National Instruments MacBus User Manual

Part Number 320073 - 01

February 1987 Edition

National Instruments
12109 Technology Boulevard
Austin, Texas 78727-6204
(512) 250-9119

Notice About Warranties

Trademarks

WARNING

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. It has been tested using a double-shielded IEEE-488 cable (National Instruments 763061-0X or Hewlett-Packard Model 10833 or equivalent) and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference, in which case the user at his expense will be required to take whatever measure may be necessary to correct the interference.

If the equipment does cause interference to radio or television reception, which can be determined by turning the equipment on and off, one or more of the following suggestions may reduce or eliminate the problem:

. Operate the computing device and receiver on different branches of your AC electrical system.
. Move the computing devices away from the receiver with which the computing device is interfering.
. Reposition the computing device or receiver.
. Reposition the receiver's antenna.
. Unplug any unused I/O cables - unterminated I/O cables are a potential source of interference.
. Remove any unused circuit boards - unterminated circuit boards are also a potential source of interference.
. Be sure the computing device is plugged into a grounded outlet and that the grounding has not been defeated with a cheater plug.

If none of these measures resolves your interference problems, contact the manufacturer or write to the US Government Printing Office, Washington, DC 20402, for the booklet How ***to Identify and Resolve Radio-TV Interference Problems,*** Stock Number 004-000-00034504.

CONTENTS

SECTION ONE - INTRODUCTION 1-1

MACBUS INTERFACE KIT DESCRIPTION 1-1

Enclosure 1-3

The GPIB-V50 Board 1-4

SCSI-PC Board 1-5

EQUIPMENT SUPPLIED 1-6

MACBUS HARDWARE DISTRIBUTION DISK 1-8

IBDIAG 1-8

IBCL Window 1-9

Run IBCL Window 1-9

Exit IBCL Window 1-10

IBCLload 1-10

Run IBCLload 1-11

SCSI_CONF 1-12

Run SCSI_CONF 1-13

Exit SCSI_CONF 1-13

gpib.com 1-14

IBCL SOURCE FILES 1-14

Scaler 1-15

MicroASM 1-15

HiFlow 1-15

721911BCL and 72191ASM 1-15

String 1-15

Forth83 1-16

Toolbox 1-16

case 1-16

BCD 1-16

Upper Case IBCL 1-16

MACBUS SOFTWARE DISTRIBUTION DISK 1-17

IBIC 1-17

IBCONF 1-17

SECTION TWO - INSTALLATION 2-1

INSPECTION 2-1

INSTALLATION 2-1
Verify Voltage Requirements 2-1
Install Internal Options 2-1
Connect Cables 2-4
Turn Power Switch to ON 2-6
Run the Diagnostic Software 2-6
SECTION THREE - IBCL 3-1
INTRODUCTION 3-1
LANGUAGE STRUCTURE 3-1
Stacks 3-3
Dictionary 3-4
Vocabularies 3-5
Definitions 3-6
Postfix Notation 3-7
Command Line Input 3-7
THE IBCL INTERPRETERS 3-8
Inner Interpreter Sequence 3-8
Outer Interpreter Sequence 3-9
Errors 3-12
STACK MANIPULATION 3-13
Return Stack 3-15
ARITHMETIC & LOGIC 3-16
Constants 3-17
Unary Operators 3-18
Binary and Ternary Operators 3-19
Signed or Unsigned Operands 3-20
Signed Operands 3-20
Mixed Length Signed Operands 3-21
Unsigned Operands 3-21
Logical, Sign Bit Not Significant 3-21
External Double Number Words 3-22
MEMORY ACCESS 3-23
Load and Store 3-23
Fill 3-26
Move 3-27
String Functions 3-28
String Literals 3-30
IBCL INPUT 3-31

ASCII-Type Input 3-31
Binary-Type Input 3-33
IBCL OUTPUT 3-34
ASCII-Type Output Words 3-34
Character-Based Words 3-36
Numeric-Based Words 3-37
Binary-Type Output 3-40
GENERAL PORT I/O 3-40
LOADING PROGRAMS 3-42
DEFINING NEW WORDS 3-42
Colon Definitions 3-43
Machine Code Definitions 3-48
Constants, Variables, and Arrays 3-49
Vocabularies 3-52
Program Segments 3-54
Defining Defining Words 3-56
Internal Workings 3-58
Headerless Words 3-59
Overlaid Code Primitives 3-60
Forget-Task 3-60
CONTROL 3-62
Vectored Execution 3-62
Conditional Execution 3-64
Loops 3-65
Case 3-67
USING ASSEMBLY LANGUAGE FROM
IBCL 3-68
Assembler Mnemonics 3-69
Macro Definition 3-70
Branch Control 3-71
Things to Remember 3-71
NEC V40/V50 Architecture 3-72
Registers 3-72
Memory Addressing 3-73
Flags 3-75
Abbreviations and Conventions 3-75
Error Messages 3-76
Timing 3-77

Instruction Set 3-77
Data Transfer 3-77
Arithmetic and Logic 3-80
Rotate and Shift 3-84
String Manipulation 3-85
Jumps, Calls, and Loops 3-86
Conditional Execution & Loops 3-91
Processor Control 3-95
Interrupts 3-97
CoProcessor Support 3-98
NEC 72191 Architecture 3-99
Data registers 3-99
Status Word 3-99
Control Word 3-101
Tag Word 3-102
Data Types 3-102
NEC 72 19 1 Instructions 3-104
Data Type Selection 3-104
Instruction Format 3-105
Data Transfer 3-106
Comparison 3-107
Constants 3-109
Arithmetic 3-109
Transcendental 3-112
Processor Control 3-113
SEGMENT MANAGEMENT 3-115
Relocation Tools 3-115
GPIB Control Words 3-117
GPIB Status Variables 3-118
Associating Names with GPIB Devices 3-118
Device Level Words 3-119
MacBus GPIB Port Level Words 3-138
Configuration Control Words 3-163
GPIB Port and Device Configuration Tables 3-171
Advanced Features 3-172
Additional GPIB Words 3-174

Configuration Control 3-174
GPIB-Macintosh Direct Transfer Words 3-175
IBCL Heap Management Words 3-176
Memory Dumping Words 3-177
SECTION FOUR - TECHNICAL INFORMATION 4-1
PHYSICAL SPECIFICATIONS 4-1
ENVIRONMENTAL SPECIFICATIONS 4-1
ELECTRICAL SPECIFICATIONS 4-2
DETAILED DESCRIPTION 4-3
Enclosure 4-3
SCSI-PC Interface 4-4
SCSI-PC I/O Addresses 4-6
SCSI ID Switch Settings 4-7
DMA Settings 4-8
Interrupt Settings 4-9
Terminating Resistors 4-9
The GPIB-V50 4-10
SECTION FIVE - DIAGNOSTICS’&’ TROUBLESHOOTING 5-1
WHERE TO BEGIN 5-1
CHECK ELECTRICAL’ CONNECTIONS 5-2
ISOLATE THE SYSTEM 5-2
RUN DIAGNOSTIC SOFTWARE 5-3

APPENDIX A - MULTILINE INTERFACE MESSAGES A-1
APPENDIX B - THE GPIB B-1
TYPES OF MESSAGES B-1
TALKERS, LISTENERS, AND CONTROLLERS B-1
THE CIC AND SYSTEM CONTROLLER B-2
GPIB SIGNALS AND LINES B-3
Data Lines B-3
Handshake Lines B-3
Interface Management Lines B-4

PHYSICAL AND ELECTRICAL CHARACTERISTICS B-5
CONFIGURATION REQUIREMENTS B-9
RELATED DOCUMENTS. B-9
APPENDIX C - IBCL COMMAND SUMMARY C-1
APPENDIX D - IBCL TUTORIAL D-1
INTRODUCTION D-1
THE LANGUAGE D-1
TUTORIAL D-2
Starting the IBCL Window D-2
Defining New Words D-4
Loops and Conditionals D-5
Stack Words D-7
More Looping D-8
Forgetting D-10
GPIB Functions D-11
Exiting the IBCL Window D-11
APPENDIX E - SCALED NUMBERS E-1
MEMORY E-3
STACK E-4
COMPARISON E-4
CONSTANTS E-5
MATH E-5
TRANSCENDENTAL E-6
CONVERSIONS E-7
TYPE CHECKING E-8
INPUT/OUTPUT E-8
NEC 72191 CONTROL E-9
NEC 72191 Notes E-10
HIGH PRECISION I/O E-10
APPENDIX F - USING LABVIEW WITH MACBUS F-1
WHY USE LABVIEW WITH MACBUS? . . . .
WHY USE MACBUS WITH LABVIEW? F-2
THE LABVIEW MACBUS INTERFACE F-3

LABVIEW/MACBUS EXAMPLES F-4
Done Poll F-5
Ulm Poll F-8
Dlm Poll F-11
Higher Level MacBus Interface
Instruments F-14
String to MacBus F-14
String from MacBus F-17
MacBus String I/O F-20
MacBus DLM F-23
MacBus ULM F-27
Applications of These Example
Instruments F-30

LIST OF FIGURES

Figure 1- 1 - MacBus Connected to a Macintosh Plus 1-2
Figure l-2 - GPIB-V50 Card 1-4
Figure l-3 - SCSI-PC Card 1-5
Figure 2-l - Rear Panel of MacBus 2-4
Figure 4-2 - SCSI-PC Parts Locator Diagram 4-5
Figure 4-3 - SCSI ID Switch Settings 4-7
Figure 4-4 - SCSI-PC DMA Channel 3 Setting 4-8
Figure 4-5 - SCSI-PC Interrupt Level 5 Setting 4-9
Figure B-l - GPIB Connector and the Signal Assignment B-6
Figure B-2 - Linear Configuration B-7
Figure, B-3 - Star Configuration B-8
Figure F-l - LabVIEW/MacBus Interface Instrument Icons F-3
Figure F-2 - Done Poll Icon, Connector Pane, and Connector Pane Controls F-6
Figure F-3 - Done Poll Panel F-6
Figure F-4 - Done Poll Diagram F-7
Figure F-5 - Ulm Poll Icon, Connector Pane, and Connector Pane Controls F-9
Figure F-6 - Ulm Poll Panel F-9
Figure F-7 - Ulm Poll Diagram F-10
Figure F-8 - Dlm Poll Icon, Connector Pane, and Connector Pane Controls F-12
Figure F-9 - Dlm Poll Panel F-12
Figure F- 10 - Dlm Poll Diagram F-13
Figure F- 11 - String to MacBus Icon, Connector Pane, and Connector Pane Controls F-15

Figure F- 12 - String to MacBus Panel F-15
Figure F- 13 - String to MacBus Diagram F-16
Figure F- 14 - String From MacBus Icon, Connector Pane, and Connector Pane Controls F-18
Figure F- 15 - String From MacBus Panel F-18
Figure F-16 - String From MacBus Diagram F-19
Figure F-17 - MacBus String I/O Icon, Connector Pane, and Connector Pane Controls F-21
Figure F-18 - MacBus String I/O Panel F-21
Figure F-19 - MacBus String I/O Diagram F-22
Figure F-20 - MacBus DLM Icon, Connector Pane, and Connector Pane Controls F-25
Figure F-21 - MacBus DLM Panel F-25
Figure F-22 - MacBus DLM Diagram F-26
Figure F-23 - MacBus ULM Icon, Connector Pane, and Connector Pane Controls F-28
Figure F-24 - MacBus ULM Panel F-28
Figure F-25 - MacBus ULM Diagram F-29

LIST OF TABLES

TABLE l-l - Equipment Supplied in the MacBus Kit l-6
TABLE l-2 - Accessory Equipment for MacBus Kit l-7

PREFACE

Following is a description of each section of the MacBus User Manual.

SECTION ONE, "INTRODUCTION," contains a brief description of MacBus, a list of equipment supplied, and a list of optional equipment.

SECTION TWO, "INSTALLATION," contains information on the installation of the MacBus.

SECTION THREE, "IBCL," contains a description of the IBCL (Interface Bus Control Language) operating system.

SECTION FOUR, "TECHNICAL INFORMATION," contains a description of technical information for MacBus.

SECTION FIVE, "DIAGNOSTICS AND TROUBLESHOOTING," contains the diagnostics and troubleshooting procedures for MacBus.

APPENDIX A, "MULTILINE INTERFACE MESSAGES," contains an ASCII chart with GPIB messages.

APPENDIX B, "THE GPIB," contains a description of the operation of the GPIB.

APPENDIX C, "IBCL COMMAND SUMMARY," contains a list of all IBCL commands.

APPENDIX D, "IBCL TUTORIAL," contains a brief tutorial of the highlights of IBCL.

APPENDIX E, "SCALED NUMBERS," contains a brief description of the scaled numbers used in IBCL

APPENDIX F, "USING LABVIEW WITH MACBUS," contains examples on how to use MacBus with LabVIEW.

SECTION ONE - INTRODUCTION

This section contains general information about the National Instruments MacBus Kit. This information includes a brief description of the MacBus Kit, a list of equipment supplied, and a list of optional equipment.

MACBUS INTERFACE KIT DESCRIPTION

MacBus, shown in Figure l-l, is a powerful microcomputer designed to give you the best of both worlds; the convenience and ease of use of the Macintosh Plus and the power and expandability of an IBM PC AT. MacBus communicates with the Macintosh Plus through a high-speed parallel interface. This interface gives the Macintosh Plus, which up until now has had closed system architecture, true expandability.

Figure l-l - MacBus Connected to a Macintosh Plus

The base-line MacBus consists of three basic components: an enclosure, a **GPIB-V50** microprocessor board, and a SCSI-PC interface board for connection to the Macintosh Plus. Each component is described in further detail in the following sections.

Enclosure

The enclosure is a small metal "AT-style" case that can be placed on any flat surface. The dimensions of the enclosure are 6 inches high by 11 3/4 inches wide by 15 1/4 inches deep. A switching power supply is mounted inside the box and is connected to the motherboard which carries power to the I/O cards. The motherboard has connectors for five AT-style cards, two of which are used by the GPIB-V50 microprocessor card and the SCSI-PC interface card. This leaves three slots available for optional adapters.

The GPIB-V50 Board

The GPIB-V50 board, shown in Figure 1-2, is a plug-in processor board with onboard RAM, ROM, and a GPIB port. The GPIB-V50 is based on the NEC V50 microprocessor which is a high integration, 16-bit, 8086-type microprocessor. The GPIB-V50 also supports NEC's uPD72191 numeric coprocessor. Both the microprocessor and coprocessor run at 8 MHz and use advanced CMOS technology. The GPIB-V50 also implements a subset of the IBM PC AT bus signals so that many existing IBM PC and IBM PC AT compatible boards may be used with the MacBus.

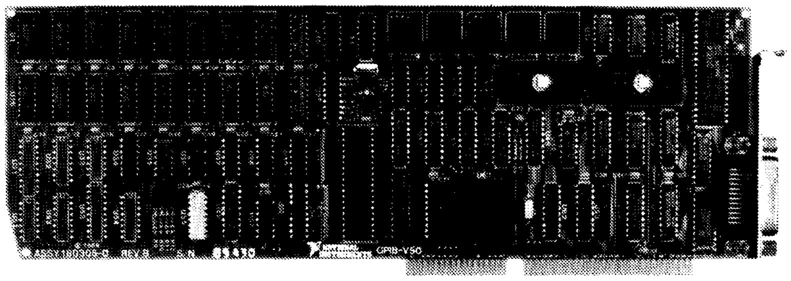


Figure 1-2 - GPIB-V50 Board

SCSI-PC Board

The SCSI-PC board, shown in Figure l-3, is a Small Computer Systems Interface (SCSI) to IBM PC bus interface which is compatible with the GPIB-V50 board. The SCSI-PC interface is a "short slot" board which is driven by the ROM based operating system on the GPIB-V50 board. The SCSI-PC uses the NCR 5380 as the SCSI controller chip and has the ability of terminating all SCSI signal lines directly on the board.

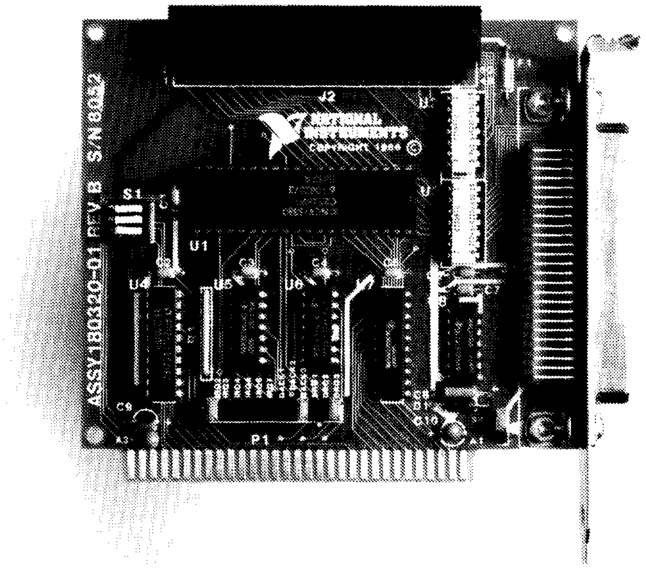


Figure l-3 - SCSI-PC Board

EQUIPMENT SUPPLIED

Table l-l lists the equipment supplied in the National Instruments MacBus Kit. Optional equipment is listed in Table l-2.

TABLE l-l - Equipment Supplied in the MacBus Kit

Description	**Part Number**
MacBus Hardware Unit **(115V)**	
with 128K bytes RAM	776139-01 *
with 512K bytes RAM	776 139-02 *
MacBus Hardware Unit (230V)	
with 128K bytes RAM	776139-03 *
with 512K bytes RAM	776139-04 *

* Includes:
- Chassis with power supply and power cord
- **GPIB-V50** microprocessor board with GPIB interface
- SCSI-PC interface board
- Distribution disk with IBDIAG, IBCL Window, **IBCLload,** SCSI_CONF, and gpib.com files **
- MacBus User Manual

** For a description of the tiles included on the distribution disk, refer to MACBUS HARDWARE DISTRIBUTION DISK paragraph in this section.

