Macros for PLC Access

C Macros for the Series 90-70 PLC

The following C macros are provided as part of the C Programmer's Toolkit to ease accessing PLC reference memory from inside the C block.

```
/* Macros for accessing %I, %Q, %M, %T, %G, %GA-%GE, %SA-%SC,   */
/* %R, %AI, %AQ, %P, and %L as bits.  There are separate macros */
/* for setting, clearing, and testing each memory type.         */
BIT_TST_I(x)          BIT_SET_I(x)          BIT_CLR_I(x)
BIT_TST_Q(x)          BIT_SET_Q(x)          BIT_CLR_Q(x)
BIT_TST_M(x)          BIT_SET_M(x)          BIT_CLR_M(x)
BIT_TST_T(x)          BIT_SET_T(x)          BIT_CLR_T(x)
BIT_TST_G(x)          BIT_SET_G(x)          BIT_CLR_G(x)
BIT_TST_GA(x)         BIT_SET_GA(x)         BIT_CLR_GA(x)
BIT_TST_GB(x)         BIT_SET_GB(x)         BIT_CLR_GB(x)
BIT_TST_GC(x)         BIT_SET_GC(x)         BIT_CLR_GC(x)
BIT_TST_GD(x)         BIT_SET_GD(x)         BIT_CLR_GD(x)
BIT_TST_GE(x)         BIT_SET_GE(x)         BIT_CLR_GE(x)
BIT_TST_SA(x)         BIT_SET_SA(x)         BIT_CLR_SA(x)
BIT_TST_SB(x)         BIT_SET_SB(x)         BIT_CLR_SB(x)
BIT_TST_SC(x)         BIT_SET_SC(x)         BIT_CLR_SC(x)
BIT_TST_R(x,y)        BIT_SET_R(x,y)        BIT_CLR_R(x,y
BIT_TST_AI(x,y)       BIT_SET_AI(x,y)       BIT_CLR_AI(x,y)
BIT_TST_AQ(x,y)       BIT_SET_AQ(x,y)       BIT_CLR_AQ(x,y)
BIT_TST_P(x,y)        BIT_SET_P(x,y)        BIT_CLR_P(x,y)
BIT_TST_L(x,y)        BIT_SET_L(x,y)        BIT_CLR_L(x,y)


/* Macros for accessing the special %S bits */
/* These macros are READ-ONLY!              */
BIT_TST_S(x)          FST_SCN               T_10MS
T_100MS               T_SEC                 T_MIN
ALW_ON                ALW_OFF               SY_FULL
IO_FULL               FST_EXE


/* Macros for accessing %I, %Q, %M, %T, %G, %GA - %GE, %S, */
/* %SA - %SC, %R, %AI, %AQ, %P, and %L as bytes.           */
IB(x)                 QB(x)                 MB(x)
TB(x)                 GB(x)                 GAB(x)
GBB(x)                GCB(x)                GDB(x)
GEB(x)                SB(x)                 SAB(x)
SBB(x)                SCB(x)                RB(x,y)
AIB(x,y)              AQB(x,y)              PB(x,y)
LB(x,y)
```

```
/* Macros for accessing %I, %Q, %M, %T, %G,    */
/*      %GA - %GE, %S, and %SA - %SC as words. */
/* NOTE:                                       */
/*   THESE MACROS ARE CONDITIONALLY AVAILABLE  */
/*   IF THE TARGET PLC CPU IS KNOWN TO SUPPORT */
/*   WORD ACCESSES INTO BIT-ORIENTED MEMORY.   */
/*   SEE “RESTRICTIONS ON MACRO USE”.          */
IW(x)               QW(x)               MW(x)
TW(x)               GW(x)               GAW(x)
GBW(x)              GCW(x)              GDW(x)
GEW(x)              SW(x)               SAW(x)
SBW(x)              SCW(x)


/* Macros for accessing %R, %AI, %AQ, %P, and %L as words. */
RW(x)               AIW(x)              AQW(x)
PW(x)               LW(x)


/* Macros for accessing %I, %Q, %M, %T, %G,      */
/*      %GA - %GE, %S, and %SA - %SC as integers.*/
/* NOTE:                                        */
/*   THESE MACROS ARE CONDITIONALLY AVAILABLE   */
/*   IF THE TARGET PLC CPU IS KNOWN TO SUPPORT  */
/*   INTEGER ACCESSES INTO BIT-ORIENTED MEMORY. */
/*   SEE “RESTRICTIONS ON MACRO USE”.           */
II(x)               QI(x)               MI(x)
TI(x)               GI(x)               GAI(x)
GBI(x)              GCI(x)              GDI(x)
GEI(x)              SI(x)               SAI(x)
SBI(x)              SCI(x)


/* Macros for accessing %R, %AI, %AQ, %P, and %L as integers. */
RI(x)               AII(x)              AQI(x)
PI(x)               LI(x)


/* Macros for accessing %I, %Q, %M, %T, %G, %GA - */
/*      %GE, %S, and %SA - %SC as doublewords.    */
/* NOTE:                                          */
/*   THESE MACROS ARE CONDITIONALLY AVAILABLE     */
/*   IF THE TARGET PLC CPU IS KNOWN TO SUPPORT    */
/*   INTEGER ACCESSES INTO BIT-ORIENTED MEMORY.   */
/*   SEE “RESTRICTIONS ON MACRO USE”.             */
ID(x)               QD(x)               MD(x)
TD(x)               GD(x)               GAD(x)
GBD(x)              GCD(x)              GDD(x)
GED(x)              SD(x)               SAD(x)
SBD(x)              SCD(x)


/* Macros for accessing %R, %AI, %AQ, %P, and %L as doublewords.*/
RD(x)               AID(x)              AQD(x)
PD(x)               LD(x)


/* Macros for accessing %R, %AI, %AQ, %P,            */
/* and %L as floating point values.      */
RF(x                AIF(x)              AQF(x)
PF(x)               LF(x)
```

```
/* Macros for accessing the %I, %Q, %M, %T, %G, %GA - */
/* %GE, %S, and %SA - %SC transition bits.            */
BIT_TST_I_TRANS(x)     BIT_TST_Q_TRANS(x)
BIT_TST_M_TRANS(x)     BIT_TST_T_TRANS(x)
BIT_TST_G_TRANS(x)     BIT_TST_GA_TRANS(x)
BIT_TST_GB_TRANS(x)    BIT_TST_GC_TRANS(x)
BIT_TST_GD_TRANS(x)    BIT_TST_GE_TRANS(x)
BIT_TST_S_TRANS(x)     BIT_TST_SA_TRANS(x)
BIT_TST_SB_TRANS(x)    BIT_TST_SC_TRANS(x)


/* Macros for accessing the %I, %Q, %M, %T, %G, %GA - %GE,*/
/* %S, and %SA - %SC transition bits as bytes.            */
IB_TRANS(x)            QB_TRANS(x)        MB_TRANS(x)
TB_TRANS(x)            GB_TRANS(x)        GAB_TRANS(x)
GBB_TRANS(x)           GCB_TRANS(x)       GDB_TRANS(x)
GEB_TRANS(x)           SB_TRANS(x)        SAB_TRANS(x)
SBB_TRANS(x)           SCB_TRANS(x)


/* Macros for accessing diagnostic memory */
BIT_TST_I_DIAG(x)      BIT_TST_Q_DIAG(x)  IB_DIAG(x)
QB_DIAG(x)             AIB_DIAG(x)        AQB_DIAG(x)
AI_HIALRM(x)           AI_LOALRM(x)       AIB_FAULT(x)
AQB_FAULT(x)           AI_OVERRANGE(X)    AI_UNDERRANGE(X)


/* Macros for accessing RACK/SLOT/BLOCK fault information */
RACKX(r)               SLOTX(r,s)         BLOCKX(r,s,b,sba)
RSMB(x)                FIPX(r,s,sba)


/* Use the following to access the size of each PLC memory table */
L_SIZE                 P_SIZE             R_SIZE             AI_SIZE
AQ_SIZE                I_SIZE             Q_SIZE             T_SIZE
M_SIZE                 G_SIZE             GA_SIZE            GB_SIZE
GC_SIZE                GD_SIZE            GE_SIZE            SA_SIZE
SB_SIZE                SC_SIZE            S_SIZE             I_DIAGS_SIZE
Q_DIAGS_SIZE           AI_DIAGS_SIZE      AQ_DIAGS_SIZE
```