TABLE l-2 - Accessory Equipment for MacBus Kit

Description	Part Number
MacBus RAM Memory Field Upgrade Kit (128K to 512K)	776139-03
MacBus Software Includes: - Distribution disk with IBIC, IBCONF, and Microsoft BASIC language interface * - Software Reference Manual	776140-01
LabVIEW Software (double sided floppies)	776141-01
SCSI Cable (External) Macintosh Plus to MacBus MacBus to SCSI peripheral	 180323-01 180330-01
SCSI Extension Cable (Internal)	180329-01
Megamax C Language Interface (includes Manual, requires MacBus software package)	776140-02
MacBus Technical Reference Manual	320075-01

* For a description of the files included on the distribution disk, refer to MACBUS SOFTWARE DISTRIBUTION DISK paragraph in this section.

MACBUS HARDWARE DISTRIBUTION DISK

The following files are included with your MacBus hardware unit on your distribution disk.

- IBDIAG
- IBCL Window
- IBCLload
- SCSI_CONF
- gpib.com

Refer to the following paragraphs for a description of each file.

IBDIAG

IBDIAG is a utility which provides the results of the onboard diagnostics run on MacBus at power-on. There are five diagnostic tests which are run. They are:

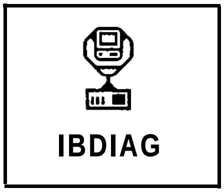
1. RAM test
2. ROM test
3. Non-interfering GPIB test
4. CPU test of internal registers
5. CPU test of onboard peripherals

The IBDIAG utility displays the results of these diagnostics. A zero indicates no error. The remaining error codes are defined as follows:

Error **Code**	Description
0	No errors
1	ROM checksum failure
2	CPU internal register failure
3	CPU onboard peripheral failure
4	RAM failure
5	GPIB failure
6	SCSI bus error

Refer to SECTION FIVE of this manual for troubleshooting the above errors.

Run IBDIAG by double-clicking the IBDIAG icon (shown below).



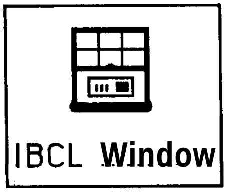
IBCL Window

The IBCL **Window** utility allows interactive communication with the IBCL operating system on MacBus. IBCL commands are entered directly to the IBCL interpreter on MacBus. Any output or error messages generated are displayed on the Macintosh screen.

Run IBCL Window

The IBCL **Window** program is provided as an executable file on your distribution disk.

To run **IBCL Window,** insert the distribution disk into the disk drive and double-click the disk icon displaying the file icons. Now double-click the **IBCL Window** icon (shown below).



You will see the IBCL prompt *ok* appear on your screen. If you see:

```
Error : ECOMM
Reset MacBus and Try Again ?
```

reset your MacBus by turning the power off and back on again.

Wait at least 10 seconds for the power-up diagnostics to complete and type 'y' **<cr>.** You will then see the IBCL prompt *ok.*

Exit IBCL Window

To exit the IBCL Window type *stop.* This will return you to the Macintosh Plus Finder.

IBCLload

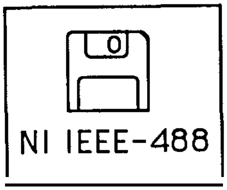
The **IBCL** utility, **IBCLload, allows** files containing **IBCL** commands to be sent directly to the **IBCL** interpreter on MacBus.

With **IBCLload, you** can send your **IBCL** programs to MacBus and then open the **IBCL** interactive window for direct access to the resources of MacBus.

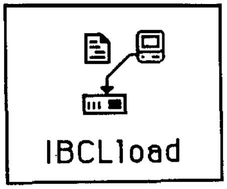
Run IBCLload

The **IBCLload** program is provided as an executable file on your distribution disk.

To run **IBCLload,** insert the distribution disk into the disk drive and double-click the disk icon displaying the file icons (shown below).



Now double-click the **IBCLload** icon (shown below).



The following screen will appear on your Macintosh screen:

```
National Instruments
IBCL File Transfer Utility Rev A.1
Copyright (C) 1986 National Instruments, Inc.
All rights reserved.

Type 'help' for help.
Type 'quit' or 'exit' to exit program
```

There are six commands in the **IBCLload** utility. They are:

load **<filename>**:	Send IBCL command tile to MacBus
type **<filename>**:	Display **file** to screen
'cmd' c:	Stop displaying file to screen or break from 'hung' bus.
ibcl:	Enter IBCL interactive window
stop:	Return from IBCL interactive window
'q' or 'e':	Exit **IBCLload** utility

Any IBCL error generated by your source tile will be displayed on your screen.

Any IBCL source file can also be downloaded from a BASIC or C program. The function format is:

```
ibclload(filename)
char *filename;
```

Any IBCL errors generated are written to the **file** IBCL.OUT. The editor can open this **file** for viewing. You must enter the editor first and then open the file. Your BASIC or C program containing the ibclload call must be linked to **CIB.O** contained on your distribution disk.

SCSI_CONF

The utility SCSI_CONF allows the SCSI ID of MacBus to be changed from its default setting of 6.

The Macintosh Plus has a fixed SCSI ID of 7. Valid SCSI **IDs** for the remaining devices are 0 to 6. MacBus has three hardware switches located on the SCSI interface board (see the MacBus User Manual for more information). These switches must be configured so that they are consistent with the software setting in this field. The factory default setting is SCSI ID = 6.

The current SCSI ID is maintained in the **file** gpib.com. This tile is provided on the distribution disk.

If you are operating under the Macintosh hierarchical **file** system, the file gpib.com must reside in one of three locations for

SCSI_CONF and other MacBus utilities to have access to it.

- The directory (or folder) where the application SCSI_CONF and/or other MacBus utilities reside.
- The system folder.
- The root directory (or folder).

The file must be named gpib.com.

When executed, SCSI_CONF first looks for the gpib.com file in the above mentioned locations. If SCSI_CONF cannot find gpib.com, you will be prompted to choose to have it created with factory ID setting of 6. The file will be placed in the system folder.

*Run **SCSI_CONF***

The SCSI_CONF program is provided as an executable file on you distribution disk.

To run SCSI_CONF, insert the distribution disk into the disk drive and double-click the disk icon displaying the file icons. Now double-click the SCSI_CONF icon.

A help screen will appear on your Macintosh Plus screen. The current SCSI ID setting is displayed. Enter the new SCSI ID by typing a number from zero to six. SCSI_CONF will verify your entry by re-displaying to the screen.

*Exit **SCSI_CONF***

Exit SCSI_CONF by typing **'e'** or **'q'.** The new SCSI ID is saved to the file gpib.com where other MacBus applications have access to it.

gpib.com

The file gpib.com contains the current SCSI ID information. It must reside in one of three locations (see SCSI_CONF paragraph above for details).

IBCL SOURCE FILES

The remaining files on the distribution disk contain IBCL source code. These files are:

- Scaler
- **MicroASM**
- **HiFlow**
- 721911BCL
- 72191ASM
- String
- Forth83
- Toolbox
- case
- BCD
- Upper Case IBCL

To use these files you must first download them to the IBCL interpreter. There are two methods of doing this. The first method uses the **IBCLload** utility. This utility was described in the **IBCLload** paragraph mentioned earlier. The second method downloads an IBCL source file to MacBus through functions or subroutines in a language interface library. This method may only be used if you have purchased a MacBus language interface library from National Instruments. Consult the appropriate language interface supplement for details.

Scaler

This tile contains IBCL source code allowing manipulation of software floating point numbers. The IBCL words defined in **Scaler** are documented in APPENDIX E - SCALED NUMBERS.

MicroASM

This file contains IBCL source code allowing you to use INTEL 8086/88 sytle assembly language mnemonics to create very fast IBCL words. For more details see SECTION THREE - USING ASSEMBLY LANGUAGE FROM IBCL.

HiFlow

This file contains IBCL source code allowing you to use high-level control flow constructs such as **begin...again,** **begin...while...repeat,** etc. These IBCL words are documented in SECTION THREE - Conditional Execution & Loops.

721911BCL and 72191ASM

These files contain IBCL source code allowing manipulation of the NEC 72191 floating point coprocessor. 721911BCL contains high-level IBCL definitions and may be downloaded at any time. 72191ASM contains assembler extensions allowing you to use 72191 opt codes in IBCL assembly language definitions. You must download the assembler utility **MicroASM** before you download 72191ASM. The definitions contained in these two files are documented in SECTION THREE - NEC 72191 Architecture.

String

This file contains IBCL source code allowing convenient manipulation of ASCII strings. The IBCL words are documented in SECTION THREE - String Functions.

Forth83

As shipped by National Instruments, IBCL closely approximates a FORTH79 standard system. **Forth83** contains IBCL source code that will bring IBCL into compatibility with the Forth83 standard. These words are not documented in this manual; consult any Forth-83 reference for their usage.

Toolbox

This file contains IBCL source code which supports IBCL debugging.

case

This executable file converts the contents of your text file from uppercase to lowercase to allow your file to run with IBCL.

BCD

This file contains IBCL source code providing IBCL support for Binary Coded Decimal numbers.

Upper Case IBCL

This file contains IBCL source code which will convert the words in the IBCL dictionary from lowercase ot uppercase.

MACBUS SOFTWARE DISTRIBUTION DISK

The two tiles included on your software distribution disk are:

. IBIC

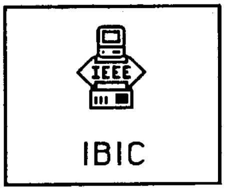
. IBCONF

Refer to the following paragraphs for a description of each file.

IBIC

The IEEE-488 Bus Interactive Control utility (IBIC) allows commands contained in the C Language and BASIC Languages libraries to be run interactively from a terminal.

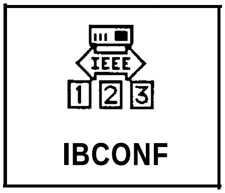
Start IBIC by double-clicking the IBIC icon (shown below).



IBCONF

The IEEE-488 Bus Configuration utility (IBCONF) is a screen-oriented, interactive program. IBCONF is used to edit the characteristics of the board and devices in the system. You must run IBCONF to change the MacBus SCSI ID if it is different from the factory setting of 6.

Start IBCONF by double-clicking the IBCONF icon (shown below).



SECTION TWO - INSTALLATION

INSPECTION

Before you install MacBus, inspect the shipping container and its contents for damage. If damage appears to have been caused in shipment, tile a claim with the carrier. Retain the packaging material for possible inspection or for reshipment.

If the equipment appears to be damaged, do not attempt to operate it. Contact National Instruments for instructions.

INSTALLATION

There are five basic steps to installing MacBus.

1. Verify voltage requirements.
2. Install optional internal adapters.
3. Connect cables.
4. Turn power switch to on.
5. Run the diagnostic software.

Verify **Voltage Requirements**

MacBus is shipped from the factory set at a particular operating voltage, either 115 VAC or 230 VAC, and cannot be changed by the user.

Verify that the voltage you are using is the same as the voltage on the label on the back of the MacBus chassis. If it is not, contact National Instruments for further instructions.

Install Internal Options

The basic MacBus computer consists of a system unit with 2 cards, a GPIB-V50 microprocessor card and a SCSI-PC interface card for connection to the Macintosh Plus. You may add compatible options to expand your system to meet your particular needs.

NOTE

> Before attempting to install an option on MacBus, check with National Instruments to verify that the option is supported on MacBus. Use of other manufacturers cards are the sole responsibility of the customer.

If you do not need to install internal options, you may skip directly to the paragraph Connect Cables. The steps for installing optional cards are listed below.

1. Turn the system unit's power switch to the OFF position.
2. Remove any externally attached devices from the MacBus back panel.
3. Unplug your system unit's power cord from the wall outlet, and also from the back of MacBus.
4. Remove the four cover mounting screws located in each corner of the rear panel.
5. Remove the cover by sliding it towards the front of the machine in a straight line.
6. Some optional cards can be damaged by static electricity caused by handling of the cards. Make sure you take the following precautions when installing optional cards.
    a. Use a grounded wrist-strap and work on a grounded work area.
    b. Remove the optional cards from their shipping containers carefully and hold the adapters by their edges only.
    c. Avoid touching any connections or components on the optional adapters or on any card already installed in the unit.

   d. When installing options, hold the adapter by their upper corners or top edges only.

7. Remove the screw that holds the expansion slot's cover in place, then remove the cover.

8. Firmly press the adapter into the expansion slot and reinstall the screw removed in Step 7. The I/O connectors on the motherboard are slot independent, that is, any adapter can be installed in any available slot.

9. When adding optional interface adapters, it may be necessary to change such parameters as I/O addresses, interrupt levels, or DMA channels, if they conflict with those of the SCSI-PC card. It may also be necessary to change the SCSI ID or to remove the power resistors in your system. To determine if these steps are necessary see SECTION FOUR - TECHNICAL INFORMATION.

10. Install the system unit's cover and reinstall the four cover mounting screws removed in Step 4.

Connect Cables

You must connect two cables to operate MacBus: the power cable and the SCSI cable, Optionally, if the GPIB is used, a shielded GPIB cable (not included) will need to be installed also. All cables connect to MacBus via the rear panel, as shown in Figure 2-l. When connecting the power cord, make sure it is plugged into a GROUNDED outlet of the proper voltage or damage to the unit could occur.

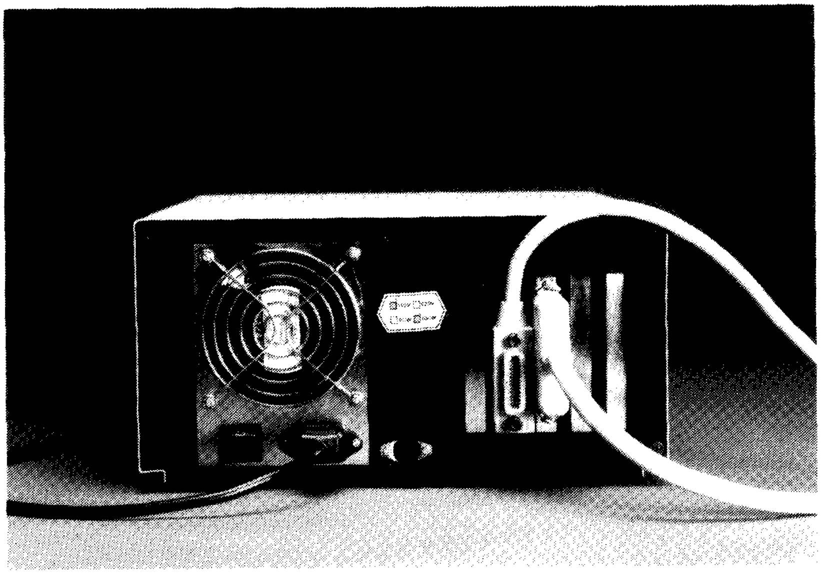


Figure 2-l - Rear Panel of MacBus

Some connector pairs use strain-relief locks to ensure proper signal connections. Make sure these locks are used if they are provided. IEEE-488 cables usually come with two screw lock assemblies on each side of the connector. The SCSI connector comes with two spring retainers. Make sure that the cables are pressed firmly into the connectors and that the strain-relief mechanisms are used.

The Macintosh Plus uses a 25-pin D-subminiature connector for its SCSI interface. Locate this connector on the rear panel of the Macintosh Plus, designated by a small symbol of a disk drive, and connect the remaining end of the SCSI connector to the rear of the Macintosh Plus.

Special Note for Multi-SCSI-Device Users

If other SCSI devices are to be used, two options exist for daisy-chaining the SCSI cable to the remaining devices. For the first option, the MacBus cable can be connected to other SCSI devices by using the optional SCSI extension cable, NI part number 180329-01. If this method is used, the SCSI signals will be input to MacBus through the rear panel connector on the SCSI-PC board and the signals will be propagated to the next SCSI device via a SCSI connector on a rear back panel slot which is connected to the SCSI-PC through a mass-terminated ribbon connector. If this method is used and there is a SCSI device attached to the SCSI connector, the SCSI terminating resistors on the SCSI-PC board should not be used (see SECTION FOUR – TECHNICAL INFORMATION).

A second method to connect MacBus into a multi-SCSI device environment is to make MacBus the last device on the SCSI bus. This is a desired configuration for two reasons. First, since there are no signals to daisy-chain to another device, you will not need the optional SCSI extension cable and, therefore, you will free up a slot in MacBus for other I/O boards. Second, you will not need Apple's SCSI cable terminator because all SCSI signals can be terminated on the SCSI-PC (see SECTION FOUR – TECHNICAL

INFORMATION).

Another important consideration for multiple SCSI device users is to make sure that no two SCSI devices have the same SCSI ID. The SCSI-PC is shipped from the factory with its SCSI ID set to 6. If this conflicts with another SCSI ID, one of the devices' IDs will need to be changed. To change the SCSI ID on the SCSI-PC refer to SECTION FOUR - TECHNICAL INFORMATION.

Turn Power Switch to ON

The next step to setting up MacBus is turning the unit's power switch to ON. Wait 10 seconds before appling power to the Macintosh Plus so that MacBus has time to complete its power-on self-test. If changes have been made to certain adapters, such as changing the SCSI ID on the SCSI-PC, they will take effect when the MacBus unit is powered on.

Run the Diagnostic Software

The last step to setting up MacBus is to run the diagnostic software to verify the system was set up correctly. The diagnostic software is included on the distribution disk which came with your MacBus Kit. It can be run by double-clicking on the IBDIAG icon. If the system was installed correctly, the diagnostic software will print "Diagnostic Completed Successfully" on the Macintosh Plus screen. If any other result is posted, refer to SECTION FIVE - DIAGNOSTICS AND TROUBLESHOOTING.