C Macros for the Series 90-30 PLC

The following C macros are provided as part of the C Programmer's Toolkit to ease accessing 90-30 PLC reference memory from inside the C block.

```
/* Macros for accessing %I, %Q, %M, %T, %G, %SA-%SC,            */
/* %R, %AI, and %AQ as bits.  There are separate macros */
/* for setting, clearing, and testing each memory type.         */
BIT_TST_I(x)          BIT_SET_I(x)          BIT_CLR_I(x)
BIT_TST_Q(x)          BIT_SET_Q(x)          BIT_CLR_Q(x)
BIT_TST_M(x)          BIT_SET_M(x)          BIT_CLR_M(x)
BIT_TST_T(x)          BIT_SET_T(x)          BIT_CLR_T(x)
BIT_TST_G(x)          BIT_SET_G(x)          BIT_CLR_G(x)
BIT_TST_SA(x)         BIT_SET_SA(x)         BIT_CLR_SA(x)
BIT_TST_SB(x)         BIT_SET_SB(x)         BIT_CLR_SB(x)
BIT_TST_SC(x)         BIT_SET_SC(x)         BIT_CLR_SC(x)
BIT_TST_R(x,y)        BIT_SET_R(x,y)        BIT_CLR_R(x,y
BIT_TST_AI(x,y)       BIT_SET_AI(x,y)       BIT_CLR_AI(x,y)
BIT_TST_AQ(x,y)       BIT_SET_AQ(x,y)       BIT_CLR_AQ(x,y)


/* Macros for accessing the special %S bits */
/* These macros are READ-ONLY!              */
BIT_TST_S(x)          FST_SCN               T_10MS
T_100MS               T_SEC                 T_MIN
ALW_ON                ALW_OFF               SY_FULL
IO_FULL               LST_SCN


/* Macros for accessing %I, %Q, %M, %T, %G, %S, %SA - %SC, */
/* %R, %AI, and %AQ as bytes.                               */
IB(x)                 QB(x)                 MB(x)
TB(x)                 GB(x)                 SB(x)
SAB(x)                SBB(x)                SCB(x)
RB(x,y)               AIB(x,y)              AQB(x,y)
```

```
/* Macros for accessing %I, %Q, %M, %T, %G,    */
/*    %S, and %SA - %SC as words.              */
IW(x)               QW(x)               MW(x)
TW(x)               GW(x)               SW(x)
SAW(x)              SBW(x)              SCW(x)


/* Macros for accessing %R, %AI, and %AQ as words. */
RW(x)               AIW(x)              AQW(x)


/* Macros for accessing %I, %Q, %M, %T, %G,      */
/*       %S, and %SA - %SC as integers.          */
II(x)               QI(x)               MI(x)
TI(x)               GI(x)               SI(x)
SAI(x)              SBI(x)              SCI(x)


/* Macros for accessing %R, %AI, and %AQ, as integers. */
RI(x)               AII(x)              AQI(x)


/* Macros for accessing %I, %Q, %M, %T, %G, %S,   */
/*       and %SA - %SC as doublewords.            */
D(x)                QD(x)               MD(x)
TD(x)               GD(x)               SD(x)
SAD(x)              SBD(x)              SCD(x)


/* Macros for accessing %R, %AI, and %AQ, as doublewords. */
RD(x)               AID(x)              AQD(x)


/* Macros for accessing %R, %AI, %AQ,             */
/*         as floating point values.              */
RF(x)               AIF(x)              AQF(x)
```

```
/* Macros for accessing the %I, %Q, %M, %T, %G,       */
/*      %S, and %SA - %SC transition bits.            */
BIT_TST_I_TRANS(x)     BIT_TST_Q_TRANS(x)
BIT_TST_M_TRANS(x)     BIT_TST_T_TRANS(x)
BIT_TST_G_TRANS(x)     BIT_TST_S_TRANS(x)
BIT_TST_SA_TRANS(x)    BIT_TST_SB_TRANS(x)
BIT_TST_SC_TRANS(x)


/* Macros for accessing the %I, %Q, %M, %T, %G,          */
/* %S, and %SA - %SC transition bits as bytes.           */
IB_TRANS(x)            QB_TRANS(x)        MB_TRANS(x)
TB_TRANS(x)            GB_TRANS(x)        SB_TRANS(x)
SAB_TRANS(x)           SBB_TRANS(x)       SCB_TRANS(x)


/* Use the following to access the size of each PLC memory table */
R_SIZE                 AI_SIZE            AQ_SIZE            I_SIZE
Q_SIZE                 T_SIZE             M_SIZE             G_SIZE
SA_SIZE                SB_SIZE            SC_SIZE            S_SIZE
```

Appendix D Calculating PLC Memory Usage for a C Block

Series 90-70 Memory Usage Calculation

For any C block, the impact on PLC memory usage may be determined by the following equation:

```
plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD
```

where:

- **GBLDATSIZ** is determined from the **.MAP** file created during the building of the of the PLC-executable version of the C program. The **.MAP** file can be found in the same directory as the built PLC-executable version of the C program. The amount of data space treated as global data is:

  GBLDATSIZ = (the start address of the first segment in the class BSS) − (the start address of the **MEM_DATA** segment).

- **MINALLOC** is determined by executing **EXEHDR.EXE** (provided with Microsoft C compiler) on the PLC-executable **.EXE** file. **EXEHDR** will display information about the **.EXE** file, the value displayed for ***Extra paragraphs needed*** is the value to use for **MINALLOC**.

- **EXEIMAGSIZ** is the size of the PLC-executable .EXE file as determined by the MS-DOS directory command.

- **OVRHEAD** is the value 530 bytes.†

Note

GBLDATSIZ and EXEIMAGSIZ should be rounded up to the nearest 16-bit boundary.

The examples provided in this appendix are based upon a version of the toolkit which was registered to GE Fanuc Automation N.A., Inc., and uses a Microsoft Version 8.0 compiler. You can expect to get slightly different file sizes if you attempt these same examples using the C Programmer's Toolkit which is registered to you and your company, or if you use a different version of the compiler. The examples are provided to illustrate the process of computing the amount of PLC memory required for a specific C block.

† OVRHEAD is 530 bytes for Release 4.02 and Release 5.0 of the Series 90-70 CPU firmware. This value may increase or decrease in future CPU firmware releases.

Smallest Possible Impact on PLC Memory

The C block with the smallest impact on PLC memory would be built from the following source (**NULL.C**):

```
#include "plcc9070.h"

main() {
    return(OK);
}
```

When **NULL.C** is built for PLC execution using **MKPLC.BAT**, the resulting **.EXE** file is 1,559 bytes in size (as displayed by the MS-DOS directory command).

```
 Volume in drive C is 9070_CBLKS
 Directory of  C:\APPS\NULL\PLC

NULL             EXE    1559   2-07-94  11:39a
       1 File(s)   2074624 bytes free
```

The **.MAP** file for **NULL.EXE** contains the following:

```
Stack Allocation = 2 bytes

Start         Stop          Length        Name                    Class
00000H        00020H        00021H        NULL_TEXT               CODE
00022H        001DFH        001BEH        _TEXT                   CODE
001E0H        00388H        001A9H        STARTUP                 CODE
00390H        00391H        00002H        EMULATOR_TEXT           CODE
003A0H        003D4H        00035H        WRT2ERR_TEXT            CODE
003D6H        003D6H        00000H        C_ETEXT                 ENDCODE
003D6H        00497H        000C2H        MEM_DATA                FAR_DATA
004A0H        004A0H        00000H        EMULATOR_DATA           FAR_DATA
004A0H        004EBH        0004CH        _DATA                   DATA
004ECH        004F9H        0000EH        CDATA                   DATA
004FAH        00500H        00007H        P_DATA                  DATA
00502H        00505H        00004H        DBDATA                  DATA
00506H        00506H        00000H        XIQC                    DATA
00506H        00506H        00000H        XIFB                    DATA
00506H        00506H        00000H        XIF                     DATA
00506H        00506H        00000H        XIFE                    DATA
00506H        00506H        00000H        XIB                     DATA
00506H        00506H        00000H        XI                      DATA
00506H        00506H        00000H        XIE                     DATA
00506H        00506H        00000H        XPB                     DATA
00506H        00506H        00000H        XP                      DATA
00506H        00506H        00000H        XPE                     DATA
00506H        00506H        00000H        XCB                     DATA
00506H        00506H        00000H        XC                      DATA
00506H        00506H        00000H        XCE                     DATA
00506H        00506H        00000H        XCFB                    DATA
00506H        00506H        00000H        XCFCRT                  DATA
00506H        00506H        00000H        XCF                     DATA
00506H        00506H        00000H        XCFE                    DATA
00506H        00506H        00000H        XIFCB                   DATA
00506H        00506H        00000H        XIFU                    DATA
00506H        00506H        00000H        XIFL                    DATA
00506H        00506H        00000H        XIFM                    DATA
00506H        00506H        00000H        XIFCE                   DATA
00506H        0051FH        0001AH        WRT2                    DATA
00520H        00520H        00000H        CONST                   CONST
00520H        00527H        00008H        HDR                     MSG
00528H        00593H        0006CH        MSG                     MSG
00594H        00595H        00002H        PAD                     MSG
00596H        00596H        00001H        EPAD                    MSG
00598H        00598H        00000H        _BSS                    BSS
00598H        00598H        00000H        XOB                     BSS
0059BH        00598H        00000H        XO                      BSS
00598H        00598H        00000H        XOE                     BSS
00598H        00598H        00000H        XOFB                    BSS
00598H        00598H        00000H        XOF                     BSS
00598H        00598H        00000H        XOFE                    BSS
```

To determine the amount of PLC memory required when **NULL.EXE** is stored to a Series 90-70 CPU, you can replace the variables in the **plc_mem_req** equation with the specific values known/calculated for **NULL.EXE**.

```
plc_mem_req   = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD

GBLDATSIZ     = 598H – 3D6H = 1,432 – 982 = 450 to 16-bit boundary 464
MINALLOC      = 1
EXEIMAGSIZ    = 1,559 to 16-bit boundary 1,568
OVRHEAD       = 530

plc_mem_req   = 464 + (1*16) + 1,568 + 530 = 2,578
```

Impact of Global Data on PLC Memory Usage

Another example of C block impact on PLC memory usage can be seen with the program **NULLDATA.C**:

```
#include "plcc9070.h"

word    wrd_array[512];

main() {
    return(OK);
}
```

The only difference between **NULLDATA.C** and the previous example **NULL.C** is that **NULLDATA.C** contains the declaration of the global variable **wrd_array**. The size of wrd_array in **NULLDATA.C** is 512 words or 1024 bytes. This delta is reflected in the MS-DOS sizes of the two PLC executable files: **NULL.EXE** is 1,559 bytes and **NULLDATA.EXE** is 2,583 bytes. The impact on PLC memory will be more than the 1024 byte difference in files sizes. Using the CPU memory required equation and the **.MAP** file for **NULLDATA**.

```
Stack Allocation = 2 bytes

Start         Stop          Length        Name                    Class
00000H        00020H        00021H        NULLDATA_TEXT           CODE
00022H        001DFH        001BEH        _TEXT                   CODE
001E0H        00388H        001A9H        STARTUP                 CODE
00390H        00391H        00002H        EMULATOR_TEXT           CODE
003A0H        003D4H        00035H        WRT2ERR_TEXT            CODE
003D6H        003D6H        00000H        C_ETEXT                 ENDCODE
003D6H        00497H        000C2H        MEM_DATA                FAR_DATA
004A0H        004A0H        00000H        EMULATOR_DATA           FAR_DATA
004A0H        0089FH        00400H        FAR_BSS                 FAR_BSS
008A0H        008EBH        0004CH        _DATA                   DATA
008ECH        008F9H        0000EH        CDATA                   DATA
008FAH        00900H        00007H        P_DATA                  DATA
00902H        00905H        00004H        DBDATA                  DATA
00906H        00906H        00000H        XIQC                    DATA
00906H        00906H        00000H        XIFB                    DATA
00906H        00906H        00000H        XIF                     DATA
00906H        00906H        00000H        XIFE                    DATA
00906H        00906H        00000H        XIB                     DATA
00906H        00906H        00000H        XI                      DATA
00906H        00906H        00000H        XIE                     DATA
00906H        00906H        00000H        XPB                     DATA
00906H        00906H        00000H        XP                      DATA
00906H        00906H        00000H        XPE                     DATA
00906H        00906H        00000H        XCB                     DATA
00906H        00906H        00000H        XC                      DATA
00906H        00906H        00000H        XCE                     DATA
00906H        00906H        00000H        XCFB                    DATA
00906H        00906H        00000H        XCF                     DATA
00906H        00906H        00000H        XCFCRT                  DATA
00906H        00906H        00000H        XCFE                    DATA
00906H        00906H        00000H        XIFCB                   DATA
00906H        00906H        00000H        XIFU                    DATA
00906H        00906H        00000H        XIFL                    DATA
00906H        00906H        00000H        XIFM                    DATA
00906H        00906H        00000H        XIFCE                   DATA
00906H        0091FH        0001AH        WRT2                    DATA
00920H        00920H        00000H        CONST                   CONST
00920H        00927H        00008H        HDR                     MSG
00928H        00993H        0006CH        MSG                     MSG
00994H        00995H        00002H        PAD                     MSG
00996H        00996H        00001H        EPAD                    MSG
00998H        00998H        00000H        _BSS                    BSS
00998H        00998H        00000H        XOB                     BSS
00998H        00998H        00000H        XO                      BSS
00998H        00998H        00000H        XOE                     BSS
00998H        00998H        00000H        XOFB                    BSS
00998H        00998H        00000H        XOF                     BSS
00998H        00998H        00000H        XOFE                    BSS
009A0H        009A0H        00000H        c_common                BSS
```

plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD

GBLDATSIZ = 998H – 3D6H = 2,456 – 982 = 1,474 to 16-bit boundary 1,488
MINALLOC = 1
EXEIMAGSIZ = 2,583 to 16-bit boundary 2,592
OVRHEAD = 530

plc_mem_req = 1,488 + (1*16) + 2,592 + 530 = 4,626

Thus, in the Series 90-70 CPU, **NULLDATA.EXE** will require 4,626 bytes of memory. Again, the difference in memory usage between **NULL.C** and **NULLDATA.C** is due to the use of global variables in **NULLDATA**.