SECTION THREE - IBCL

INTRODUCTION

IBCL (Interface Bus Control Language) is a powerful interactive programming language used to program MacBus. IBCL resides in a pair of EPROM memory chips on the GPIB-V50 board. IBCL boots automatically when you power on MacBus; no initialization sequence is necessary.

IBCL is a stack-based language that can be tailored to specific applications by the addition of new commands. Users who are familiar with the Forth programming language will recognize many similarities.

This section describes existing IBCL commands and techniques for adding new ones to the system. For tutorial-style information, consult one of the Forth language books listed below:

Forth, An Applications Approach by David L. Toppen, McGraw-Hill.
Forth Programming by Leo J. Scanlon, Howard W. Sams Publications.
Mastering Forth by Anita Anderson **&** Martin Tracy, Prentice Hall.
Starting Forth by Leo Brodie, Prentice Hall.
Thinking Forth by Leo Brodie, Prentice Hall (Advanced Techniques).

LANGUAGE **STRUCTURE**

An IBCL program is a list of numbers or one-word commands. A word is an unbroken string composed of up to 63 characters. The IBCL standard word set includes the following characters:

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ a b c d e f g h i j k l m n o p q r s t u v w x y z [  1
```

IBCL will define and recognize words composed of any sequence of 8-bit bytes. Space, carriage return, and linefeed serve as word delimiters. Also some ASCII codes may have a special meaning to your system. It is usually best to limit yourself to the standard character set, but you may use uppercase and the special graphics characters if you wish.

Learning IBCL is similar to adding a few hundred words to your vocabulary. The names of the words will often relate to English words that you already know. The definitions of these words are detailed and specific; they are neither ambiguous nor dependent on context, and can be learned a few at a time.

The definition of a new IBCL word is a list of previously defined IBCL words or a machine code primitive.

An IBCL program is executed by executing a sequence of words. If a word in the sequence is defined by a code primitive, that code is executed. When a word is defined by a list of other IBCL words, execution of the original list is suspended until the list from the definition is executed. When you run an IBCL program, each word in the sequence composing the program executes in turn.

This execution sequence is different from subroutine oriented languages. In a subroutine oriented language, you may still define a higher level subroutine as a list of lower level ones, but time is always wasted by returning to the high-level routine before proceeding to the next routine in the definition. The more efficient course is apparent:

```
subroutine compiler                 IBCL
--------------------                -----
s
u >>>>>>>>                          I >>>>>>>>
b        subroutine b               B        IBCL-word-b
r <<<<<<<<                          C        v
o                                   L        v
u >>>>>>>>                                   v
t        subroutine c               w        IBCL-word-c
i <<<<<<<<                          o        v
n                                   r        v
e >>>>>>>>                          d        v
/        subroutine d                        IBCL-word-d
a <<<<<<<<                          a <<<<<<<<
```

Stacks

IBCL keeps the data it is using on a stack. IBCL words generally take their input parameters from this stack and leave their results on it. The most fundamental IBCL words are defined in machine code and perform the following functions.

- Place an address on the stack.
- Replace an address on the stack with the contents of that address.
- Replace the top element(s) on the stack with the result of some math or logic operation using them.
- Place a copy of some stack element on top of the stack.
- Rearrange the top few elements of the stack.
- Delete element(s) from the top of the stack.

A stack may be viewed as a deck of cards lying face up with each card only partially covering the one below, as in some solitaire games. In this game, you have the ability to create a copy of any card you see in the deck and place it on top of the stack. You can also remove any card and place it on top, but this takes much longer. The top three cards are most easily copied or rearranged. A math or logic operator like max (maximum value) would take the top two cards from the stack, place the higher valued one back, and destroy the other. Instead of playing cards, we could have a stack of plain cards with numbers written on them and a supply of blank cards. Addition is defined as removing the top two cards, writing the sum of their numbers on a blank card, and placing the new card on the stack.

IBCL has another stack which usually takes care of itself. It is called the return stack. When a higher level word is executed, each lower level word in its definition is executed. But each of these words may also be defined in terms of yet lower level words, until the lowest level words defined in machine code are reached. As IBCL descends through each level of a definition, it leaves the address of the next word of the current level on the return stack. When the lower level is completed, this address is removed from the return stack and execution proceeds from that point.

Sometimes it is convenient to temporarily move some word from the data stack to the return stack, usually to perform some operation using words lower on the data stack. Because of the definition interpreter's use of the return stack, that word should only be used in the definition placing it on the return stack. It must be removed from the return stack before execution of the definition terminates.

Dictionary

IBCL 'remembers' word definitions using a data structure called the dictionary. When you define a new word, IBCL adds a dictionary entry for that word. The only words IBCL understands which are not in the dictionary are numbers.

IBCL separates the dictionary entry for each word into three parts in three different segments (a segment is a contiguous block of MacBus memory). The word's ASCII name string is stored in a vocabulary segment. Its definition is stored in the definition list segment. The first byte-pair of the list contains the address in the code segment of the code primitive that begins execution of the word. (To avoid confusion with IBCL "words", byte-pair will be used instead of word when referring to a 16-bit memory location.)

Since each aspect of the definition naturally defaults to its own segment register, the partition generates no memory or time overhead loss. In addition, no more memory pointers are needed than in a conventional Forth system. The pointer into the definition list segment replaces the pointer to the previous word found in conventional systems. The pointer to the previous word is not needed since only word names occupy the vocabulary segments and since each vocabulary has its own segment.

Vocabularies

The vocabulary entry has three parts. The first byte-pair contains the address in the definition list segment of the beginning of the word's definition. The following bytes contain the ASCII name string. The last byte contains the number of bytes needed for the string. Its high-order bit is set to make it easier to find during the forward searches required by **decompilers.** The second to high-order bit is set if the word is defined as immediate. When a word is encountered in a definition, the address of its definition list (from the first byte-pair of the vocabulary entry) is usually entered into the new definition. If the word is immediate, it is executed instead.

IBCL allows many separate vocabularies. This allows you to separate definitions into well-organized groups, much like you would place related C functions in a single file. IBCL can find words faster when it only has to search a couple of vocabularies instead of the entire dictionary. IBCL provides a separate segment for each vocabulary defined.

The two vocabularies, 'context' and 'current', are always singled out for special treatment. The 'context' vocabulary is searched first for words encountered in the input stream. If the word is not found, the root directory (named **IBCL)** is searched. The 'current' vocabulary is the vocabulary to which new definitions are added. The variables 'context' and 'current' contain pointers to these two vocabularies.

Definitions

The actual definition of an IBCL word consists of two parts: the **code field** and the **parameter field.** The code field is a single byte-pair designating the address in the code segment of the machine code primitive associated with the word. The parameter field immediately follows the code field. Its use varies from word to word; a pointer to the parameter field is passed to the machine code primitive, which may use the parameter field in any way it chooses.

The simplest IBCL words are written in machine code. These words have no parameter field. IBCL executes the code addressed by the pointer stored in the parameter field. These words execute very fast.

High level definitions are more complicated. The parameter field of a high-level definition contains a list of addresses. Each address points to the code field of one of the lower level words making up the high-level word. The code field of a high-level word points to code that saves IBCL's 'program counter' on the return stack and replaces its value with the address of the high-level word's parameter field (remember, the parameter field address is made available to the machine code primitive for any purpose; in this case it contains the list of addresses composing the word's definition.).

The last address in the list making up a high-level definition will point to the code field of a word that removes the address from the return stack and restores IBCL's 'program counter' to this value, thus returning control to the next level up.

Other commonly used IBCL word types are constants and variables. The code field of a constant points to a routine that places the **contents** of the first byte-pair of the parameter field on IBCL's data stack. The code field of a variable points to a routine that places the **address** of the first byte-pair of the parameter field on the data stack.

Postfix Notation

IBCL uses postfix notation syntax. In postfix, one writes the stack numbers and then the operators. Numbers are pushed onto a stack and taken from it. For instance:

```
        7  212  3  / * -
```

First 7, and then 2, 12, and 3 are pushed onto the stack. Then 12 is divided by 3 and the result is placed on the stack. Now we have 7, 2 and 4 on the stack and the next operator is *****, leaving 7 and 8 on the stack. The **-** leaves -1 on the stack.

Note that it is not the syntax of IBCL that gives rise to this appearance of postfix notation, but the semantics of those words used to implement mathematical operations most efficiently. Other forms of notation can be developed on this system if desired.

Command Line Input

IBCL executes the word **query** when it has exhausted its ASCII input and needs some more. This word sets status that allows the MacBus' host system to learn of the situation and download new input. The exact method used by the host to accomplish this varies depending on the host software package you have; see SECTION ONE - IBCL Source Files for details.

THE IBCL INTERPRETERS

IBCL has two interpreters, the inner interpreter and the outer interpreter. The inner interpreter does nothing except branch from one machine code routine to the next. The nesting and unnesting routines supporting high-level IBCL definitions are among the code routines through which execution passes.

The outer interpreter accepts text from the host. It then attempts to parse the text string as a sequence of IBCL words and/or numbers. In execute mode, words are executed and numbers are placed on the stack. In compile mode, words and numbers are entered into the definition of the new word.

Inner Interpreter Sequence

IBCL uses the si register as its 'program counter'. si always points to the next entry in a word's definition list to be executed. **IBCL's** inner interpreter is the simple four-byte machine code sequence:

```
lodsw          ;move code field address of next word
               ;into ax register
xchg di, ax    ;move next word cfa into di
jmp  [di]      ;execute code pointed to by cfa
```

Note that [di+2] will contain the address of the parameter field, which the code executed may use in any way it chooses.

If the definition list which the inner interpreter is interpreting consists of a list of pointers to simple machine code primitive instructions such as stack and math words, execution will simply proceed from one word to the next in the list. A few special machine code primitives alter this orderly flow.

One of these diverting primitives is **<docol>**. **<docol>** is the primitive that nests control to a lower level definition as discussed earlier.

exit is the last pointer in a definition list. Its machine code primitive pops the top element from the return stack and continues list interpretation at that address. This is the word from which control was originally diverted.

There are several other words which alter the sequential interpretation of a definition list. **<.">** and <abort"> are compiled by the immediate words, **."** and **abort". They** control display of the following **inline** string and cause interpretation to skip to the word following that string. **lit** and **dlit** cause the following word or double word value to be pushed onto the stack; interpretation continues after the last value.

execute causes a branch to the word pointed at by the top value on the stack, just as if the pointer to that words code field address had been in the list instead of **execute.**

The remaining control-flow altering words handle the high-level flow control within a single definition list. **branch** causes control to skip forward or backward the number of words contained in the following location. **Obranch** does so only if the top word on the data stack is zero. Otherwise control continues with the word following the unneeded relative offset.

The do loop terminating words are similar in function and appearance to **Obranch.** First they perform the additional task of updating an index and comparing it to a limit. If the limit has exceeded bounds, control is transferred as with branch. If the bound has not been exceeded interpretation continues after the relative offset.

Outer Interpreter Sequence

Text is accepted a line at a time from the host. A line can be up to 1024 bytes long. The interpreter further breaks each line or block into individual words and processes them one at a time in sequence. A word is a string of characters preceded and followed by blank spaces or by a null. A few words require text strings as following arguments and use a special delimiter such as quote to end the string. Within these strings, blanks are not interpreted as word separators. These strings are processed by the preceding word rather than by the interpreter.

One such special string is the comment which opens with (. The interpreter ignores input after the (word until the next) or until the end of the current line or block. The initial (is a true IBCL word but the closing) is only a delimiter and need not be preceded by a blank.

Once a word is extracted, an attempt is made to locate it in the dictionary. If it is found, its code field address is returned. In execution mode the definition beginning at this address is executed, but when compiling a higher level word the address is appended to the definition being created unless it is an ‘immediate’ word. These execute even within a colon definition.

The IBCL word **immediate** makes the most recently compiled word immediate.

If the word was not located, the interpreter assumes that it is a number and attempts to convert it to binary form. The value stored in **base** identifies the current number system (ten,decimal;eight,octal;sixteen,hexadecimal). The number may begin with a minus sign. If it contains a decimal point, it is converted as a double length number, otherwise it must fit in a single byte-pair. When a single byte-pair number is too large, high-order bits are lost. A single byte-pair number may be as small as -32768 or as large as 65535, a double byte-pair number may be as small as -2147483648 or as large as 4294967295. The upper half of the positive range is for unsigned numbers only. Double byte-pair numbers cannot overflow, but the correct decimal point location must be determined from the user variable **dpl.**

The decimal point in the double numbers identifies them as double numbers but does not affect the binary value generated. The two numbers 123. and 1.23 produce the same binary value. The location of the decimal point is available in the user variable dpl, which is 0 and 2 for the above numbers. **dpl** is used by the application to scale numbers according to the location of the decimal point.

In execution mode, the binary value is placed on the stack. For single byte-pair numbers in compilation mode, the code field address of lit (literal) is appended to the definition followed by the binary value. For double byte-pair numbers in compilation mode, the code field address of **dlit** (double literal) is appended to the definition followed by the low-order byte-pair of the double number and then the high-order byte-pair.

If the string cannot be converted, the interpreter aborts with an error message. The stacks are cleared and the rest of the block or line being interpreted is ignored.

The interpreter uses **-find (see** DEFINING NEW WORDS paragraph) to locate the potential word in the dictionary. Since the source string for **-find** is the next word in the input stream, this also advances the interpreter over the input text.

If the string is not a word, **number** is used to convert it to binary form. **number** is vectored through **'number to <number> so you** may change the operation of **number** by storing the code field address of your numeric conversion routine in 'number. <number> expects the address of the source string's count byte on the stack. It replaces the address with the double word binary value converted using the current **base.** If conversion is not possible, <number> aborts with a "NOT RECOGNIZED" error. The user variable dpl will contain 8000H if no decimal point was present in the numeric string. In this case the number was single word length and the top word on the stack may be dropped. Other values in **dpl represent** the power of 10 by which the double word integer returned must be multiplied in order to recover the original decimal value. The interpreter ignores **dpl except as a flag to drop** the top word of single word entries.

When all words in the input stream have been executed, query is used to obtain more input and the entire cycle repeats.

Errors

When an error is encountered during interpretation, an error message is usually generated using **abort".** Execution of the run time portion of this word, **<abort">,** clears the stacks and prints an error message. Control is then returned to the terminal to await the next line of input. See DEFINING NEW WORDS, COLON DEFINITIONS, and IBCL INPUT paragraphs.

abort is vectored through **'abort to <abort>. You** may alter the operation of **abort** by storing the code field address of your abort word in the variable **'abort.** It generates no error messages but does clear the stack, resets both **context** and current vocabularies to IBCL, and returns control to the terminal.

Storing 0 in the user variable warning will suppress non-fatal system error messages. The only non-fatal error in the base system is the "ISN'T UNIQUE" message generated for attempts to redefine a word that is already in the **current vocabulary or in** the IBCL vocabulary. The word will be redefined, and access to the old definition is lost. Previously defined words that used the old definition are not affected.

STACK MANIPULATION

IBCL uses two stacks: the parameter or data stack and the return stack. The parameter or data stack is used to pass information from one word to the next. It is often referred to as “the stack”. The return stack is used by the interpreter to find its way back up through nested sequences of words being executed. It is always called “the return stack”. Occasionally the return stack is used within a word as temporary storage. Any temporary items on the return stack must be removed before the word completes execution. The return stack also holds the index and limit for do loops within colon definition words. These are automatically removed when the loop terminates.

Parameter stack words rearrange, drop, and duplicate words on the parameter stack. Note that “words on the parameter stack” refer to 16-bit numbers, not to IBCL definitions.

The parameter stack grows downward from an offset within the stack segment. The parameter stack pointer occupies the sp register. The return stack grows upward from the bottom of the stack segment. The return stack stack pointer occupies the bp register. Both reside in the stack segment and use the stack segment register.

Any words which refer to the state of the stack refer to the state that existed before the word was executed.

Parameter Stack

?dup	Duplicate top word on stack if it is non-zero.
depth	Return number of words on stack before operation.
drop	Drop top word from stack.
dup	Duplicate top word on stack.
over	Duplicate second from top word on stack.
n pick	Duplicate n'th word on stack. 1 pick is equivalent to dup 2 pick is equivalent to over
n roll	Remove n'th word from stack, leaving it on top. 2 roll is equivalent to **swap** 3 roll is equivalent to rot This is a time consuming operation since all words on stack from top through n'th must be moved.
rot	Remove third word from stack, leaving it on top.
swap	Remove second word from stack, leaving it on top.
sp@	Return offset into stack segment of top of stack. (Top of stack has the lowest physical address.) (Return stack pointer, stored in the sp register)
sp!	Initialize stack pointer to **s0,** clearing stack.
sp0	Return the address within the stack segment descriptor pointing to the bottom of the stack.
so	Return the address of the bottom of the stack. When the stack pointer has this value, the stack is empty.
ddrop	Drop the top two words from the stack.
ddup	Duplicate the top pair of words on the stack. Produces same result as **over over.**

The following external words are from the Forth 79 & 83 Standard double number word set, with the prefix character changed from 2 to d. They correspond to their single word counterparts, but operate on pairs of words.

High level implementation.

```
: dover        4 pick      4 pick ;
: drot         6 roll      6 roll ;
: dswap        4 roll      4 roll ;
```

Code level implementation (see USING ASSEMBLY LANGUAGE FROM IBCL paragraph).