Impact of Floating Point on PLC Memory Usage

The final two examples of C block impact on PLC memory usage both use the same C source file **NULLFP.C**.

```
#include "plcc9070.h"

main() {
    float x;

    x = 1.3456;

    return(OK);
}
```

The difference in these last two examples will be in how **NULLFP.C** is built. **NULLFP.C** can be built using **MKPLC.BAT**, which will generate a PLC-executable file based upon the **ALTERNATE** math library (**LLIBCA**) and which will execute on any Series 90-70 CPU that supports C blocks. In addition, **NULLFP.C** can also be built using **MKPLC7.BAT**, which will generate a PLC-executable file based upon the **COPROCESSOR** (**8087**) library and which will 0%1 execute on a Series 90-70 CPU that contains a math coprocessor. The **.EXE** and **.MAP** file for **NULLFP.C**, when built using **MKPLC**, will be called **NULLFPA.EXE** and **NULLFPA.MAP**. Similarly, the same two files, when **NULLFP.C** is built using **MKPLC7**, will be called **NULLFP7.EXE** and **NULLFP7.MAP**.

Alternate Library Floating Point

The MS-DOS size of **NULLFPA.EXE** is 12,129 bytes. **NULLFPA.MAP** contains:

```
Stack Allocation = 2 bytes


Start         Stop          Length        Name                    Class
00000H        0002DH        0002EH        NULLFPA_TEXT            CODE
0002EH        025EEH        025C1H        _TEXT                   CODE
025F0H        02798H        001A9H        STARTUP                 CODE
027A0H        027A1H        00002H        EMULATOR_TEXT           CODE
027B0H        027E4H        00035H        WRT2ERR_TEXT            CODE
027E6H        027E6H        00000H        C_ETEXT                 ENDCODE
027E6H        028A7H        000C2H        MEM_DATA                FAR_DATA
028B0H        028B0H        00000H        EMULATOR_DATA           FAR_DATA
028B0H        02D0DH        0045EH        _DATA                   DATA
02D0EH        02D1BH        0000EH        CDATA                   DATA
02D1CH        02D22H        00007H        P_DATA                  DATA
02D24H        02D27H        00004H        DBDATA                  DATA
02D28H        02D28H        00000H        XIQC                    DATA
02D28H        02D28H        00000H        XIFB                    DATA
02D28H        02D28H        00000H        XIF                     DATA
02D28H        02D28H        00000H        XIFE                    DATA
02D28H        02D28H        00000H        XIB                     DATA
02D28H        02D2BH        00004H        XI                      DATA
02D2CH        02D2CH        00000H        XIE                     DATA
02D2CH        02D2CH        00000H        XPB                     DATA
02D2CH        02D2CH        00000H        XP                      DATA
02D2CH        02D2CH        00000H        XPE                     DATA
02D2CH        02D2CH        00000H        XCB                     DATA
02D2CH        02D2CH        00000H        XC                      DATA
02D2CH        02D2CH        00000H        XCE                     DATA
02D2CH        02D2CH        00000H        XCFB                    DATA
02D2CH        02D2CH        00000H        XCFCRT                  DATA
02D2CH        02D2CH        00000H        XCF                     DATA
02D2CH        02D2CH        00000H        XCFE                    DATA
02D2CH        02D2CH        00000H        XIFCB                   DATA
02D2CH        02D2CH        00000H        XIFU                    DATA
02D2CH        02D2CH        00000H        XIFL                    DATA
02D2CH        02D2CH        00000H        XIFM                    DATA
02D2CH        02D2CH        00000H        XIFCE                   DATA
02D2CH        02D45H        0001AH        WRT2                    DATA
02D46H        02D51H        0000CH        CONST                   CONST
02D52H        02D59H        00008H        HDR                     MSG
02D5AH        02E3DH        000E4H        MSG                     MSG
02E3EH        02E3FH        00002H        PAD                     MSG
02E40H        02E40H        00001H        EPAD                    MSG
02E42H        02E45H        00004H        _BSS                    BSS
02E46H        02E46H        00000H        XOB                     BSS
02E46H        02E46H        00000H        XO                      BSS
02E46H        02E46H        00000H        XOE                     BSS
02E46H        02E46H        00000H        XOFB                    BSS
02E46H        02E46H        00000H        XOF                     BSS
02E46H        02E46H        00000H        XOFE                    BSS
02350H        02E57H        00008H        c_common                BSS
```

The amount of Series 90-70 CPU memory used when **NULLFPA.EXE** is stored to the PLC is:

```
plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD

GBLDATSIZ = 2E42H – 27E6H = 11,842 – 10,214 = 1,628 to 16-bit boundary 1,632
MINALLOC = 2
EXEIMAGSIZ = 12,129 to 16-bit boundary 12,144
OVERHEAD = 530

plc_mem_req = 1,632 + (2*16) + 12,144 + 530 = 14,338
```

Therefore, **NULLFPA.EXE**, when stored to the Series 90-70 CPU, will occupy 14,338 bytes of user program memory.

Coprocessor (8087) Library Floating Point

The size of **NULLFP7.EXE** under MS-DOS is 10,294 bytes. The **.MAP** file for **NULLFP** built with the 8087 library contains:

```
Stack Allocation = 2 bytes

Start         Stop     Length           Name                   Class
00000H        00029H   0002AH           NULLFP7_TEXT           CODE
0002AH        017D9H   017B0H           _TEXT                  CODE
017E0H        01988H   001A9H           STARTUP                CODE
01990H        02108H   00779H           EMULATOR_TEXT          CODE
02110H        02144H   00035H           WRT2ERR_TEXT           CODE
02146H        02146H   00000H           C_ETEXT                ENDCODE
02146H        02207H   000C2H           MEM_DATA               FAR_DATA
02210H        02345H   00136H           EMULATOR_DATA          FAR_DATA
02346H        0256FH   0022AH           _DATA                  DATA
02570H        0257DH   0000EH           CDATA                  DATA
0257EH        02584H   00007H           P_DATA                 DATA
02586H        02589H   00004H           DBDATA                 DATA
0258AH        0258AH   00000H           XIQC                   DATA
0258AH        0258AH   00000H           XIFB                   DATA
0258AH        0258AH   00000H           XIF                    DATA
0258AH        0258AH   00000H           XIFE                   DATA
0258AH        0258AH   00000H           XIB                    DATA
0258AH        0258DH   00004H           XI                     DATA
0258EH        0258EH   00000H           XIE                    DATA
0258EH        0258EH   00000H           XPB                    DATA
0258EH        0258EH   00000H           XP                     DATA
0258EH        0258EH   00000H           XPE                    DATA
0258EH        0258EH   00000H           XCB                    DATA
0258EH        0258EH   00000H           XC                     DATA
0258EH        0258EH   00000H           XCE                    DATA
0258EH        0258EH   00000H           XCFB                   DATA
0258EH        0258EH   00000H           XCFCRT                 DATA
0258EH        0258EH   00000H           XCF                    DATA
0258EH        0258EH   00000H           XCFE                   DATA
0258EH        0258EH   00000H           XIFCB                  DATA
0258EH        0258EH   00000H           XIFU                   DATA
0258EH        0258EH   00000H           XIFL                   DATA
0258EH        0258EH   00000H           XIFM                   DATA
0258EH        0258EH   00000H           XIFCE                  DATA
0258EH        02547H   0001AH           WRT2                   DATA
025A8H        025B7H   00010H           CONST                  CONST
025B8H        025BFH   00008H           HDR                    MSG
025C0H        02702H   00143H           MSG                    MSG
02703H        02704H   00002H           PAD                    MSG
02705H        02705H   00001H           EPAD                   MSG
02706H        02719H   00014H           _BSS                   BSS
0271AH        0271AH   00000H           XOB                    BSS
0271AH        0271AH   00000H           XO                     BSS
0271AH        0271AH   00000H           XOE                    BSS
0271AH        0271AH   00000H           XOFB                   BSS
0271AH        0271AH   00000H           XOF                    BSS
0271AH        0271AH   00000H           XOFE                   BSS
02720H        02727H   00008H           c_common               BSS
```

plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD

GBLDATSIZ = 2706H – 2146H = 9,990 – 8,518 = 1,472 to 16-bit boundary 1,472
MINALLOC = 3
EXEIMAGSIZ = 10,294 to 16-bit boundary 10,304
OVRHEAD = 530

plc_mem_req = 1,472 + (3*16) + 10,304 + 530 = 12,354

Series 90-30 Memory Usage Calculation

For any C block, the impact on PLC memory usage may be determined by the following equation:

```
plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD +
                          EXE_stack_size
```

where:

- **GBLDATSIZ** is determined from the **.MAP** file created during the building of the of the PLC-executable version of the C program. The **.MAP** file can be found in the same directory as the built PLC-executable version of the C program. The amount of data space treated as global data is:

  GBLDATSIZ = (the start address of the first segment in the class BSS) − (the start address of the **MEM_DATA** segment).

- **MINALLOC** is determined by executing **EXEHDR.EXE** (provided with Microsoft C compiler) on the PLC-executable **.EXE** file. **EXEHDR** will display information about the **.EXE** file, the value displayed for ***Extra paragraphs needed*** is the value to use for **MINALLOC**.
- **EXEIMAGSIZ** is the size of the PLC-executable .EXE file as determined by the MS-DOS directory command.
- **OVRHEAD** is the value 474 bytes.†
- **EXE_stack_size** is the global variable declared within the program and indicates the size of the C subroutine block's stack.

† OVRHEAD is 474 bytes for Release 8.0 of the Series 90-30 CPU firmware. This value may increase or decrease in future CPU firmware releases.

Smallest Possible Impact on PLC Memory

The C block with the smallest impact on PLC memory would be built from the following source (**NULL.C**):

```
#include "plcc9030.h"
EXE_stack_size = 2048;
main() {
   return(OK);
}
```

When **NULL.C** is built for PLC execution using **MK3PLC.BAT**, the resulting **.EXE** file is 1,803 bytes in size (as displayed by the MS-DOS directory command).

```
 Volume in drive C is 9030_CBLKS
 Directory of  C:\APPS\NULL\PLC

NULL             EXE    1803   2-07-97  11:39a
        1 File(s)   2074624 bytes free
```

The **.MAP** file for **NULL.EXE** contains the following:

```
LINK : warning L4021: no stack segment
Start         Stop          Length        Name                    Class
00000H        00017H        00018H        NULL_TEXT               CODE
00020H        00200H        001E1H        TEXT                    CODE
00210H        00462H        00253H        STARTUP                 CODE
00470H        00471H        00002H        EMULATOR_TEXT           CODE
00480H        004B4H        00035H        WRT2ERR_TEXT            CODE
004B6H        004B6H        00000H        IRDATA_SEG_START        ENDCODE
004B6H        004B6H        00000H        C_ETEXT                 ENDCODE
004B6H        004B6H        00000H        IRDATA                  FAR_DATA
004B6H        00577H        00C2H         MEM_DATA                FAR_DATA
00580H        00580H        00000H        EMULATOR_DATA           FAR_DATA
00580H        005CDH        0004EH        _DATA                   DATA
005CEH        005DBH        0000EH        CDATA                   DATA
005DCH        005E5H        0000AH        P_DATA                  DATA
005E6H        005E9H        00004H        DBDATA                  DATA
005EAH        005EAH        00000H        XIQC                    DATA
005EAH        005EAH        00000H        XIFB                    DATA
005EAH        005EAH        00000H        XIF                     DATA
005EAH        005EAH        00000H        XIFE                    DATA
005EAH        005EAH        00000H        XIB                     DATA
005EAH        005EAH        00000H        XI                      DATA
005EAH        005EAH        00000H        XIE                     DATA
005EAH        005EAH        00000H        XPB                     DATA
005EAH        005EAH        00000H        XP                      DATA
005EAH        005EAH        00000H        XPE                     DATA
005EAH        005EAH        00000H        XCB                     DATA
005EAH        005EAH        00000H        XC                      DATA
005EAH        005EAH        00000H        XCE                     DATA
005EAH        005EAH        00000H        XCFB                    DATA
005EAH        005EAH        00000H        XCFCRT                  DATA
005EAH        005EAH        00000H        XCF                     DATA
005EAH        005EAH        00000H        XCFE                    DATA
005EAH        005EAH        00000H        XIFCB                   DATA
005EAH        005EAH        00000H        XIFU                    DATA
005EAH        005EAH        00000H        XIFL                    DATA
005EAH        005EAH        00000H        XIFM                    DATA
005EAH        005EAH        00000H        XIFCE                   DATA
005EAH        00603H        0001AH        WRT2                    DATA
00604H        00604H        00000H        CONST                   CONST
00604H        0060BH        00008H        HDR                     MSG
0060CH        00677H        0006CH        MSG                     MSG
00678H        00679H        00002H        PAD                     MSG
0067AH        0067AH        00001H        EPAD                    MSG
0067CH        0067CH        00000H        _BSS                    BSS
0067CH        0067CH        00000H        XOB                     BSS
0067CH        0067CH        00000H        XO                      BSS
0067CH        0067CH        00000H        XOE                     BSS
0067CH        0067CH        00000H        XOFB                    BSS
0067CH        0067CH        00000H        XOF                     BSS
0067CH        0067CH        00000H        XOFE                    BSS
```

To determine the amount of PLC memory required when **NULL.EXE** is stored to a Series 90-30 CPU, you can replace the variables in the **plc_mem_req** equation with the specific values known/calculated for **NULL.EXE**.

```
plc_mem_req  = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD +
EXE_stack_size

GBLDATSIZ    = 67CH - 4B6H = 1,660 - 1,206 = 454
MINALLOC     = 1
EXEIMAGSIZ   = 1,803
OVRHEAD      = 474
EXE_stack_size=2048

plc_mem_req  = 454 + (1*16) + 1,803 + 474 + 2,048 = 4,795
```

Impact of Global Data on PLC Memory Usage

Another example of C block impact on PLC memory usage can be seen with the program **NULLDATA.C**:

```
#include "plcc9030.h"
EXE_stack_size = 2048;

word    wrd_array[512];

main() {
    return(OK);
}
```

The only difference between **NULLDATA.C** and the previous example **NULL.C** is that **NULLDATA.C** contains the declaration of the global variable **wrd_array**. The size of wrd_array in **NULLDATA.C** is 512 words or 1024 bytes. This delta is reflected in the MS-DOS sizes of the two PLC executable files: **NULL.EXE** is 1,803 bytes and **NULLDATA.EXE** is 2,827 bytes. The impact on PLC memory will be more than the 1024 byte difference in files sizes. Using the CPU memory required equation and the **.MAP** file for **NULLDATA**.

```
LINK : warning L4021: no stack segment
Start         Stop          Length        Name                    Class
00000H        00017H        00018H        NULLDATA_TEXT           CODE
00020H        00200H        001E1H        _TEXT                   CODE
00210H        00462H        00253H        STARTUP                 CODE
00470H        00471H        00002H        EMULATOR_TEXT           CODE
00480H        004B4H        00035H        WRT2ERR_TEXT            CODE
004B6H        004B6H        00000H        IRDATA_SEG_START        ENDCODE
004B6H        004B6H        00000H        C_ETEXT                 ENDCODE
004B6H        004B6H        00000H        IRDATA                  FAR_DATA
004B6H        00577H        000C2H        MEM_DATA                FAR_DATA
00580H        00580H        00000H        EMULATOR_DATA           FAR_DATA
00580H        0097FH        00400H        FAR_BSS                 FAR_BSS
00980H        009CDH        0004EH        _DATA                   DATA
009CEH        009DBH        0000EH        CDATA                   DATA
009DCH        009E5H        0000AH        P_DATA                  DATA
009E6H        009E9H        00004H        DBDATA                  DATA
009EAH        009EAH        00000H        XIQC                    DATA
009EAH        009EAH        00000H        XIFB                    DATA
009EAH        009EAH        00000H        XIF                     DATA
009EAH        009EAH        00000H        XIFE                    DATA
009EAH        009EAH        00000H        XIB                     DATA
009EAH        009EAH        00000H        XI                      DATA
009EAH        009EAH        00000H        XIE                     DATA
009EAH        009EAH        00000H        XPB                     DATA
009EAH        009EAH        00000H        XP                      DATA
009EAH        009EAH        00000H        XPE                     DATA
009EAH        009EAH        00000H        XCB                     DATA
009EAH        009EAH        00000H        XC                      DATA
009EAH        009EAH        00000H        XCE                     DATA
009EAH        009EAH        00000H        XCFB                    DATA
009EAH        009EAH        00000H        XCFCRT                  DATA
009EAH        009EAH        00000H        XCF                     DATA
009EAH        009EAH        00000H        XCFE                    DATA
009EAH        009EAH        00000H        XIFCB                   DATA
009EAH        009EAH        00000H        XIFU                    DATA
009EAH        009EAH        00000H        XIFL                    DATA
009EAH        009EAH        00000H        XIFM                    DATA
009EAH        009EAH        00000H        XIFCE                   DATA
009EAH        00A03H        0001AH        WRT2                    DATA
00A04H        00A04H        00000H        CONST                   CONST
00A04H        00A0BH        00008H        HDR                     MSG
00A0CH        00A77H        0006CH        MSG                     MSG
00A78H        00A79H        00002H        PAD                     MSG
00A7AH        00A7AH        00001H        EPAD                    MSG
00A7CH        00A7CH        00000H        _BSS                    BSS
00A7CH        00A7CH        00000H        XOB                     BSS
00A7CH        00A7CH        00000H        XO                      BSS
00A7CH        00A7CH        00000H        XOE                     BSS
00A7CH        00A7CH        00000H        XOFB                    BSS
00A7CH        00A7CH        00000H        XOF                     BSS
00A7CH        00A7CH        00000H        XOFE                    BSS
00A80H        00A80H        00000H        c_common                BSS
```

plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD

GBLDATSIZ = A7CH – 4B6H = 2,684 – 1,206 = 1,478
MINALLOC = 1
EXEIMAGSIZ = 2,827
OVRHEAD = 474
EXE_stack_size = 2048

plc_mem_req = 1,478 + (1*16) + 2,827 + 474 + 2,048 = 6,843

Thus, in the Series 90-30 CPU, **NULLDATA.EXE** will require 6,843 bytes of memory. Again, the difference in memory usage between **NULL.C** and **NULLDATA.C** is due to the use of global variables in **NULLDATA**.

Impact of Floating Point on PLC Memory Usage

The final two examples of C block impact on PLC memory usage both use the same C source file **NULLFP.C**.

```
#include "plcc9030.h"
EXE_stack_size = 2048;

main() {
    float x;

    x = 1.3456;

    return(OK);
}
```

The difference in these last two examples will be in how **NULLFP.C** is built. **NULLFP.C** can be built using **MK3PLC.BAT**, which will generate a PLC-executable file based upon the **ALTERNATE** math library (**LLIBCA**) and which will execute on any Series 90-30 CPU that supports C blocks. In addition, **NULLFP.C** can also be built using **MK3PLC7.BAT**, which will generate a PLC-executable file based upon the **COPROCESSOR** (**8087**) library and which will 0%1 execute on a Series 90-30 CPU that contains a math coprocessor. The **.EXE** and **.MAP** file for **NULLFP.C**, when built using **MK3PLC**, will be called **NULLFPA.EXE** and **NULLFPA.MAP**. Similarly, the same two files, when **NULLFP.C** is built using **MK3PLC7**, will be called **NULLFP7.EXE** and **NULLFP7.MAP**.

Alternate Library Floating Point

The MS-DOS size of **NULLFPA.EXE** is 12,357 bytes. **NULLFPA.MAP** contains:

```
LINK : warning L4021: no stack segment
Start         Stop          Length        Name                    Class
00000H        00019H        0001AH        NULLFP_TEXT             CODE
00020H        02600H        025E1H        _TEXT                   CODE
02610H        02862H        00253H        STARTUP                 CODE
02870H        02871H        00002H        EMULATOR_TEXT           CODE
02880H        028B4H        00035H        WRT2ERR_TEXT            CODE
028B6H        028B6H        00000H        IRDATA_SEG_START        ENDCODE
028B6H        028B6H        00000H        C_ETEXT                 ENDCODE
028B6H        028B6H        00000H        IRDATA                  FAR_DATA
028B6H        02977H        000C2H        MEM_DATA                FAR_DATA
02980H        02980H        00000H        EMULATOR_DATA           FAR_DATA
02980H        02DDFH        00460H        _DATA                   DATA
02DE0H        02DEDH        0000EH        CDATA                   DATA
02DEEH        02DF7H        0000AH        P_DATA                  DATA
02DF8H        02DFBH        00004H        DBDATA                  DATA
02DFCH        02DFCH        00000H        XIQC                    DATA
02DFCH        02DFCH        00000H        XIFB                    DATA
02DFCH        02DFCH        00000H        XIF                     DATA
02DFCH        02DFCH        00000H        XIFE                    DATA
02DFCH        02DFCH        00000H        XIB                     DATA
02DFCH        02DFFH        00004H        XI                      DATA
02E00H        02E00H        00000H        XIE                     DATA
02E00H        02E00H        00000H        XPB                     DATA
02E00H        02E00H        00000H        XP                      DATA
02E00H        02E00H        00000H        XPE                     DATA
02E00H        02E00H        00000H        XCB                     DATA
02E00H        02E00H        00000H        XC                      DATA
02E00H        02E00H        00000H        XCE                     DATA
02E00H        02E00H        00000H        XCFB                    DATA
02E00H        02E00H        00000H        XCFCRT                  DATA
02E00H        02E00H        00000H        XCF                     DATA
02E00H        02E00H        00000H        XCFE                    DATA
02E00H        02E00H        00000H        XIFCB                   DATA
02E00H        02E00H        00000H        XIFU                    DATA
02E00H        02E00H        00000H        XIFL                    DATA
02E00H        02E00H        00000H        XIFM                    DATA
02E00H        02E00H        00000H        XIFCE                   DATA
02E00H        02E19H        0001AH        WRT2                    DATA
02E1AH        02E25H        0000CH        CONST                   CONST
02E26H        02E2DH        00008H        HDR                     MSG
02E2EH        02F11H        000E4H        MSG                     MSG
02F12H        02F13H        00002H        PAD                     MSG
02F14H        02F14H        00001H        EPAD                    MSG
02F16H        02F19H        00004H        _BSS                    BSS
02F1AH        02F1AH        00000H        XOB                     BSS
02F1AH        02F1AH        00000H        XO                      BSS
02F1AH        02F1AH        00000H        XOE                     BSS
02F1AH        02F1AH        00000H        XOFB                    BSS
02F1AH        02F1AH        00000H        XOF                     BSS
02F1AH        02F1AH        00000H        XOFE                    BSS
02F20H        02F27H        00008H        c_common                BSS
```