```
code dover          bx push.                                 bp sp xchg.
                    bx 4 +[bp] mov. ax 6 +[bp] mov.          sp bp xchg.
                    ax push.                                 end- code
code drot           2 +[bp] pop.
                    dx pop.             cx pop.
                    ax pop.             di pop.
                    cx push.            dx push.
                    2 +[bp] push.       bx push.
                    di push.            bx ax xchg.          end-code
code dswap          cx pop.             ax pop.              dx pop.
cx push. bx push.   dx push.            bx ax xchg. end-code
```

Return Stack

>r	Transfer top word from data stack to return stack.
+>r	Add top word on data stack onto top word of return stack.
dup>r	Copy top word from data stack to return stack.
rdrop	Drop top word from return stack.
r@	Copy top word from return stack to data stack.
r>	Transfer top word from return stack to data stack.
rp@	Return value of return stack pointer (bp register).
rp!	Clear the return stack.

ARITHMETIC & LOGIC

The arithmetic and logic words find all of their inputs on the data stack, remove the inputs from the stack, and return their results on the stack.

The ranges for the supported number types are:

integer type	decimal range		hexadecimal range	
signed	-32,768	32,767	-8000	7FFF
unsigned	0	65,535	0	FFFF
double	-2,147,483,648	2,147,483,647	-8000.0000	7FFF.FFFF

logical 0 for false; -1 generated, non-zero accepted for true

Some people don't like using -1 for true, they would rather use 1 to be able to use it as a counter. Negative one can be used as a counter and the eventual sum negated with negligible time loss. It also has compelling advantages, too numerous to cover completely.

Suppose we want to find the sum of a subset of a series of numbers. A parallel series contains true flags (-1) for those elements we want to include. The numbers are constants ending with **n,** and the flags are variables ending with **f.**

```
an af @ and   bn bf @ and + cn cf @ and +  etc.
```

If truth had been 1 instead we could have resorted to a complex sequence of **if** statements within colon definitions or could have used:

```
an af @  *   bn  bf  @ * + cn  cf  @  *  +  etc.
```

Of course, * requires five times the execution time that **and** uses. This type of summation is important in math, logic, electronics, and even games.

Error conditions other than divide overflow are not detected, although the flags from the operation can be returned on the stack by the **flags>** word. It should be used immediately after the arithmetic operator. Remember that for complex operators, the flags are from the last operator invoked. The corresponding assembler operation (see USING ASSEMBLY LANGUAGE FROM IBCL paragraph) will indicate which flags are set. Errors may also be detected by checking whether the result has an obviously incorrect sign.

The divide error uses interrupt vector number zero. If you wish to replace this routine, the segment paragraph number at OOOO:OOOO and the offset at 0000:0002 must be set for your routine. The error will trap attempts to divide by zero and divisions that produce a quotient larger than the range for the divisor type.

When an arithmetic operation results in a number that is too large, positive or negative, the high-order bits are truncated. The result returned is usually very different from the desired result and often doesn't even have the correct sign. Adding one to 32767 gives -32768.

Division follows the conventions of Forth-79. The remainder has the same sign as the dividend and the quotient is rounded toward zero. Note that this is the same type of division as performed by the 8088 division instruction, making it the most efficient division for this processor. The Forth-83 standard requires the remainder to have the same sign as the divisor and the quotient to be rounded downward. The extra overhead to convert to this form should be incurred only if it is genuinely essential. If the remainder will be used to round the quotient, the form is unimportant.

Constants

A few small integers are used so frequently that it is worthwhile to implement them as constants. When the interpreter encounters them, they are located in the dictionary rather than being parsed by **number.** More important, when used in definitions, they result in compilation of a single word rather than the **lit** and value pair of words produced by other integers. When executed, a constant pushes its value onto the top of the stack.

The IBCL constants are: -2 -1 0 1 2 3 4

Unary Operators

These words alter the top word length integer on the stack. Most operate on either signed or unsigned integers. The few exceptions (**0<** **0>** abs neg **b->w**) must obviously deal with signed quantities. The divide by 2 operator, **2/**, also applies to signed quantities, since division requires the sign bit to propagate downward as the number occupies fewer significant bits.

IBCL maintains the top word of the stack in the **bx** register, so no memory accesses are required for these operators, not even a push or pop. **bx** will be used as a shorthand notation for the top word on the stack.

2-	Subtract 2 from bx.
1-	Subtract 1 from bx.
1+	Add 1 to bx.
2+	Add 2 to bx.
2*	Multiply bx by 2. (Shift bx left one bit.)
2/	Divide bx by 2. (Shift bx arithmetic right one bit, if bx is negative, increment by one.)
4+	Add 4 to bx.
6+	Add 6 to bx.
0<	If bx is less than 0, replace with -1, else with 0.
o=	If bx equals 0, replace with -1, else with 0.
0>	If bx is greater than 0, replace with -1, else with 0.
abs	Replace bx with its absolute value.
negate	Replace bx with its 2's complement (negate bx).
b->w	Propagate the high-order bit of the low-order byte

	in bx through the upper byte (convert signed byte to signed word).
s->d	Convert signed word length integer to signed double word integer. The former top word is pushed one deeper into the stack. The new bx is all one bits if the old bx was negative, all zero bits otherwise.

The following unary operators replace the top two stack words. The high-order word is at the top of the stack in the bx register, the low-order word is below it.

dabs	Replace double word integer with its absolute value.
dnegate	Replace double word integer with its 2's complement.
udsqrt	Replace unsigned double word integer with unsigned singleword square root.

Binary and Ternary Operators

Binary integer operators remove the top two words from the stack and replace them with the result of the operation, usually a single word. Ternary integer operators remove the top three words from the stack and replace them with one or two results.

Mixed word length operators have one operand that is a double word. For double word length operators, both operands are double words. A double length word occupies two words on the stack. The high order half is toward the top of the stack with the low-order half under it. Mixed operators generally begin with **m,** double with **d.** To run some Forth-79 standard programs, it may be necessary to **redefine** the **d** prefix words as 2 prefix words, or change 2 prefix words in the program to d prefix words.

IBCL maintains the top word of the stack in the **bx** register and the second stack word at the top of the physical stack. A single pop is necessary to bring the second operand into the registers, assuring maximum efficiency. **bx** will be used as a shorthand notation for the top word on the stack, and nx for the next word on the stack. The third word, when required, will be called **tx** and a fourth **qx.** Double length words require two of these codes separated by a period.

All input words are removed from the stack and the result becomes the new **bx.** If more than one word is returned, a new nx is created as well.

Signed or Unsigned Operands

+	Add bx to nx
	Subtract bx from nx
=	-1 if nx and bx were identical, 0 otherwise
d+	Return sum of double integers bx.nx and tx.qx
d+-	Return nx.tx, negated if bx was negative.

Signed Operands

+-	Return nx, negated if bx was negative.
*****	Multiply nx by bx. If product is too large, high-order bits are lost.
/	Divide nx by bx, rounding quotient toward zero.
/mod	Like / but the remainder is returned in nx with the same sign as the dividend.
mod	**Return** the remainder of the division of nx by bx.
***/**	Multiply nx by tx and divide the 32-bit product by bx, rounding toward zero. If result exceeds range (-32768 through **32767),** a 0 divide error occurs.
***/moD**	Like ***/** but the remainder is returned in nx with the same sign as the intermediate product.
<	-1 if nx is less than bx, 0 otherwise

>	-1 if nx is greater than bx, 0 otherwise
max	Return the greater of nx and bx.
min	Return the lesser of nx and bx.

Mixed Length Signed Operands

m+	Add double integer nx.tx to bx, return in bx.nx.
m*	Multiply bx by nx, return double integer bx.nx.
m/	Divide double integer nx.tx by bx, rounding toward zero. Return remainder in nx, quotient in bx.
m*/	Multiply double integer tx.qx by nx and divide the 48-bit product by bx. Round double integer quotient bx.nx toward zero.
m/mod	Like **m/** but return double word quotient bx.nx and remainder tx.
d<	bx becomes -1 if tx.qx is less then bx.nx, 0 otherwise

Unsigned Operands

u*	Multiply nx by bx, returning double word product. Low order word in nx, high-order word in bx.
u/mod	Divide double integer nx.tx by bx. Low order word of dividend is tx, high-order word is nx. Return remainder in nx, quotient in bx.
u<	**-1** if nx is below bx, 0 otherwise Below is the unsigned counterpart of less than.

Logical, Sign Bit Not Significant

and	**Bitwise** logical conjunction of bx and nx. For each bit position, 1 if corresponding bits in both

	nx and bx are one, 0 otherwise.
not	-1 if bx is zero, 0 otherwise
or	Bitwise logical inclusive disjunction of bx and nx. For each bit position, 1 if corresponding bits in either nx or bx are one, 0 otherwise.
xor	Bitwise logical exclusive disjunction of bx and nx. For each bit position, 1 if a single corresponding bit in nx or bx is one, 0 otherwise.
slr	Shift nx right by bx bits, shifting in zeroes.

External Double Number Words

d-	Double negative of top double word.
d0=	True if top double word is zero.
d2/	Arithmetic right shift top double word.
d=	True if top two double words are equal.
d>	True if bottom element of double word pair is greater than top.
dmax	Return larger of signed double word pair.
dmin	Return smaller of signed double word pair.
du<	True if bottom element of double word pair is below the top element, unsigned.

```
code d-     bx ax xchg.    dx pop.  cx pop.  bx pop.
            cx ax sub.  bx dx sbb.  cx push.                end- code

code dO=    ax pop.  bx ax or.  bx 0 iw mov.
            jnz false. bx dec. >>> false.                   end- code

code d2/    ax pop.  bx 1 sar.  ax 1 rcr.  ax push.         end-code
```

MEMORY ACCESS

These words store values into memory or retrieve them from memory using an address and possibly a segment paragraph number from the data stack. A single byte, word, or double word may be stored or returned, or an entire block of bytes may be filled, anywhere in memory. Any contiguous block less than 64K bytes may be moved anywhere in memory.

Load and Store

The root of these words is **@** (at) for load and **!** for store. A word with the root **@** requires an address from the stack. A word with the root **!** takes two parameters from the stack, an address from the top of the stack and a number under the address. Long addresses include a segment paragraph number beneath the offset. Double word numbers store the most significant portion toward toward the top, just below the address word or words.

A word with the root **@** replaces an address on the stack with the value stored at that address.

A word with the root **!** stores the number from the stack under the address into the location at that address.

nnnn **@**	Returns the word from the memory location with offset nnnn in the definition list segment.
number nnnn **!**	Stores number into word at the memory location with offset nnnn in the definition list segment.

Several of the addressing modes use the segment paragraph number found in es, the extra segment register. This register is not used by the IBCL stack, arithmetic, logic, load, store, port I/O, or control words. It may be set with es!

```
new-segment-paragraph-nunber es!
```

NOTE: The **xx@e** and xx!e addressing modes may only be used within colon definitions since the interpreter uses the es register for dictionary searches. Within colon definitions (Defining New Words paragraph) the es segment register retains its contents throughout execution of words from the sections listed above. This extra segment mode is almost as **efficient** as the default definition list segment addressing mode. It may be the only addressing mode required when working with an array or arrays in a single foreign segment. If two arrays from different segments must be used, this mode should be used for the one accessed most frequently and the long address mode should be used for the other. In the long address mode, the address occupies two words. The lower word on the stack is the segment paragraph number, the top word on the stack is the offset into that segment.

Addressing modes are identified by a **suffix** appended **@** or **!**

suffix	segment paragraph address
none	Definition list segment, ds register.
E	Extra segment paragraph number, es register.
L	Address occupies two words on stack, the segment paragraph number is under the offset.

The data type is identified by a prefix preceding **@** or **!**

prefix	data type
none	Word, two bytes.
c	Character or byte, lower byte of word on stack upper byte set to zero for **@** and ignored for !.
d	Double precision, two words Low order word is the first word in memory, and is followed by the high-order word. For small positive numbers the high-order word is zero, for small negative numbers the high order word is all ones (-1). The low-order word is

lowest on stack, with the high-order word toward top.

The indexed addressing mode uses load and store words generated above prefixed by **i.** Before the load or store takes place, the top two numbers on the stack are added. This allows byte offsets into arrays without the linkage overhead of the otherwise equivalent + **!** or + **@** forms.

The load and store words covered so far are:

@	@E	@L	C@	C@E	C@L	D@	D@E	D@L
I@	I@E	I@L	IC@	IC@E	IC@L	ID@	ID@E	ID@L
!	!E	!L	C!	C!E	C!L	D!	D!E	D!L
I!	I!E	I!L	IC!	IC!E	IC!L	ID!	ID!E	ID!L

All load and store words are machine code primitives implemented with code sharing. The many addressing modes occupy little more code than the basic modes. This is also true of the hybrid access words below. The address plus index should not exceed 64K. An offset exceeding 64K wraps back to the beginning of the current segment.

The +! words resemble **!** in use but instead of replacing the word length number located at the memory address, the number on the stack is added into it. All addressing modes are included, but only for word length operands.

+! I+! I+!E I+!L

Often counters must be incremented or decremented by one or two. These may almost always be kept in the definition list segment and are usually word length.

1+! 1-! 2+! 2-!

The most common number which memory must be set or reset to is zero. The value for the following stores need not be fetched from the stack since it is zero.

O! O!E OC!E

The following words are used in the implementation of IBCL and have many general uses.

dup@	Duplicate the address before returning contents. The address is often needed later.
dupc@	Like **dup@** but for byte access.
step	Increment the address by two, then do **dup@**. Used for stepping through word arrays.
cstep	Increment the address by one, then do **dupc@**. Used for stepping through byte arrays.

Fill

These words fill a block of memory with copies of a single byte length number. The destination paragraph number is stored in the es register and remains there. Nothing is returned on the stack. The address plus count ordinarily should not exceed 64K. Offsets past 64K do not extend into the next segment, but back to the beginning of the first one.

```
addr n byte fill
```

Fill n consecutive memory bytes beginning at addr in the definition list segment with the byte. If the count n is negative, no action is taken. Use of the machine code primitive **<fill>** circumvents the sign check.

```
seg-para-# addr n byte filll
```

This is the long address version of **fill** used to fill a block anywhere in memory. The count is unsigned and may range up to 64K.

```
        addr n blank
```

This behaves like **fill,** but the byte stored is hex 20 (blank).

Move

These words copy a block of memory to a new, possibly overlapping block. The source uses the **ds** segment register, the destination uses the es register. The **ds** register is restored to its original value after the operation, but the es register retains the segment paragraph number of the destination. Nothing is returned on the stack.

Neither the source address plus count nor the destination address plus count should exceed 64K. Offsets past 64K do not extend into the next segment, but back to the beginning of the original segment.

```
        source-addr dest-addr n cmove
```

This moves a block of memory n bytes long beginning at the source address to the block at the destination address. Both blocks are in the definition list segment, **lists.** If n is not positive, no action takes place. The sign check is circumvented by using the code primitive for **cmove, <cmove>**. The lowest addressed bytes are moved first.

The two blocks may overlap if the destination is below the source. If the two blocks overlap and the destination is above the source the copy will proceed smoothly until the source address equals the original destination address. At that point, the original data has been overwritten and the sequence of bytes copied to that point will repeat throughout the remainder of the copy.

The long address form of **cmove** is:

```
        src-para# src-adr  dest-para# dest-adr n cmovel
```

This moves any block from any segment to the same segment or to any other segment. The count is unsigned and may range up to

64K. The lowest addressed byte is moved first. **cmovel** behaves like cmove for overlapping blocks where the destination is at a higher address.

The overlap problem can be avoided by using **<cmovel**, which moves the highest addressed byte first. It is slightly less efficient than **cmovel** and should only be used for overlapping blocks where the destination is an address below the source (unless you are trying to produce a downward ripple):

```
        src-para# src-adr dest-para# dest-adr n <cmovel
```

The above words all move byte oriented blocks. **move** is similar to **cmove** except that it moves blocks of words.

```
        src-adr dest-adr n move
```

For any copy to take place, n must be between 1 and 3FFF. The copy is performed by **cmove** after doubling the count, n.

move is included for compatibility, but its function may be performed equally well by the various cmove instructions which also access segments other than the definition list segment.

String Functions

A string is any sequence of bytes where the first byte contains the number of bytes in the sequence, excluding that first byte. The string support words are loaded from a utility file.

See SECTION ONE - IBCL Source Files for methods of inputting utility files to IBCL.

Strings will frequently reside in the **pad** buffer, which serves as temporary storage for terminal input and string functions. The **pad** buffer is at a fixed offset above the allotted portion of the lists segment. When more space is allotted, the buffer is moved and the old contents are left behind. The **pad** is for temporary storage only.

Space for more permanent strings is allotted using **$variable** or **$constant. The** run time code for both is the same as for variable, it simply returns the address of the first byte. To reserve n bytes plus a count byte for a string, type:

```
        n $variable  string-variable-name
```

$constant can be used to initialize a string variable as it is created:

```
        $constant s-v-name the_initial_sequence"
```

If you have created an extra segment to hold arrays, you can also allot strings in that segment:

```
        segment-name #bytes $var-far string-variable-name
```

Execution of this string variable name will push the segment paragraph number and the offset onto the stack.

Strings are moved among string variables and the pad with store words:

```
                   src-addr                dst-addr    $!
        src-seg    src-addr     dst-seg    dst-addr    $!l
        lists @    pad          dst-seg    dst-addr    $!l
```

It is also possible to exchange the contents of two string locations — but only if both locations are large enough to hold the longer of the two strings.

```
                 addr-a             addr-b    Sxchg
          seg-a  addr-a      seg-b  addr-b    Sxchgl
                sv-name                pad    Sxchg
            sv-far-name       lists @  pad    $xchgl
```

The following words all create a new string in the pad buffer and leave the address of pad on the stack.

```
              addr-a               addr-b  $+        (concatenate strings)
        seg-a addr-a     seg-b     addr-b  $+l
```

```
          addr   n      left$      (n characters at left)
          addr   n      right$     (n characters at right)
   addr   start  n      mid$       (n char beginning at start)
                 byte   chr$       (1 byte string of byte)
          addr   n      string$    (n byte string of first)
                                   (char from string at addr)
```

asc is the inverse of **chr$**. It returns the first byte after the count byte on the stack.

```
addr asc
```

$find will find the offset of characters from a string in any segment. The template string must be in the **lists** segment.