The amount of Series 90-30 CPU memory used when **NULLFPA.EXE** is stored to the PLC is:

```
plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD +
EXE_stack_size

GBLDATSIZ = 2F16H – 28b6H = 12,054 – 10,422 = 1,632
MINALLOC = 2
EXEIMAGSIZ = 12,375
OVERHEAD = 474
EXE_stack_size = 2048

plc_mem_req = 1,632 + (2*16) + 12,357 + 474 + 2,048 = 16,543
```

Therefore, **NULLFPA.EXE**, when stored to the Series 90-30 CPU, will occupy 16,543 bytes of user program memory.

Coprocessor (8087) Library Floating Point

The size of **NULLFP7.EXE** under MS-DOS is 10,620 bytes. The **.MAP** file for **NULLFP** built with the 8087 library contains:

```
LINK : warning L4021: no stack segment
Start         Stop     Length            Name                    Class
00000H        00019H   0001AH            NULLFP_TEXT             CODE
0001AH        01859H   01840H            _TEXT                   CODE
01860H        01AB2H   00253H            STARTUP                 CODE
01AC0H        02238H   00779H            EMULATOR_TEXT           CODE
02240H        02274H   00035H            WRT2ERR_TEXT            CODE
02276H        02276H   00000H            IRDATA_SEG_START        ENDCODE
02276H        02276H   00000H            C_ETEXT                 ENDCODE
02276H        02276H   00000H            IRDATA                  FAR_DATA
02276H        02337H   000C2H            MEM_DATA                FAR_DATA
02340H        02475H   00136H            EMULATOR_DATA           FAR_DATA
02476H        026A3H   0022EH            _DATA                   DATA
026A4H        026B1H   0000EH            CDATA                   DATA
026B2H        026BBH   0000AH            P_DATA                  DATA
026BCH        026BFH   00004H            DBDATA                  DATA
026C0H        026C0H   00000H            XIQC                    DATA
026C0H        026C0H   00000H            XIFB                    DATA
026C0H        026C0H   00000H            XIF                     DATA
026C0H        026C0H   00000H            XIFE                    DATA
026C0H        026C0H   00000H            XIB                     DATA
026C0H        026C7H   00008H            XI                      DATA
026C8H        026C8H   00000H            XIE                     DATA
026C8H        026C8H   00000H            XPB                     DATA
026C8H        026C8H   00000H            XP                      DATA
026C8H        026C8H   00000H            XPE                     DATA
026C8H        026C8H   00000H            XCB                     DATA
026C8H        026C8H   00000H            XC                      DATA
026C8H        026C8H   00000H            XCE                     DATA
026C8H        026C8H   00000H            XCFB                    DATA
026C8H        026C8H   00000H            XCFCRT                  DATA
026C8H        026C8H   00000H            XCF                     DATA
026C8H        026C8H   00000H            XCFE                    DATA
026C8H        026C8H   00000H            XIFCB                   DATA
026C8H        026C8H   00000H            XIFU                    DATA
026C8H        026C8H   00000H            XIFL                    DATA
026C8H        026C8H   00000H            XIFM                    DATA
026C8H        026C8H   00000H            XIFCE                   DATA
026C8H        026E1H   0001AH            WRT2                    DATA
026E2H        026EDH   0000CH            CONST                   CONST
026EEH        026F5H   00008H            HDR                     MSG
026F6H        02838H   00143H            MSG                     MSG
02839H        0283AH   00002H            PAD                     MSG
0283BH        0283BH   00001H            EPAD                    MSG
0283CH        0284FH   00014H            _BSS                    BSS
02850H        02850H   00000H            XOB                     BSS
02850H        02850H   00000H            XO                      BSS
02850H        02850H   00000H            XOE                     BSS
02850H        02850H   00000H            XOFB                    BSS
02850H        02850H   00000H            XOF                     BSS
02850H        02850H   00000H            XOFE                    BSS
02850H        02857H   00008H            c_common                BSS
```

plc_mem_req = GBLDATSIZ + (MINALLOC*16) + EXEIMAGSIZ + OVRHEAD + EXE_stack_size

GBLDATSIZ = 283CH – 2276H = 10,300 – 8,822 = 1,478
MINALLOC = 2
EXEIMAGSIZ = 10,620
OVRHEAD = 474
EXE_stack_size = 2048

plc_mem_req = 1,478 + (2*16) + 10,620 + 474 + 2,048 = 14,652

Appendix E

Series 90-70 CPU Execution Time for printf( )

This appendix illustrates the impact on the PLC sweep of executing a **printf( )** function. Execution times are provided for the Series 90-70 CPU models 731, 732, 771, 772, 781, 782, 914, and 924. The measured times include the time required to format the string, the time required to program the DMA controller to transmit each string, and the overhead time for DMA activity to transmit each of the characters out the CPU serial port.

The times for executing a similarly formatted **sprintf( )** would be approximately the same (slightly less since no DMA activity is required).

Table E-1. Typical Series 90-70 CPU Execution Times for printf()

n a u	t% mn s	P nse s% sp r 0				O2C	O3C
		4 1					
ASCII characters	"–" "–a" "–ab" . . . etc.	1.01 ms for the first character + 0.05 ms for each additional character.		0.92 ms for the first character + 0.02 ms for each additional character.		0.092 ms for the first character + 0.009 ms for each additional character.	0.069 ms for the first character + 0.002 ms for each additional character.
integers "%u" "%d" "%lu" "%ld"	0 +32767 4294967295 2147483647	2.11 ms to 3.32 ms		0.99 ms to 1.54 ms		0.13 ms to 0.31 ms	0.10 ms to 0.17 ms
floating point "%f" "%6.3g"	0 987654.321 etc.	731/771 N/A	732/772	781 N/A	782 1.53 ms to 2.11 ms	0.24 ms to 0.37 ms	0.17 ms to 0.23 ms
long string "%s"	2048 characters in a C string (null terminated)	105.37 ms		44.09 ms		8.92 ms	4.50 ms

Note

The 73x/77x execution times listed in this table are for the IC697CPU731, IC697CPU732, IC697CPU771, and IC697CPU772 CPU. The execution times do ___ include time for processing **printf( )** format flags or preceding fill characters.

Note

Formatting of floating point values is ___ supported on Series 90-70 CPU models 731, 771, or 781.

Appendix F Installing Earlier Compilers

Section 1: Installing the Microsoft C Compiler

Microsoft Visual C, Version 8.0, comes with the purchase of the C Programmer's Toolkit. If, however, you are using the C Programmer's Toolkit with a 90-70 PLC and have an earlier version of the compiler that you prefer to use, the 90-70 PLCs and toolkit are compatible with the earlier version discussed in this chapter. For installation of the Version 8 Microsoft Visual C compiler, refer to the last section of Chapter 2.