```
ts-addr segment start-addr stop-addr $find
```

Three values are returned on the stack. The segment, the address of the first instance of the string in the range, and a flag set to -1 if the string was found. If the string was not found the flag is set to 0 and the address is set to the start address.

cr$ and **eof$** are **predefined** one byte string variables used to find carriage returns and end of **file** bytes.

$= compares two byte sequences of equal length. A zero is returned if the sequences are identical. Otherwise, the absolute value of the number returned is the number of the first mismatched byte pair. It is negative if the byte from sequence 2 is larger than the byte from sequence 1 and positive otherwise.

```
seg1 addr1 seg2 addr2 length $=
```

String Literals

The string literal is so useful that it is included in the base system. It can be used in a colon definition or to initialize a string array. In the first case it compiles a word that, when executed, leaves the address of the string literal on the stack and jumps past the string and then compiles the count byte and characters of the string. In the second case, only the count byte and characters are compiled. The following examples illustrate the two uses.

```
        : message    ," hi there! " type ;

or      15 $variable your-name      $"Alex" your-name $!
        : prefix    ," Your name is " your-name $+ count type ;

or      create item 0 c, ,"paper","scissors","stone"  oklw
        :.choice ( addr i -) 0 do count + loop count type ;
        item 2 .choice
```

Do you see how the loop indexes through the string array?

IBCL INPUT

Just as IBCL has separate words to manage ASCII and binary output, so it has separate words to manage ASCII and binary input. This section documents these words.

ASCII-Type Input

Whenever IBCL exhausts its ASCII input stream, it executes the word **expect.** This word takes an address and count from the stack and waits for the Macintosh Plus to initiate an ASCII data transfer over the SCSI bus. For example, the following IBCL fragment will create a buffer and **fill** it **five** times with ASCII data from the Macintosh Plus:

```
        variable string-buffer  3E allot

        : 5-fills
        5 o d o
        string-buffer 40 expect
        loop
```

The IBCL phrase variable **string-buffer** allocates memory for a two-byte IBCL integer variable. The phrase 3E **allot** adds an additional hex 3E bytes to the two already allocated, increasing the size of the buffer to hex 40 bytes. When later executed, the word **string-buffer** will leave the address of the 40 byte buffer on the stack.

The remainder of the fragment is a colan definition implementing an indexed loop. The word 5-tills will execute the IBCL sequence:

```
        string-buffer 40 expect
```

five times. Each time execution of **string-buffer** leaves the buffer address on the stack, execution of 40 leaves the buffer size on the stack, and execution of **expect** pauses until the Macintosh Plus transmits an ASCII data string over the SCSI bus. This string is placed **string-buffer.**

IBCL executes **expect** not only when the user explicitly uses it interactively or in a program, but also when the IBCL interpreter itself needs more ASCII input.

Obviously, expect requires cooperation from the Macintosh Plus. This can be accomplished in several ways, but one example may be taken from the Megamax C MacBus support library, available from National Instruments as a separate purchase. This library includes the following C functions:

```
        string-out (s, n, id) char *s; int n, id;
```

This function sends **n** bytes of ASCII data to MacBus. The data is taken from the character buffer **s,** and the string is sent to MacBus at SCSI address **id.**

```
        poll(id) int id;
```

This function reads a block of status data from MacBus at SCSI address **id.** By analyzing this status block, the Macintosh Plus can determine (among other things) when MacBus has exhausted its ASCII input stream.

The following C fragment waits for MacBus to be ready for ASCII input, then sends an IBCL command string:

```
/* wait for MacBus to exhaust its ASCII input */
poll(ID);
while ((fstat & DONE)==0)
poll(ID);

/* put the MacBus GPIB interface online and send
   Interface Clear */
string_out ("1 bonl bsic", 11, ID);
```

After executing this fragment, the Macintosh Plus is free to do other things while MacBus tends to the GPIB. This is just one example of how MacBus provides increased performance by allowing parallel processing in many situations.

Binary-Type Input

The IBCL word dlm allows the Macintosh Plus to transmit large arrays of binary data to MacBus. This word expects a count on top of the stack and a long buffer address under that. It waits for the Macintosh Plus to send the specified number of bytes over the SCSI bus and places the data at the specified long address. The following example illustrates operation of this word:

IBCL program:

```
ibfind plotter ( put GPIB plotter on line)
1000 alloc data_buffer( allocate a buffer for data)
data-buffer 1000 dlm( download data from Mac+)
plotter data-buffer 1000 wrt( write data to the plotter)
```

Macintosh C Program:

```
poll(ID);        /* wait for MacBus to be ready for data download */
while((fstat & DLM)==0)
                 poll(ID);
dlm(0x1000, buff, ID);/* download the data */
while((fstat & DONE)==0)/* wait for MacBus to finish */
                 poll(ID);
```

IBCL OUTPUT

During normal operation IBCL discards all of its terminal output. All information the host needs from IBCL is contained in a small status block. During debugging, however, you might like to see all IBCL output. If you execute the IBCL string:

```
        'find iaemit 'emit !'
```

IBCL begins to place its ASCII output in a buffer that a host routine allows you to retrieve. To turn output off, execute the IBCL string:

```
        'find drop 'emit !'
```

IBCL provides several words that transmit information to the Macintosh Plus. These words may be placed in one of two categories; ASCII-type output words and binary-type output words. This section documents IBCL's collection of output words.

ASCII-Type Output Words

Many different ASCII output words exist, but all of them work by calling the IBCL word **emit.** This word outputs a single ASCII character.

During normal operation you will have little use for IBCL's ASCII output. During debugging, however, you will find it indispensable. Therefore, IBCL is capable of "turning off" its ASCII output. The following IBCL command strings turn ASCII output on and off:

```
        find iaemit 'emit !   ( turn ASCII output ON)
        find drop   'emit !   ( turn ASCII output OFF)
```

MacBus powers up with ASCII output OFF. Whenever ASCII output is off, you don't need to worry about it at all; emit simply throws its characters away.

When ASCII output is on, **emit** places each character in a text output buffer in MacBus memory. At all times this buffer is null terminated, i.e., there is a zero byte following the last valid output byte. If the buffer fills up, IBCL will stop what it is doing and wait for the Macintosh to read the buffer.

Obviously, the Macintosh must pay attention to the output buffer whenever ASCII output is ON. If you are using the IBCL interactive window utility supplied with your MacBus, this is automatic. If, however, you write your own programs using, for instance, the Megamax C MacBus support library (available from National Instruments as a separate purchase), you Macintosh program must monitor the status of the output buffer anytime ASCII output is on. The following example shows how you might coordinate IBCL and Macintosh programs that use ASCII output.

IBCL Definition:

```
: letter-a's
        0 do
                ascii a emit
        loop
```

Megamax C Program:

```
/* send IBCL command string */
string_out("10 letter_a's", 13, ID);

/* loop till MacBus is finished with command */
poll(ID);
while((fstat & DONE)==0) {
        poll(ID);
        if (fstat & OUTPUT) {
                fout(s, ID);
                print ("%s", s);
        }
}
```

The C functions **string_out** and **poll** were described in the Terminal Input section. The function **fout** tills the string pointed to by s with the current contents of the IBCL output buffer. This is a normal C string, and can be printed using a printf statement, as in the example above.

This example repeatedly polls the MacBus. After each poll it checks for available ASCII output in the output buffer. If there is output in the buffer, the C program retrieves and prints it. This continues until the result of poll indicates that IBCL is waiting for more input (i.e., it has finished with the command.)

All other IBCL words that produce ASCII output invoke emit one or more times. The remainder of this sections describes these words.

Character-Based Words

space will emit one blank space. spaces will take the top number on the stack and emit that number of spaces. **bl** will leave the ASCII code for a space on the stack.

type uses the top number on the stack as a character count and the next number as a source address. Consecutive characters beginning at the source address are emitted until the count is satisfied. If the count is zero or negative no action takes place and the address and count remain on the stack. type only applies to strings in the definition list segment. The address is an offset into this segment.

type1 performs the function of type for strings in any segment. The segment paragraph number must be on the stack under the source offset

Two words are often used before type. count assumes the top number on the stack is the address of the count field of a string in the definition lists segment. It increments the address by one and returns it and then the count byte on the stack. -trailing expects the count byte on the stack with the address of the first character under it, in the form returned by count. Both address and count are returned on the stack, after the count has been reduced to discard any trailing blanks.

countl performs the function of count for strings in any segment. The segment paragraph number must be on the stack under the source offset.

Numeric-Based Words

The representation of a number depends on the base being used. There are 50 states in the United States if the base is decimal, but there are 32 states if the base is hexadecimal. A jigsaw puzzle of the United States could be divided into five piles of ten states each with none left over, or it could be divide into three piles of sixteen states each with two left over. The representation base is stored in the user variable base. base contains ten when in decimal mode and sixteen when in hexadecimal mode, but may be set to other values. **decimal** stores ten in **base** and hexadecimal stores sixteen in base. Octal could be set by:

```
8 base !
```

The following words output a number from the stack. The top stack word contains a field width for some of them. The individual digits are output by **emit.**

number		Display number with a single trailing blank and, if required, a leading negative sign.
double-number	d.	Like **.** except for double word length number. The high-order word is on top of the stack with the low-order word under it.
number	u.	Like **.** but the number is unsigned and the magnitude may therefore range from 0 through 65535 (decimal) or 0 through FFFF (hexadecimal)
number #char	.r	Display number right aligned in field #char characters wide. The sign is included only if it is negative. If #char is to small, no leading blank appears but the field is expanded to include all digits and sign.
double #char	d.r	Like .r but for double word length number.

number	u.r	Like .r but the number is unsigned.
source-address	**?**	Print the number stored at the address. :? **@. ;**

The only punctuation included in the above numbers is the leading minus sign. If more specific formatting is required, words are available to convert numbers one digit at a time. The following example will output the negative decimal single word number -12345 and insert a decimal point between the 3 and 4.

```
decimal -12345 dup s->d dabs <# # # 46 hold #s rot sign #> type
```

decimal -12345	
dup	Place two copies of -12345 on stack
s->d	Sign extend top copy to double length
dabs	Take absolute value of double number
<#	Initialize for output conversion
#	Place the lowest order digit (5) in buffer
#	Place second lowest order digit in buffer
46	ASCII code for decimal point
hold	Place decimal point in buffer
#s	Place remaining digits in buffer
rot	Rotate original signed number to top of stack
sign	Place sign of number in buffer
#>	Terminate output conversion and leave buffer address below number string length on stack
type	type number - 123.45

The **<# ...#>** construct converts an unsigned double length number to a string. The string is built rightmost character first and growing downward from the buffer address returned by **pad.** The opening **<#** stores this address in the user variable hld, which thereafter holds the address of the character most recently added to the string.

Each instance of **#** extracts the next higher order digit from the double number on the stack and adds it to the downward growing string. The unsigned double number is divided by the base. The double word quotient is left on the stack, eventually becoming zero. The remainder is converted to its ASCII code and added to the string. If # is used after all digits have been converted, leading zeroes will be added to the string.

will convert all remaining digits but stop before generating any leading zeroes.

Any character may be inserted anywhere in the string by placing its ASCII code on the stack and using **hold. hold** can be used to insert decimal points, commas, hyphens, slashes, etc.

```
$1,234,567.89    4-15-82      4/15/82       2:37:15
```

The following ASCII codes are in decimal:

	35 #	43 +	46 .	58 :
32 blank	36 $	44 ,	47 /	59 ;
	37 %	45 -		

If a sign is required, a number with the correct sign must be available. The double word number on the stack cannot be used since it must be converted to its absolute value. In the example, the signed number was kept on the stack under the double word unsigned number. This location is convenient but not necessary. The sign is usually added after converting all of the digits, placing it in the number string's first character position. It could just as easily be added to the string before converting any digits, placing it at the end of the number string as required by some financial formats. For instance: 123.45

The **#>** drops the double number from the stack. At this point, it should have been zero. The address of the first character in the string (from **hld)** is returned on the stack under the number of characters included in the string. This address and count are the arguments expected by **type,** which is used to output the string. Alternatively, the string could just as easily be moved to a file buffer.

Binary-Type Output

IBCL provides a single word to perform binary-type output. Binary output is ALWAYS turned “on”, and the Macintosh must always pay proper attention to MacBus when an IBCL program that does binary output is running.

IBCL’s binary output word is **ulm.** This word expects a count on top of the stack and a long address just below that. It waits for the Macintosh Plus to request a binary data upload, then sends the specified number of bytes over the SCSI bus, starting at the specified long address.

Just as the Megamax C MacBus Support Library provides the **fout** function to read ASCII output, so it provides the ulm function to read binary output from MacBus. Proper handling of binary output involves cooperative action by MacBus and the Macintosh Plus, as the following example shows.

IBCL program:

```
1000 alloc data-buffer        ( allocate a buffer for data)
data-buffer fill-up           ( we don’t care how buffer gets filled)
data-buffer 1000 ulm          ( upload buffer to the Macintosh)
```

Macintosh C program:

```
poll(ID);                  /* wait for MacBus to get ready for
                           memory upload */
while ((fstat & ULM)==0)
        poll (ID);

ulm(0x1000,buff, ID);      /* upload results from MacBus */
```

GENERAL PORT I/O

The port input/output words transfer data between the top of the stack and any of the 64K I/O ports. For specific port assignment, see the Technical Reference Manual for your system.

The following words assume that the port number is at the top of the stack:

```
pw@     ( place word from port on stack)
p@      ( place byte from port on stack,
        zero fill high-order byte of stack word)
```

The following words assume that the port number is at the top of the stack and that the output word (or byte) is the next word on the stack. For an output byte, the high-order byte of the stack word is ignored.

```
pw!     ( output word to port)
p!      ( output byte to port)
```

In practice, these words must often be used several times to input or output a single data item. Each interface may be composed of several ports. Both incoming and outgoing status ports may be needed in addition to the incoming and outgoing data ports. You will usually be required to write into the outgoing status ports before any data transfers may take place. This accomplishes such tasks as setting the baud rate, parity, and number of bits, depending on the type of interface. It also initializes the interface to a known state.

Next, an incoming status port is interrogated to see whether the interface is ready to receive or transmit data, depending on the pending operation. If you are ready to output data, and the interface is not ready to receive it you have two options. You may continue doing something else and interrogate the incoming status port later, or you may repeatedly interrogate the status port until the appropriate bit or bits indicates the interface is ready to accept outgoing data. Then you may output data to the outgoing data port. If you output data when the port is not ready, it will be lost and may interfere with the previous data output.

If you are trying to input data and the status port indicates none is ready, you will read garbage or reread the previous data item. Again, you may do something else for a while or continue reading incoming status until data is ready.

LOADING PROGRAMS

IBCL treats programs just like normal text input. The routines running on your host machine are responsible for breaking program files into pieces smaller than 1024 bytes and downloading the pieces to IBCL. See SECTION ONE - IBCL Source Files for details on downloading.

DEFINING NEW WORDS

This section is the heart of IBCL. By defining new words interactively with minimal overhead costs, IBCL surpasses both interpreted and compiled high-level languages. Since word definitions can be kept short without excessive overhead, they can be easier to write than the longer subroutines usually written in higher level languages. They are much easier to write or maintain than the endless nest of branches and **goto's** found in BASIC.

IBCL can define several kinds of words, and can even define words that define new types of words. This latter ability gives IBCL power that is as yet beyond our imaginations to fully tap. At the simplest level, it can provide direct language support for almost any data type or structure imaginable. The dot product of order n vectors can easily be reduced to **a-vector b-vector dot.** This is simultaneously both simpler and more efficient than BASIC with:

```
result=0 ; for i = 1 to n ; result=result+a(i)*b(I) ; next i
```

Even high-level languages with decent subroutine syntax quickly fill with distracting call's and parenthesis' that have nought to do with your algorithm, let alone your problem.

The primary word used to define all new words is **create,** as in:

```
create new-name
```

This enters **new-name** in the **context** vocabulary with a pointer to the next free memory word in the definition list segment. This word is initialized to point to the run time code for variables, but may be changed either by the defining word invoking **create** or by another word within or terminating the definition. **create** is used directly by the user only to define new defining words. It is used by all system defining words.

create will truncate names longer than the value contained in the user variable **width.** The initial value, also the maximum value, is decimal **63** characters. If truncation occurs, the system remembers only the shortened length.

Colon Definitions

These are the most pervasive definitions in IBCL. They resemble the subroutines or functions of other high-level languages such as Pascal or **Fortran** — with some important differences.

The syntax is not cluttered with parenthesis and parameter lists. This enables IBCL words to be used more nearly like the words in a human language — admittedly more like German than English since the action is specified after any values or addresses required.

Values and addresses are passed either on the data stack or through locations specified within the definition. Use of the data stack aids in the creation of more generally useful words.

The other crucial difference is that the definition is compiled when it is entered. No distracting, time consuming compile and link sequence is required.

In a very small way, BASIC shares this convenient lack of extra steps. BASIC may be used calculator style like IBCL, or it may be used to define the equivalent of a single IBCL word with the name RUN. In IBCL, one could type:

```
        : run   IBCL equivalent of BASIC program ;
```

In IBCL, of course, **run** could be named anything and you could have hundreds of programs at your fingertips simultaneously. No need for BASIC's incomprehensible tangle, single program limits, or incomparable slowness (of use mainly to hardware manufacturers trying to sell more expensive machines).

The basic format of the colon definition is:

```
: name-of-new-word    words comprising definition ;
```

The colon and name must be on the first line, but the remainder of the definition may occupy as many lines as required. Each word or number must be complete on a single line.

The definition of **:** is:

```
: :     sp@ csp !              (save current stack pointer)
                               ( in user variable csp)
        current @ context !
        create                 (create dictionary entry for name)
        ]                      (switch state to compile mode)
        ;' docol               (set code field to docol)
```

docol points to the machine code routine that nests the interpreter down one level and transfers control to the word after the code field containing that instance of the pointer to docol. state is a user variable which may be set to compile mode by **]** or to execute mode by **[**.

After **:** has initialized the definition and set compilation mode, the following words are compiled into the definition for execution when the defined word is executed. When a word is compiled, the address of its code field is appended to the list being created for the word being defined. If a number is encountered, the word lit (or dlit for double word numbers) is compiled into the definition followed by the number. Later execution of lit will cause the number to be placed on the stack and the interpreter will skip the location that held the number. The **;** terminates the definition by compiling an exit at its end and setting execution mode. exit will unwind the interpreter nesting one level, returning control to the word after the instance of the one that finished execution.

The following example will print the number followed by a % sign when the words name is entered: (37 is ASCII code of %)

```
decimal
: fifteen-percent 15 . 37 emit ; ok
fifteen-percent 15 % ok
hex ok
fifteen-percent E % ok
```

Numbers are interpreted using the current base. In the example above, the previous base was discarded in favor of decimal. Changing the base to hex changes the output representation of the number but not the ASCII character. The output of an ASCII character requires no numeric conversion. Note that typing:

```
: fifteen-percent decimal 15 . 37 emit ;
```

would not have changed the base until the definition executed. The 15 and 37 would be interpreted according to the previous base, and typing **fifteen-percent** would always change the base to hex.

The words **.** and **emit perform** no action when used in a definition. Instead, their code field addresses are stored in the definition and will be executed only when the defined word is executed. All non-immediate words follow this pattern.

Another type of word executes even when used within a colon definition. The word may, but need not, alter or add to the definition. Primary examples include the flow control words, definition terminator words, and embedded string words. To create an immediate word, type:

```
: name    definition    ; immediate
```

Every definition needs at least one immediate word — the word that signals its end. **;** provides this service in the above example and for all simple high-level colon definitions.

Another immediate word often used in definitions is **'**. This word places the parameter field address of the next word in the input stream on the stack. Assuming that we have a code field address on the stack, we could determine whether it was a variable with the following word: (memory is an IBCL variable)

```
: ?var   ' memory [ cfa @ ] literal =
        if ." variable" else ." not variable" then ;
```

The ." immediate word is used to include a message in a definition. If it is in an active path, the message prints when the word is executed.

Sometimes it is necessary to cause compilation of an immediate word as if it were a non-immediate one. This is accomplished by preceding the immediate word with [compile]. A word to print the address of another word could be defined as:

```
: .address  [compile] ' cr ." address is " . ;
.address some-word
address is .lP
```

This is basically a means for reusing the function of an immediate word within another word which is itself often immediate.

Occasionally it is necessary to cause execution of non-immediate words while creating a colon definition. This is accomplished by a pair of words, [(switches state from compile to execute mode), and] (switches state from execute to compile mode). The [word leaves the definition open. The most common use for this pair would be the calculation of some offset, address, or constant. This pair is frequently used with the literal word, which takes the top value on the stack and enters it into the current definition. The following are equivalent ways to define a word returning the address of the fifth line of a block given its base address, except that the first is inefficient since the operations are performed every time the word is executed:

```
: line-5    64 4 * + ;
64 4 *    : tine-5 literal + ;
: line-5 I64 4 * I literal + ;
```

The second format could lead to ambiguity in a real program, and might not be usable if the stack was busy with control parameters for loops. A similar word, dliteral, is available for compiling double length values from the stack into the definition.

Sometimes the word we are defining will be used to build part of the definition of other words. In this case our definition may contain words that we don't want to execute even when the word executes. Instead we want the word to be copied to the definition being created. An example is the definition of ; which must compile exit at the end of the definition being created:

```
        : ;   compile exit ?csp [;   immediate
```

None of the words defining ; are immediate. When ; executes, the **?csp** and **[** check the stack level and terminate compilation mode. The definition for compile takes the code field address for exit and appends it to the definition being created. Note that compile takes the next word from the definition list, the word following compile should never be immediate.

Let's summarize the behavior of some word, we'll call it a-word, depending on whether or not it is immediate.

		non-immediate	immediate
	a-word	executed	executed
: q	[a-word 1 ;	executed	executed *
: q	a-word ;	compiled	executed
: q [compile]	a-word ;	compiled *	compiled
: q compile	a-word ;	compiled **	error

* These forms are not really used since they are redundant.
** This q must be used in a definition and a-word will be compiled into that definition.

Comments may be inserted within the definition by enclosing them in parenthesis. The opening one must be preceded and followed by a space since it is an IBCL word. The terminating one is just a delimiter.

```
        :name    some words (comments) more words ;
```

Machine Code Definitions

The words in this section provide a simple means of entering machine code definitions for words which must execute rapidly or which require machine resources not immediately available in high-level IBCL. They also support the assembler extensions discussed in the ASSEMBLER paragraph. IBCL remains in the execution mode when creating machine code definitions.

```
code name-of-new-word   words to install code end-code
```

The format resembles the colon definition with **code** replacing : and end-code replacing **;** but words used in colon **definitions** are not generally applicable. Code enters the new name in the current vocabulary and establishes a link to the definition list segment and from there to the code segment. **end-code** generates the code for the inner interpreter **next** linkage and terminates the definition.

Bytes may be added to the definition with **c%** and words with **w%.** It is important to remember that the lowest order byte of a word is stored in the lowest memory location when using the V50. Both **c%** and **w%** use the top word on the stack to obtain the byte or word needed. **w%** is often used to append an address offset from the stack after an opcode that requires an address. **w%** or **c%** are also used to append immediate values.

A string of machine code bytes without addresses or immediate values can be more easily entered using the **<%** and **%>** pair. The following are equivalent ways to enter the definition for +:

```
hex
code +      58 c%     03 c% d8 c%      end-code
code +      5 8  c% d803 w%            end-code
code +      <% 58 03 d8 %>             end-code
```

Variable names and arithmetic to calculate offsets are often used in code definitions to provide addresses. Referencing a constant is often used to provide an immediate value, or an absolute address such as an interrupt vector location.

Constants, Variables, and Arrays

The words in this section provide a basic set of data objects, which can be extended to meet the user's specific needs.

A constant may be defined by typing:

```
nnnn constant name-of-new-constant
```

The top word on the stack provides the value for the new constant. Whenever the new constant is executed, the number nnnn will be pushed onto the top of the stack. The constant can be executed by entering its name outside of a colon definition or within the square bracket pair inside a colon definition. It can also be executed when any definition into which the constant has been compiled is executed.

A signed constant may range from **-32768** through 32767 decimal and an unsigned one from 0 through 65535 decimal.

```
5 constant five ok
five . 5 ok
: print-five five . ; ok
print-five 5 ok
```

A variable is **defined** similarly, except that no initial value is given:

```
variable  name-of-new-variable
```

Whenever the new variable is executed, its parameter **field** address is pushed onto the stack. Values may be stored and retrieved from this location. Initially a single memory word is allotted, but this may be increased using the **allot word** to allocate mmmm more bytes:

```
variable mytable mmmm allot
```

This will create an array of **mmmm+2** bytes within the definition list segment. **allot** checks to see that sufficient room exists within the definition list segment, and issues an error message if the segment will overflow. **allot** is used by all words that assign space from the definition list segment.

Since constants are usually few in number, they can be limited to the definition list segment. Variable arrays, however, can easily require much more room than would be available if they were all confined to a single segment. In IBCL it is possible to give each array its own segment. Each array can be 64K bytes long.

If space permits, assignment within the definition list segment yields programs which execute slightly faster. Placing arrays that participate in the same computations in the same segment will also provide faster execution, if appropriate memory addressing words are used.

To allocate an array in a segment other than the definition list segment type:

```
name-of-segment  length-in-bytes  array-far  array-name
```

In the following example, we will create several arrays:

```
hex 7000 segment my-arrays
my-arrays 1000 array-far employee#
my-arrays 1000 array-far wage/hour
my-arrays 1000 array-far hours-worked
my-arrays 1000 array-far total-pay
```

Before each assignment, a check is performed for sufficient space in the segment. Executing any of these array names will cause the segment paragraph number to be pushed onto the stack, followed by the offset of the array in that segment. This pair is used by all memory addressing words of the form **xxL** (Memory Access paragraph).

Sometimes it is necessary to address into segments other than the definition list segment without allocating space. A label can be created that returns the segment paragraph number and the offset address with which it is initialized.

```
segment-paragraph-nunber  offset-address  label-far  Label-name
```

This word is often used for addressing interrupt vectors and DOS system variables. For instance:

```
0  8  label-far  non-maskable-interrupt-vector
```

User variables are a special type of variable that permit multi-tasking and multi-user applications. They are generally system variables that can vary for different tasks and users. They are assigned sequentially beginning at an address contained in the variable **user-base.** If multiple copies of this array are needed, the user must create another array in the definition list segment, copy the old user array to it, and place the appropriate address in **user-base** for each task.

The user variables at system initialization are:

00 tib	02 >in	04 out	06 c/l
08 voc-link	OA context	OC current	OE fence
10 eprint	12 width	14 warning	16 state
18 base	IA dpl	1C fld	1E hld
20 blk	22 offset	24 scr	26 reserved
28 seg	2A seg-size	2C asmseg	2E reserved
30 '-find	32 reserved	34 'abort	36 reserved
38 'cr	3A 'emit	3C 'expect	3E 'interpret
40 reserved	42 reserved	44 'number	46 reserved
48 'vocabulary	4A 'word	4C span	4E #tib
50 reserved	52 reserved	54 reserved	56 reserved
58 unused	5A unused	5c unused	5E unused

If more user variables are required, a new copy of the array must be made. A new user variable could be assigned by typing:

```
hex    52  user  my-var
```

When executed, **my-var** would push the sum of the address contained in **user-base** and the offset 52 onto the stack. The original user variable array is just below the block of definitions which must be erased to disable the outer interpreter.

Vocabularies

An IBCL system initially contains a single vocabulary named **ibcl. New** words are added to this vocabulary as they are defined. It is possible to create additional vocabularies and to limit the scope of word searches to one of the additional vocabularies followed by the IBCL vocabulary.

In IBCL, the three primary parts of a definition are split into three separate segments. Machine code for all vocabularies occupies the **codes** segment. High level definition lists occupy the **lists** segment. The word's ASCII representation, along with its immediate flag and definition list pointer occupy another segment, with words from different vocabularies accumulating in their own private segments. The IBCLS segment contains the IBCL vocabulary. Creation of new segments for new vocabularies is automatic. The number of bytes to be allocated is contained in the user variable seg-size. This variable is initially set to four kilobytes, but may be reset by the user.

To change the default segment size type:

```
new-size seg-size !
```

If the previous default size is adequate, a new vocabulary and its segment may be created by typing:

```
vocabulary new-vocabulary-name
```

for instance:

```
vocabulary assembler
```

The segment name will be the vocabulary name with **-seg** appended. The example would create the empty **assembler** vocabulary in the assembler-seg segment.

To cause the assembler vocabulary to be searched before the IBCL vocabulary, type:

```
assembler
```

At this point, no words will be found in the assembler vocabulary, but the user variable context will contain a pointer to it rather than to the IBCL vocabulary. New definitions would still be assigned to the IBCL vocabulary.

To cause new definitions to be assigned to the context vocabulary, type:

```
definitions
```

Now the user variable current points to the assembler vocabulary instead of the IBCL vocabulary. current governs which vocabulary receives new definitions. Had we wanted to enter new definitions in my-words, but limit our interpreter searches to IBCL, we would have had to type:

```
vocabulary my-words my-words definitions assembler
```

This would have reset context to point to assembler. Caution, entering a colon definition sets context to current.

In the course of defining new words, you may discover that you have made a mistake. Words can be forgotten and dictionary space can be recovered by typing:

```
forget word-to-forget-thru
```

This type of forget may only be used in the newest vocabulary. If that vocabulary is still IBCL, the user variable fence contains a pointer to a word below which forgetting is disabled, to protect you from forgetting the system.

A more general forget supports switching from one application to another. It requires you to create a task boundary before you **define** the applications vocabularies, segments, and words. It can recover both segments and portions of segments allocated after the boundary. Before entering the application, type:

```
task    name-of-task
```

Forget back to the boundary by typing:

```
name-of-task    forget-task
```

Segment relocation is not affected.

One task boundary is included in the system. Typing:

```
empty    forget-task
```

will forget all words, vocabularies, and segments entered by the user. This will reclaim memory from all non-system words and segments, but will not restore system segments to their original positions in physical memory. **empty** is preserved by **forget-task** but user task names are forgotten along with the task.

Program Segments

IBCL provides limited automatic segment management. The goal is not to provide completely general segment management, but to provide the most economical set of words that still supports use of the entire address space. Words are provided for the creation of new segments.

Initially the IBCL system occupies four segments. The names of these segments are variables that return the address of the first word of the five word segment descriptor table:

1. Segment paragraph number. Absolute address of first byte of segment divided by 16
2. Maximum number of bytes allocated to this segment.
3. Temporary working end of segment. Address of bottom of stack for stack segments
4. End of segment. Current stack pointer for stack segments.
5. Link to first word of segment descriptor for previously defined segment.

The segment names are:

codes	machine code segment
stacks	stack segment, return stack at bottom data stack at top
lists	definition list segment, code address followed by parameter field entries, if any
ibcls	ASCII representation of words

The **memory** variable returns the address of the four word memory descriptor table:

1. Paragraph at which user memory starts, base of **codes.** Paragraphs below this one belong to IBCL.
2. Last available paragraph, set by system board switches.
3. Paragraph at which free memory begins. Initially end of IBCL, updated as new segments are allocated.
4. Pointer to first word of most recently created segment descriptor, this address is returned by **last-seg.**

To create a new segment, type:

```
size-in-bytes  segment  name-of-new-segment
```

The maximum size is 64K. An error message is generated if an attempt is made to exceed memory limits. **segment** uses another word which allocates additional memory and updates **memory,** but does not create a new segment descriptor. This word can be used to increase the length of the top memory segment. The initial last segment is the **lists** segment which usually requires the most space. Its initial space allocation is limited by the need for the system to tit into systems with only 64K bytes of memory.

```
size-in-bytes  dup  allot-seg  last-seg  2+  +!
```

The last line updates the memory limit for the last segment. Be certain that the old value and your increase do not exceed 64K or it will wrap around to a low value. For extra protection, use the following definition:

```
: grow  last-seg 2+ ddup @
        ddup 1+ u< abort" already larger"
            allot-seg     ! ;

final-size grow
```

Two temporary user variables are provided for segment values: seg and seg-size. seg may hold the address of some segment descriptor. seg-size should contain a default segment size. It is used by **vocabulary** to set the size of any new vocabulary segments.

Defining Defining Words

Two actions must be specified when defining defining words. The first is executed when the defining word is executed. The next is executed when the word defined using the defining word is executed. As an example, let us assume that the system does not provide the word defining constants. One way to define this defining word is:

```
: constant   create ,  does   @ ;
```

create and **does>** are immediate words, they execute when the definition is entered. The **@** is compiled as usual.

To define the constant five using this defining word, type:

```
5 constant five
```

The 5 is placed on the stack and momentarily ignored. Referring to the definition of **constant,** the **create** requires a word from the input stream. It takes the string five and adds it to the current vocabulary, with a pointer to the next free word in the definition list segment. This word is initialized with a pointer to the code for variables, and the working end of the definition list segment is incremented by two to point to the parameter field of the word being defined, five. Next the **,** takes the top value on the stack, 5, and stores it at the working end of the definition list segment, the parameter field of **five,** and increments the end pointer by two. Next the **does>** replaces the the contents of the code field with a pointer to a few bytes of code created by **does>** each time it is used. The code has two functions. It nests the interpreter one level deeper, transferring control to the word after does>, and it places the parameter field address of the word being **defined, five,** on the stack. The code is executed only when **five** is executed. Finally, the **@** is compiled and the **;** causes an exit to be compiled and then terminates the definition.

When **five** is executed, the code created by **does>** is executed. The address of **five's** parameter field is placed on the stack and the interpreter nests down to the **@** in the definition of **constant.** A 5 is waiting at the parameter field address, and is returned on the stack. The exit compiled by **;** returns the interpreter to the next higher level, with the 5 remaining on the stack. Or, stated simply, executing **five** causes 5 to be left on the stack.

In actual practice, a word as important as **constant** should define words that execute as rapidly as possible. The run time action should be defined at the machine code level rather than the **high**-level IBCL level used above. This is the technique used in IBCL:

```
        code docon <% 53 8b 5d 02 %> end-code
```

or:

```
        code docon bx push.   bx 2 +[di] mov. end-code
```

This defines the run time behavior of the defined word. Next define **constant itself:**

```
        : constant create , ;' docon
```

The ;' replaces the code field of the word being defined with the code field of **docon**.

Note that **does>;'** and **;code** must all be used with a word that initiates a dictionary entry. **create** is presently the only system supplied word that performs this function,

The above example was a relatively simple use of IBCL to create a defining word. This is admittedly the most complex topic in IBCL, and the vast majority of other languages don't even try to provide this capability.

For a slightly more complex case, consider a double length constant.