Note

If you are creating programs **for a 90-30 PLC**, **Version 8.0 of the Microsoft Visual C compiler is required**.

Installing Microsoft C Version 6.0

Microsoft C Version 6.0 is installed using the **SETUP** program supplied on the Microsoft C diskettes. When this program is run, you are led through a series of options presented on three screens. There are three selections that are of importance when installing this software for use with the C Programmer's Toolkit: math option, memory model, and the question of whether to copy the C startup sources.

Math Options

The math option selection appears on the first screen of the **SETUP** program. Either **Alternate** or **8087** must be selected for the math options. Both options may be selected, if desired.

- Programs linked with the **alternate** library will run on any Series 90-70 CPU model. The **alternate** library will not use a math coprocessor even if one is present.
- If you know that you will be using a CPU model which has a math coprocessor (CPU732, CPU772, CPU782, CPU914, CPU915, CPU924, or CPU925), you should choose the **8087** option. The **8087** library only runs with the coprocessor present and provides better performance of the C application than the **alternate** library.

Memory Models

Select **Large** for the memory model, since Series 90-70 C applications are limited to using the **large** memory model.

- If you choose the **alternate** library and the **large** memory model, a library called **llibca.lib** is installed on your hard disk.
- If you choose the **8087** library and the **large** memory model, a library called **llibc7.lib** is installed on your hard disk. Make sure sure LLIBCE.LIB is not present or if present temorarily rename it before installation of the toolkit.

Copy C Startup Sources Option

The second screen of the **SETUP** program asks a number of questions concerning the installation of additional files. You must answer **Y** (Yes) to the question "Copy C startup sources [N]." This will obtain the files which are needed to build copies of the installed libraries that are compatible with Series 90-70 PLC CPUs.

If you are installing the Microsoft C software for the first time, make the choices indicated above to obtain the files needed to build Series 90-70 PLC CPU compatible libraries. If you have already installed the Microsoft C software and did not select the **large** memory model or did not copy the startup code, you can run **SETUP** again, this time making the required choices.

After the installation is complete, make sure that the Microsoft C **bin** directory is in front of the DOS directory in the MS-DOS path. If the Microsoft C **bin** directory is not in the path or the DOS directory is in front of the Microsoft C **bin** directory, the toolkit will not work properly.

Installing Microsoft C Version 7.0

Like Microsoft C Version 6.0, Microsoft's Version 7.0 C compiler is installed using the **SETUP** program supplied on the Microsoft C diskettes. The main menu of the **SETUP** utility provides a list of all of the numerous options available through the **SETUP** program. To use the Version 7.0 C compiler to create Series 90-70 C applications, you should select: **"Custom Installation of MS C/C++ compiler"**. There are two selections on the Custom Installation menu that are of importance when installing this software for use with the C Programmer's Toolkit: **math options** and **memory models**.

Note

If you are creating programs **for a 90-30 PLC**, **Version 8.0 of the Microsoft Visual C compiler is required**.

Math Options

If this is a first-time installation, you **must** select the following items on the Custom Installation menu:

```
Install Microsoft C/C++ compiler components.

Install C/C++ run-time library components.

Math options:  80x87 chip math, Alternate math.
```

Either **`Alternate math`**, **`80x87 chip math`**, or both must be selected for the math options.

- Programs linked with the **`alternate`** library will run on any Series 90-70 CPU model. The **`alternate`** library will not use a math coprocessor even if one is present.
- If you know that you will be using a CPU model which has a math coprocessor (CPU732, CPU772, CPU782, CPU914, CPU915, CPU924, or CPU925), you should choose the **`80x87 chip math`** option. The **`80x87`** library only runs with the coprocessor present and provides better performance of the C application than the **`alternate`** library.

Memory models

Large must be selected for the memory model, as Series 90-70 C applications are limited to using the large memory model.

- If you choose the **alternate** library and the **large** memory model, a library called **llibca.lib** is installed on your hard disk. If you choose the **80x87** library and the **large** memory model, a library called **llibc7.lib** is installed on your hard disk.
- If you choose both the **80x87 chip math** and the **alternate math**, then both **llibca.lib** and **llibc7.lib** will be created on your hard disk during installation of the Microsoft C compiler. Make sure sure LLIBCE.LIB is not present or if present temorarily rename it before installation of the toolkit.

Other Selections

In addition to the above list of "**MUST**" selects, you may also want to select:

```
Copy CodeView debugger files.

Copy Online Help Files.
```

The remaining available selections are not required for operation with the C Programmer's Toolkit.

If you are installing the Microsoft C software for the first time, make the choices indicated above to obtain the files needed to build Series 90-70 PLC CPU compatible libraries. If you have already installed the Microsoft C software and did not select the **large** memory model or did not select the correct math option, you can run **SETUP** again, this time making the required choices.

After the installation is complete, make sure that the Microsoft C **bin** directory is in front of the DOS directory in the MS-DOS path. If the Microsoft C **bin** directory is not in the path, or the DOS directory is in front of the Microsoft C **bin** directory, the toolkit will not work properly.

Installing Microsoft C Version 8.0

Version 8.0 of the Microsoft Visual C compiler comes with the purchase of the C Programmer's Toolkit; however, if you purchased Version 8.0 previously and wish to install it instead of the slightly limited version that comes with the Toolkit, refer to the installation instructions the came with your Microsoft Visual C compiler or follow the directions shown below.

Like Microsoft Version 6.0 and Version 7.0, Microsoft Version 8.0 is installed using the **SETUP** program supplied on the Microsoft C diskettes. However, the **SETUP** program **must** be run under Microsoft Windows, Windows 95, or Windows NT in order to install Version 8.0.

1. If you are using Windows 95 or Windows NT, skip to the next step. If you are using Windows rather than Windows 95 or Windows NT, verify that **buffers=30** (or higher) is in your **config.sys** file. If it is not set to at least 30, edit your **config.sys** file to read **buffers=30**. Then reboot your PC.
2. Insert the CD-ROM or installation diskettes from Microsoft into the appropriate drive.
3. Using Explorer, File Manager, Startup or My Computer functions, access the **SETUP.EXE** file and double-click it.
4. Follow the prompts, e.g., in the Welcome window, choose **CONTINUE** unless you wish to abort the installation.
5. In the Visual C++ Setup window, select **OK**.
6. In the Installation Options window:
   A. Verify that **Microsoft C/C++ Compiler** is selected (indicated by an X in the box).
   B. Verify that **Run-Time Libraries** is selected.
   C. Click on **Libraries** to display the Library Options windows.
      1) Verify that **Large/Huge** is selected.
      2) Verify that **80x87** and/or **Alternate** is selected **but not Emulator**.
      3) Click on **OK**.
7. Click on **CONTINUE**.
8. Follow the remaining installation instructions.

In addition to this list of **"MUST"** selects, you may also want to select:

- Copy **CodeView** debugger files.
- Copy Online Help files.

The remaining available selections are not required for operation with the C Programmer's Toolkit.

If you are installing the Microsoft C software for the first time, make the choices indicated above to obtain the files needed to build Series 90-70 PLC CPU-compatible

libraries. If you have already installed the Microsoft C software but did not select the **Large/Huge** memory module or did not select the correct math option, you can run **SETUP** again to make the required choices.

After the installation is complete, make sure that the Microsoft C **bin** directory is in front of the DOS directory in the MS-DOS path. If the Microsoft C **bin** directory is not in the path, or the DOS directory is in front of the Microsoft C **bin** directory, the toolkit will not work properly.

A

B

C

D

E

F

G

I

L

M

O

P

R

S

T

V