```
: dconstant    create  swap  ,  ,
does>   dup @  snap  2+ @ ;
hex 1234.5678 dconstant longfellow
```

dconstant create a double length constant named longfellow. When longfellow executes, it leaves a double length number on the stack. First 5678 is pushed onto the stack, then 1234.

segment is an example of an even more complex defining word. When a segment is being defined, space must be allotted from free memory and a segment descriptor initialized. The run time action is the same as for a variable — the parameter field address is left on the stack unchanged.

vocabulary is the most complex defining word in the system. It must allocate a segment and create definitions and descriptors for both the segment and the vocabulary. The run time action moves the parameter field address from the stack into context.

The only limit to the complexity and utility of words that define words is your imagination. BASIC is one dimensional programming; compiled languages with good subroutine facilities provide a second dimension. IBCL with its ability to define defining words clearly provides a third dimension. Learning to use that dimension well will give you an edge unattainable in other languages.

Internal Workings

Defining a new word increases the memory allocated in the current word ASCII representation segment and the definition list segment. It may increase allocation in the machine code segment. While a definition is being created, the working end of segment is greater than the verified segment end. If an error causes the definition to abort before completion, memory allotted in these segments is reclaimed. The address stored as the verified end is the address of the start of the **definition** being created.

	address of working end	address of verified end	working end
			(end? @)
machine codes	**endc**	**oldc**	here-c
definition lists	**endl**	**oldl**	here
current vocabulary	endw	oldw	here-w

nextw will return the segment and the address of the next free word in the **current** vocabulary segment. A byte or word may be stored to this location from the top of the stack and **endw** incremented with **c,words** or **,words.**

Error checking is performed by the following words:

?comp	error if not compiling
?csp	error if stack position is not that in csp
?pairs	error if top two stack elements unequal
?stack	error if stack out of bounds
?stream	error if input stream exhausted (top of stack true)

Headerless Words

IBCL provides useful features not possible in systems with conventional architectures. Since the vocabulary segments contain only names, count bytes, and pointers into the definition list segment, the entry for each word is completely relocatable and can be deleted when no longer needed. This can only be done with a time consuming (and often expensive) cross compiler on other systems. The following words delete **a** name or range of names and shift the remainder of the **current** vocabulary segment down to close the gap created. lpa stands for the list pointer address field, the first byte pair of the vocabulary entry of a word.

```
        lpa-of-lst-word-name-to-delete    lpa-of-last-name  <behead>

        behead'    name-to-be-deleted
        behead''   name-of-lst-word-name-to-delete  name-of-last
```

<behead> should be used with caution, if at all, since it does not check for valid **lpa's,** and and invalid one will crash the system. behead' and behead" will both respond with "NOT FOUND" if the word was not found. behead" will delete all word names in the physically sequential range. The contents of a vocabulary may be checked with vlist.

Overlaid **Code** Primitives

IBCL code primitives are overlaid upon one another if the "tail" of one happens to be identical to some other useful word. This may also be viewed as having code primitives with multiple entry points.

```
        code  stepbx inc.   bx inc.
        code+ dup@bx push.
        code+     @bx Cbxl mov.     end-code
```

This overlay used 9 machine code bytes instead of the 22 required by conventional systems. **end-code** does not automatically clear the jump table, so branches may occur throughout this structure. High level flow constructs must open and close within a single code, code+ or end-code pair.

Forget-Task

forget-task expects the parameter **field** address of the task on the stack. It checks for a valid task parameter **field** address by verifying that the preceding code field points to **<dotsk>.** forget-task forgets the task descriptor block created by task along with the associated task. It will forget segments and parts of segments and reclaim space for segments or for free memory that has been allotted since the task was declared. It cannot rearrange segments, so segment sizes and positions should not be changed within the confines of a task . . forget-task pair. Otherwise the forget-task may leave gaps in the allotted segment map and may reduce the free memory pointer to a value below that of the highest segment actually allotted.

```
                task name-of-task
                definitions
                definitions
name-of-task    forget-task
```

CONTROL

IBCL contains high-level control structures similar to those found in BASIC and Pascal. These perform conditional execution and repeated execution of word blocks, and case selection. They eliminate the need for any program position labels such as BASICS line numbers.

Words that control the flow of program execution are used only within colon definitions (Detining New Words section). They are immediate words which execute when the colon definition is first compiled. Most cause branches or conditional branches to be compiled into the definition list of the word being compiled, but a few merely save an address and identifier on the stack for use by a later control word.

The branch compiled into the definition list may be a conditional Obranch or an unconditional branch. The Obranch is ignored if the top word on the stack is nonzero. In either case the branch fills two words in the definition list. The first, as with any compiled word, is a pointer to the code field address of the word, in this case branch or Obranch. The second word is the byte offset of the destination relative to the second word.

The conditional branch always uses and drops the top stack word.

Vectored execution permits words to be defined after they have been used in creating other definitions. It is usually useful only within colon definitions.

Vectored Execution

You cannot actually include a word in a definition if that word has not already been defined. If the function you wish to perform cannot be defined before the word in which it is used, you must first define a variable that will eventually contain the code field address of the not yet defined word.

```
variable vector-name
: some-word      words    vector-name @ execute words ;
: future-word    words ;
' future-word cfa vector-name !
some-word
```

The vector name used in some-word compiles like any variable. When executed, it leaves its parameter field address on the stack and @ replaces that address with the variable's contents. This variable was initialized on the last line to contain the code field address of future-word. (' returns the parameter field address of the next word in the input stream and cfa converts the parameter field address to the code field address.) execute executes the word whose code field address is on the stack, just as if it had been compiled into the definition.

Since execute is almost always preceded by @, IBCL includes execute@ — a more efficient replacement for this pair.

These words are used when the implementation of a word may have to change after words which use it are defined. System words with this capability are vectored through user variables.

system word	user variable	routine
-find	'-find	<-find>
abort	'abort	<abort>
cr	'cr	<cr>
emit	'emit	<emit>
expect	'expect	<expect>
interpret	'interpret	*interpret*
number	'number	<number>
vocabulary	'vocabulary	<voc79>
word	'uord	<word>

These words are also necessary to fully implement recursion. They enable the creation of a circular set of definitions in which each word can include previously defined words as usual, but may also include words defined later in the sequence. Recursion is not very efficient for numerical operations such as factorial evaluation but is very efficient for symbolic manipulations.

Conditional **Execution**

The if true-phrase else false-phrase **then** construct is used within colon definitions to enable a number on the stack to control whether or not groups of words within the definition are executed. A phrase is any list of words normally allowed in a colon definition. If conditional or loop constructs are included, they must be completed within the phrase. Nesting is limited only by stack size — overlapping is forbidden.

The following example will display a game score along with one of two messages (the new score is on the stack):

```
: .scoredup high-score @>
        if   dup high-score !  ." new high score!!!" .
        else  ." your score is "
                ." high score is " high-score @.  then ;
```

When **.score** is executed your latest score should be at the top of the stack. It is duplicated and compared with the old high score. The comparison sets the top number on the stack to 0 (false) if your score is not greater than the old high score. It sets the top number to a **nonzero** (true) value otherwise. If the number is 0, execution will branch to the words after the **else.** If it is **nonzero,** execution will continue after the **if,** then skip the words between **else** and then. The true part sets the new high score, then displays **new high** score!!! and the new score. The false part displays your score and the old high score.

else and the words between it and **then** may be omitted, in which case no action is taken if the condition is false.

The **if** compiles a **Obranch and puts** the address of its destination field on the stack. It then places an identifier on the stack to signal its presence to **else** or **then.** The **else** checks for if's identifier and issues an error message if it isn't found. else next compiles an unconditional branch. It calculates the offset from the address on the stack to the word after the branch and stores that offset into the original Obranch. The address of the destination field of the branch is placed on the stack, followed by another copy of the identifier. The **then** aborts with an error message if the identifier isn't found, but does not need to know whether it follows an **if** or an **else.** It calculates the offset from the address on the stack to the next free word and stores it into the previous branch.

Loops

Loop constructing words are similar to the above words in that they compile branches and leave addresses on the stack. As above, a phrase may be any list of words normally allowed in a colon definition. If conditional or loop constructs are included, they must be completed within the phrase. Nesting is limited only by stack size — overlapping is forbidden.

There are three types of conditional loops:

```
        begin phrase again
```

This is really an unconditional infinite loop since it has no exit. The only legal exit would be an **abort** or **abort"** within the phrase or within a word in the phrase.

```
        begin phrase until
```

until is a bit like **if** except that it compiles a backward branch to the beginning of the phrase. The phrase executes repeatedly until the top word on the stack is true (nonzero). The phrase always executes at least once.

```
        begin test-phrase while phrase repeat
```

After the test phrase is executed, the top word on the stack is examined. If it is true (nonzero), the phrase is executed and control branches back to the test-phrase. If it is false (zero), the loop is exited and execution continues after the **repeat.** The second phrase will not execute even once if the initial test-phrase was false.

There are also three types of do loops:

```
limit start do phrase loop
```

This **do** loop starts execution with an index set to start and increments that index by one for every encounter of loop. The phrase is executed repeatedly until the index equals or exceeds the limit using a signed comparison. The limit and start values are taken from the data stack at execution time. While executing the loop, the index is on top of the return stack with the limit under it. You may use the return stack within the phrase, but its condition at the end of the phrase should be the same as at the beginning. The data stack is not used other than on entry.

```
limit start do phrase +loop
```

This is similar to the first do loop, but the **+loop** takes a signed number from the data stack and adds this to the index instead of incrementing the index by one. If the increment is positive, termination is the same as for loop. If the increment is negative, execution repeats until the index is less than the limit.

```
limit start do phrase /loop
```

This is similar to the above loops, but **/loop** takes an unsigned number from the data stack and adds this to the index instead of one. The limit and start values are naturally unsigned also.

Several words are useful only within the above do loops.

i	Place a copy of the index on the data stack.
i'	Place a copy of the index on the data stack. (Assuming it has been buried one deep and is the second element on the return stack)
j	Place a copy of the index of the next outer do loop on the data stack. (The index should be the third word on the return stack)

leave	Set the index equal to the limit, causing exit of this do loop when execution of the phrase completes.

Case

Machine code level support is provided for implementing efficient case statements. The word **match** signifies that the required case has been found and execution should skip the next higher level colon definition. This often avoids the necessity of deeply nested if . . . else . . . **then** clauses.

These examples assume a temperature is available on the stack:

```
: ?hot          dup high-temp @ >  ;
: ?cold         dup   Low-temp @ < ;
: ?good         dup good-temp @ = ;
: goldy         ?hot
                if  ." This porridge is too hot!"
                else    ?cold
                        if ." This porridge is too cold!"
                        else    ?GOOD
                                if ." This porridge is just right!”
                                else ." Oh well, it will do."
                                then   then  then  drop  ;

: hot ?hot if ." This porridge is too hot!"     drop match then ;
: cold ?cold if ." This porridge is too cold!"    drop match then ;
: good ?good if ." This porridge is just right!” drop match then ;
: fine          ." Oh well, it will do." drop match ;
: goldy   hot cold good fine ;
```

The version using **match** is much easier to read. Execution exits **goldy** after finding a single match. Words in goldy beyond the match are not entered. **match itself including** its copy of the inner interpreter next routine requires only 6 bytes of machine code.

USING ASSEMBLY LANGUAGE FROM IBCL

MacBus uses a NEC **V50** microprocessor and the NEC 72191 numerical coprocessor. Most users, however, are more familiar with the Intel 8088 and 8087 instruction sets, with which the NEC chips are completely compatible. For this reason, we will use Intel mnemonics throughout the discussion below.

National Instruments supplies the assembler in IBCL source form on the Macintosh Plus diskette shipped with MacBus. In order to use the assembler, you must download the file microasm to MacBus.

The assembler is loaded by having MacBus download the utility file microasm. Refer to SECTION ONE - Source Code Files for more details.

If IBCL-like high-level flow control words are required and you are not loading **autoopt,** you must also load the control words.

The high-level flow control words are loaded by having the host download the utility file **hiflow**. See SECTION ONE - IBCL Source Files for details.

After the assembler has been loaded, the program you are writing may be directed to another vocabulary. The following words will cause your words to be defined in the my-words vocabulary, even though words in the definition are found in the microasm vocabulary.

```
vocabulary my-words my-words definitions microasm
```

If you are using the assembler to write a stand alone machine code program rather than add to the IBCL primitives, you may want to direct the code to a segment other than the IBCL codes segment. The assembler will assemble code at the working end of any segment. The user variable asmseg contains the segment descriptor address of the current assembly segment. Initially it points to codes. To create a new segment and assemble our original example into it, type:

```
500 segment my-code          (create new segment to receive code)
my-code esmseg !             (redirect new code definitions)
>>>far my-entry              (my-entry will return address)
                             (of next byte of code assembled)
                             (in this segment.)
(assembly)
( code)
(   definitions)
here-c oldc !                (set segment end to working end )
codes    asmseg !            (rest to system codes segment )
```

Remember to reset the code definition segment to **codes** in case any future definitions generate code in the IBCL machine code segment **codes. >>>far** creates a label named my-entry. Execution of this label will return the segment paragraph number and offset with that segment for use by future calls and jumps. The here-c **oldc!** sequence sets a fence below which future errors will not reclaim memory. Without it an error assembling n'th subroutine would reclaim the entire segment instead of just the space allotted to that word. The end-code terminating word of a code definition provides this function when defining IBCL code primitive words.

Usually code should only be used when asmseg points to codes. If you need more than 64K of pure code, export your longer routines and call them through the **codes** segment. Overhead using this method is negligible. One exception is that asmseg may point to an alternate segment to create a special purpose **codes** segment that will later replace the original IBCL codes segment. Replacing IBCL system segments is an advanced capability best left to experienced programmers familiar with the system.

Assembler Mnemonics

The assembler accepts mnemonics similar to those of the IBM-PC Macroassembler. The main difference is that operands are listed before operators, and operators other than jumps end with a period. If you also choose to end your jump label names with a period, this leads to an instruction syntax that resembles sentences, and is easier to read than typical Forth assembly language. The following simple example defines a word that adds the contents of variables x and y and stores the result in z.

```
code add  ax  x +[] mov.  ax y +[] add.  z +[] ax mov. end-code
```

First specify the destination, then the source, then the operator. The **+[]** operand assembles an offset without base or index register. Some operators have only a source or destination, or an operand that is both. The operand is always typed before the operator, with the exception of labels for jump and label operators. If an offset or an immediate value is required, it is taken from the data stack.

The complete Intel NEC **V40/V50** instruction set is supported. This assembler may be used to write any program that could be written with any conventional assembler.

Macro Definition

Macro capability is inherent by the very nature of IBCL. The source code sequence forming the macro is used in a colon definition to define the macro. Any element in the code sequence may be replaced with a dummy entry created by the defining word **param.** All dummy entries must be set before the macro is used by a code definition. For the above example let us assume that the temporary register **ax** may vary:

```
: param create does cfa execute ; (utility)

param  temp
: mat-add      temp  x  +[]  mov.  temp y +[]  add.
         z +[] temp mov.;              (adder macro)
' ax   ' temp  !
code add       mat-add      end- code                  (use ax)
' dx   ' temp  !
code add-dx    mac-add      end-code                   (use dx)
```

Branch Control

The assembler provides high-level branch control similar to IBCL control structures in function, but implemented in an altogether different manner. Do not use the IBCL control words inside the assembler. The assembler control words end with a period.

Machine level branch control is also available for both forward and reverse branches. A jump table of hex 80 bytes is provided. This value may be increased by editing the value of **jmp-lim** in the **microasm** file. Each label requires space for the number of characters in the label plus 3 bytes. An entry occurs for each non-reconcilable reference to a label. When a label's location becomes known, these entries are removed and an entry for the label itself is created. The definition of the word that returns the absolute value of a number on the stack illustrates label and branch syntax.

```
code abs
  bx bx or.  jns positive. bx neg. >>> positive. end-code
```

The period ending the labels is a strongly encouraged convention. It ensures that all operational units end with the same visual cue. The label is not an IBCL word, but a string argument for the jump or label operator. The label operator **>>>** makes label locations easy to spot at a glance.

Things to Remember

When learning assembly language from an IBCL system, remember that you have a marvelous tool that lets you learn in small steps, and that enables you to check your understanding at the end of every step. Learning assembly language has never been so easy. While you are learning, remember that IBCL uses all the registers while interpreting from word to word except for ax, cx, dx, di, and es. These five registers may be used within your practice machine code words without crashing the system. The remaining registers would have to be saved and restored — fine if you are an expert but unnecessary if you are learning. Even experts would first look for a way to avoid the overhead of first saving a register to memory and then restoring it. The five free registers may be changed within any word, don't count on their values across word boundaries unless you know the structure of the intervening words.

NEC V40/V50 Architecture

Registers

Now for a word on the NEC V50. You already know that you have up to a megabyte of memory that can be addressed by a paragraph number and an offset. The most common addressing mode assumes the **lists** segment for a default paragraph number — you only need to provide the offset to store and retrieve 16-bit and occasionally 8-bit numbers. In assembly language you have a dozen more addresses at which your numbers may be stored. They are not addressed by number, but by a two character name. The addresses of the 16-bit registers are:

ax	
bx	top element of the stack is at bx
cx	
dx	
bp	offset into stack segment of top of return stack
sp	offset into stack segment of top of data stack
si	offset into lists segment of word to execute next
di	
cs	paragraph number of code segment (codes)
ss	paragraph number of stack segment (stacks)
ds	paragraph number of definition list segment (lists)
es	

The first four 16-bit registers may also be addressed as though they were eight 8-bit registers. They are split into high and low-order halves.

ah al bh bl ch cl dh dl

Remember that bh holds the high-order half of the top word of the data stack and bl holds the low-order half. bh and **bl** are not separate registers from bx, but only a way to address it a byte at a time.

All registers except the segment registers may be used as source or destination for most math and logic operations. The byte accumulator al or the word accumulator ax is always the assumed destination for multiplication and division. The segment registers are restricted to **mov. push.** and pop. operations.

Memory Addressing

Some of the registers may also be used to provide an offset into a segment for memory addressing. The simplest example is the code for **@** which replaces the top word on the stack with the number at that offset in the definition list segment.

```
code @   bx [bx] mov.   end-code
```

The source is the 16-bit number at the offset found in bx. The destination is the top of the stack, namely the **bx** register. Most operators use the paragraph number in ds to select the segment, but any segment register may usually be explicitly selected.

```
code @e es:   bx [bx] mov. end-code
```

In the definition of **@e,** the es register is selected to provide the paragraph number. The segment selectors (cs: ss: ds: es:) immediately generate one byte of machine code, so they may also be used directly before the memory operand.

```
code @e   bx es: [bx] mov. end-code
```

This produces the same code as above.

The following registers and combinations may be used to provide memory offsets. If more than one register is used, the contents of the two are added in a temporary scratch register.

```
[bx]    [si]    [bx+si]    [bx+di]
        [di]    [bp+si]    [bp+di]
```

It is also possible to form an offset by adding a constant to the contents of the above registers or combinations. If a constant is to be added use the names below.

```
+[bx]   +[si]   +[bx+si]   +[bx+di]      +[]
+[bp]   +[di]   +[bp+si]   +[bp+di]
```

Note that the addressing mode corresponding to **[bp]** does not exist on this microprocessor. It has been replaced with a special mode **+[]** which uses a constant offset with no base register. In all cases the constant used is the top word on the stack.

Single operand instructions need to know whether memory operands are bytes or words. The following instruction to increment memory at the offset stored in **bx** is incomplete and generates an error since it does not indicate whether a byte or word should be incremented:

```
Cbxl inc.          (error  b/u?)
```

To specify byte or word, use one of the following:

```
Cbxl byte-ptr inc.
[bx] word-ptr inc.
```

Both the source and destination may be a register, but a memory offset is legal for only one at a time. Attempting to use memory offsets for both source and destination will generate an error message.

For some operators, the source may be an immediate value. In this case the machine code created will include the immediate value. An immediate byte or word is declared by typing **ib** or **iw** instead of the register or offset words.

```
ax 339 iw mov.  (store 339 into ax)
dl  51 ib add. (add   51 into dl)
```

The destination can usually be either a register or memory. The immediate operands obviously can't be used as destinations.

Flags

The flags and their abbreviations are as follows. The abbreviations are not assembler words. Assembler words to set and reset the flags and move them to and from the stack and the **ah** register are covered in the instruction section. Flags are generally set by math or logic operations and are unaffected by move, stack, or control transfer instructions.

overflow	of	or	0
direction	df		d
interrupt	if		i
trap	tf		t
sign	sf		**s**
zero	**zf**		z
auxiliary carry	af		a
parity	**Pf**		**P**
carry	cf		c

Abbreviations and Conventions

The following abbreviations are used for sets of legal operands.

b-reg	ah bh ch dh al bl cl dl
w-reg	ax bx cx dx bp sp si di
reg	b-reg or w-reg
seg-reg	cs **ss** ds es
memory	[bx] [si] **[bx+si]** . . . **+[]** . . . **+[bp+di]**
reg/mem	reg or memory
immed	ib iw
accum	byte or word accumulator **-** ax or al

Source and destination operands must both be byte or both be word length operands.

All operands of the form **+[?]** use a number from the stack to provide an offset relative to the contents of the registers. The ib and **iw** operands use a number from the stack to provide the immediate value. The operand identifier words above do not use the stack, so no conflict exists with immediate values or offsets on the stack. If both immediate and offset values are required, the immediate value should either be below the offset, or preferably not placed on the stack until just before the **ib** or **iw.** Both numbers are included in the machine code created for the current operator.

If a short 8-bit offset or immediate is legal and the value is small enough, the assembler will generate the code for that form.

Error **Messages**

not found	Attempted to use undefined word.
too many	Cannot have more than two operands.
byte,word	Byte and word operands cannot be combined.
mem,mem	Source and destination cannot both be memory.
immediate	Immediate operand not permitted here.
segreg	Segment register not permitted here.
operands?	Missing operands.
dest=cs	Destination cannot be code segment register.
dest?	Missing destination.
source?	Missing source.
2 **operands**	Instruction accepts only one operand.
operand ?	Single operand missing.
byte	Word operand required.
count ?	Count missing for shift or rotate.
too large	Value too large for byte representation.
too large	External opcode exceeds 6 bits for ESC.
not acc	Input destination or output source must be an accumulator.
table full	Jump table full.
too far	Label out of range for short relative jump. Destination must be within -128 through +127 bytes of byte following jump instruction.

Timing

Since the main purpose of programming in assembly language is to achieve the fastest possible execution times, the clock cycles required are listed along with the instructions. Often special forms of an instruction provide significantly faster execution. If a + follows the number of clocks, you must add a correction to number of clocks for that instruction. The correction depends on the addressing mode used:

operand		clocks
[bx]	[si] [di]	5
+[]		6
[bx+si]	[bp+di]	7
[bx+di]	[bp+si]	8
+[bx] +[bp]	+[si] +[di]	9
+[bx+si]	+[bp+di]	11
+[bx+di]	+[bp+si]	12

For a word operand at an odd address, add 4 more clock cycles.

Instruction Set

Data Transfer

mov. Move a byte or word from one location to another. The source and flags are not affected.

dest	source	operator	clocks	comment
reg	reg	mov.	2	
seg-reg	w-reg	mov.	2	cs illegal dest
w-reg	seg-reg	mov.	2	cs legal source
reg	immed	mov.	4	
accum	+[]	mov.	10	
41	accum	mov.	10	
reg	memory	mov.	8+	

memory	reg	mov.	9+
memory	seg-reg	mov.	8+
seg-reg	memory	mov.	9+

push. Push the contents of the source location onto the stack. Decrement the stack pointer sp by 2, and copy the source contents to that address in the stack segment. Flags are not affected.

w-reg	push.	10
seg-reg	push.	10
memory	push.	16+

POP. Pop the top of the stack into the destination location. Copy the word in the stack segment at offset contained in sp to the destination, and increment sp by 2. Flags are not affected.

w-reg	pop.	8
seg-reg	pop.	8
memory	pop.	17+

xchg. Exchange the byte or word source and destination operands. Flags are not affected.

accum	reg	xchg.	3
reg	reg	xchg.	4
memory	reg	xchg.	17+

lea. Load effective address. Load the effective address of the source operand into the destination operand. Unlike the mov. instruction, the address rather than the contents of that address are loaded into the destination. The flags are not affected.

w-reg	memory	lea.	2+

Ids. Load data segment pointer. Load the destination register from the word at the memory offset. Load the ds segment register from the next higher word. Flags are not affected.

w-reg	memory	**lds.**	**16+**

les. Load extra segment pointer. Load the destination register from the word at the memory offset. Load the es segment register from the next higher word. Flags are not affected.

w-reg	memory	les.	**16+**

lahf. Load ah from the flag registers.
sahf. Store ah to the flag registers.
(sign zero ? aux-carry ? parity ? carry)
The bits marked **?** are undetined

lahf.	4
sahf.	4

pushf.
popf. Push the flag registers onto or pop the flag registers from the stack. The flags occupy these bit positions:

bit 11	overflow	of
10	direction	df
9	interrupt	if
8	trap	tf
7	sign	sf
6	zero	zf
4	auxiliary carry	af
2	parity	Pf
0	carry	cf
	pushf.	10
	popf.	8

xlat. Translate. Replace byte in al with byte at offset given by sum of bx and al. The action would be equivalent to al [bx+al] mov., except that this instruction is illegal. Essentially, this command provides an 8-bit index register. A similar but slower instruction with a 16-bit index register is al [bx+si] mov. Flags are not affected.

	xlat.	11

in.
out.

Transfer a byte or word from an input port to ax or al. Transfer a byte or word from ax or al to an output port. The port number may range from 0 through 255. Flags are not affected.

accum	**port#**	in.	**10**
accum	**port#**	out.	**10**

in-dx.
out-dx.

Transfer a byte or word from an input port to ax or al. Transfer a byte or word from ax or al to an output port. The port number must be in dx and may range from 0 through 64K. Flags are not affected.

accum	in-dx.	8
accum	out-dx.	8

Arithmetic and Logic

Two operand instructions: Flag usage is indicated to right, using bit order as would be pushed to stack. Blank bit positions are undefined, . are unaffected.

			BA9876543210
adc.	Add with carry	dest + source + cf	o...sz ap c
add.	Add	dest + source	o...sz ap c
and.	Bitwise logical AND	dest AND source	o...sz p c
cmp.	Compare virtual	(dest - source)	o...sz ap c

or.	Bitwise logical OR	dest OR source	o...sz p c
Sbb.	Subtract with borrow	dest - source - cf	o...sz a p c
sub.	Subtract	dest - source	o...sz ap c
xor.	gitwise exclusive OR	dest XOR source	o...sz p c

The above words require a destination followed by a source in one of the following forms.

dest	source	operand	clocks
reg	reg	xxx.	3
acc	immed	xxx.	4
reg	immed	xxx.	4
reg	memory	xxx.	9+
memory	reg	xxx.	16+
* memory	reg	cmp.	9+
memory	immed	xxx.	17+

		Flags
test.	Bitwise logical AND, destination unaffected	o...sz p c

reg	reg	test.	3
acc	immed	test.	4
reg	immed	test.	4
reg	memory	test.	9+
memory	reg	test.	9+
memory	immed	test.	10+

One operand instructions:

			Flags
dec.	Decrement	dest - 1	o...sz a p .
inc.	Increment	dest + 1	o...sz a p .
neg.	Negate	2's complement of dest	o...sz a p c
not.	Bitwise logical not	l's complement of dest	....

The above instructions accept one operand in one of the following forms:

dest	operand	clocks
reg	xxx.	3
memory byte-ptr	xxx	**16+**
memory word-ptr	xxx	**16+**

For the multiplication and division instructions, an accumulator is always the destination. The high-order extension of the al accumulator is the **ah** register. The high-order extension of the **ax** accumulator is the dx register. For multiplication, the accumulator is multiplied by the source and the result overflows into the extension if necessary. For division, the numerator is a double length number held in the accumulator and its extension. A single length signed numerator may be converted to double length by **cbw.** or cwd. (convert byte to word or convert word to double word). The extension for an unsigned numerator may be set to zero.

The remainder is left in the extension register and has the same sign as the numerator. The quotient is left in the accumulator. If it is too large, a type 0 divide overflow interrupt is generated.

Flags

div. Unsigned division. ...

b-reg	div.	90
w-reg	div.	155
memory byte-ptr	div.	**90+**
memory word-ptr	div.	**155+**

idiv. Signed division. **o...sz** a p c

b-reg	idiv.	112
w-reg	idiv.	177
memory byte-ptr	idiv.	**112+**
memory word-ptr	idiv.	**177+**

mul. Unsigned multiplication. Overflow and o... c
carry flags set if extension becomes **nonzero.**

b-reg	mul.	71
w-reg	mul.	124
memory byte-ptr	mul.	**71+**
memory word-ptr	mul.	**124+**

imul. Signed multiplication. Overflow and o... c
carry flags set if extension is not the
sign extension of the accumulator.

b-reg	imul.	90
w-reg	imul.	144
memory byte-ptr	imul.	**90+**
memory word-ptr	imul.	**144+**

The following instructions accept no operands. The source and destination, the **ax** or **al** accumulator, are implied by the command.

cbw.	Sign extend al into ah	2	....
cwd.	Sign extend ax into dx	5	
aaa.	ASCII adjust after addition	4	... a c
aas.	ASCII adjust after subtraction	4	... a c
aam.	ASCII adjust after multiplication	83	. ..sz p
aad.	**ASCII** adjust after division	60	. ..sz p
daa.	Decimal adjust after addition.	4	. ..**sz** a p c
das.	Decimal adjust after subtraction.	4	. ..**sz** a p c

The ASCII adjust operations are used to correct the accumulator after arithmetic on unpacked decimal numbers, numbers with one decimal digit per byte. **al** may exceed the value of a legal decimal digit after such an operation. These operators store the legal higher order digit in ah, leaving al within the 0 to 9 range.

The decimal adjust operations perform a similar function for packed binary coded decimal where each half of al stores a separate decimal digit.

Rotate and Shift

The rotate and shift instructions accept a single location operand and a number from the stack to indicate a single or a multiple bit shift. The number from the stack is not actually used to set the shift count, but only to select the single versus multiple bit machine code pattern. A one on the stack will select the single bit shift, anything else will select multiple. For multiple bit shifts, the count must have been set in the cl register by some previous instruction.

Rotation causes bits that are forced out one end of the byte or word to be reinserted at the other. Bits that are forced out by a shift never reenter the operand. Bits forced out of either end of any operand are moved into the carry flag cf.

Note: If count is not one, the overflow flag of is undefined. For right shifts and rotates by one the overflow flag is set to one only if the two new high-order bits are unequal. For left shifts and rotates by one the overflow flag is set to one only if the new high-order bit does not equal the carry flag.

		Flags	
rol.	Rotate left, high-order bit to low.	0.....	. . c
for.	Rotate right, low-order bit to high.	0.....	. . c
rcl.	Like rol. but cf is included in the circle.	0.....	. . c
rcr.	Like ror. but cf is included in the circle.	0......	. . c
shl.	Shift left, zero low-order bit.	o...sz	p c
shr.	Shift logical right, zero high-order bit.	o...sz	p c

The above instructions accept one operand and a count on the stack in one of the following formats. The count may follow the operand or may already be on the stack.

dest	count	operator	clocks
reg	1	xxx.	2
reg	2..15	xxx.	8 +4 per bit
memory byte-ptr	1	xxx.	15+
memory word-ptr	1	xxx.	15+
memory byte-ptr	2..15	xxx.	20+ +4 per bit
memory word-ptr	2..15	xxx.	20+ +4 per bit

String Manipulation

The string manipulation instructions accept no operands. Register usage is implied by the instruction. Generally the data segment, extra segment, source and destination registers (ds es si di) must be set by previous instructions. In some cases one of the default segment registers may be replaced by an explicit declaration using the segment register prefixes (cs: ss: ds: es:). These may be used in conjunction with the repeat prefix (repx.) and/or the bus lock prefix (lock:). If more than one prefix is used, interrupts must be disabled. The return from an interrupt returns control to at most one prefix byte before the instruction.

Both byte and word versions of all string operators exist.

The source string defaults to the data segment at the offset given by the source index register. The segment may be over-ridden by an explicit segment prefix. The destination string is always in the extra segment at the offset given by the destination index register. If the operation has a repeat prefix, the repeat count must have been placed in the cx register by a previous instruction. The repeat prefix will cause the following string operator to be repeated and cx decremented by one until cx reaches zero. If the string operator is **cmpsx.** or **scasx.,** the operation will be ended prematurely if the zero flag becomes set when using **repz: or repe:** or if the zero flag becomes zero when using **repnz: or repne:.**

After each operation, the **si** and **di** registers are incremented if the direction flag is clear and decremented if the direction flag is set. The delta is one for byte operators and two for word operators.

	clocks	
movsb.	17	Move byte (word) string. Transfer byte (word)
movsw.	17	from source to destination . . . .
cmpsb.	22	Compare byte (word) string. For this compare,
cmpsw.	22	the destination is subtracted from the source
		but only the flags are affected.o...sza p c
lodsb.	12	Load byte (word) string. Transfer byte (word) to
lodsw.	12	accumulator al or ah from source . . . .
		di not used or affected.
stosb.	10	Store byte (word) string. Transfer byte (word)
stosw.	10	from accumulator to destination . . . .
		si is not used or affected.
scasb.	15	Scan byte (word) string. For this compare,
scasw.	15	the destination is subtracted from
		the accumulator but only the flags
		are affected. si is not used or
		affected.o...sza p c
repz:	6	Repeat following string operation until cx becomes
repe:	6	zero, decrementing cx each iteration. Terminate
repnz:	6	repz: or repe: if zero flag becomes one.
repne:	6	Terminate repnz: or repne: if zero flag becomes
		zero. Premature termination possible only for
		compsb. compsw. scasb. and scasw.

Jumps, Calls, and Loops

There are two formats for jumps and calls. Calls, indirect jumps, and intersegment jumps follow the usual operand first syntax. Intrasegment direct jumps, both conditional and unconditional, are followed by the target label. Loops are like conditional jumps except that a count in the cx register is decremented by one. The branch is taken only if the cx register has not become zero.

The segment and offset for intersegment direct jumps and calls are most easily provided by creating a label with **>>>far**. Suppose we are assembling code into an auxiliary segment and want the target of a direct call or jump to be within that code:

```
        previous code >>>far come-here remainder of code
then
        come-here  callf.
```

will assemble a far call to come-here.

Calls push the offset of the next instruction onto the stack. For intersegment calls the cs register is pushed first. Control then transfers to the target location.

Calls may be direct or indirect, intrasegment or intersegment. For direct calls, the destination offset is on the stack. The target segment is next on the stack for far calls.

For indirect calls the usual destination operand format is used.

			clocks	
	offset	call.	11	Call intrasegment direct.
	reg	**calli.**	13	Call intrasegment indirect.
	memory	**calli.**	**13+**	Call intrasegment indirect.
segment	offset	callf.	20	Call intersegment direct.
	memory	callfi.	**29+**	Call intersegment indirect.

A called subroutine may return control to the instruction after the call by using a return instruction. The top stack value is popped into the instruction pointer. For far returns the next stack value is popped into the code segment register. If the return contained the + character, it was assembled with a number from the stack which will be added to the stack pointer **sp** to discard parameters. The number should be even, since all stack operations are on words.

Intrasegement returns must be used with intrasegment calls, and far (intersegment) returns must be used with far calls.

	ret.	8	Intrasegment return.
#bytes-drop	ret+.	12	Intrasegment return, add immediate to sp.
#bytes-drop	**retf+.**	17	Intersegment return, add immediate to sp.

Unconditional jumps may be direct or indirect, intrasegment or intersegment. For direct intersegment jumps, the destination offset and target segment are on the stack. For indirect jumps, the usual destination operand format is used. Direct intrasegment jumps are covered with conditional jumps.

segment	offset	jmpf.	7	Jump intersegment direct.
	reg	jmpi.	7	Jump intrasegment indirect.
	memory	jmpi.	**7+**	Jump intrasegment indirect.
	memory	**jmpfi.**	**16+**	Jump intersegment indirect,

Conditional and unconditional intrasegment jumps transfer to a location at an offset relative to their own position. Use of a jump table to reconcile jumps and targets prevents any space from being permanently lost in the dictionary. The table is necessary since the location of forward jumps is unknown until the target label is assembled.

The relative offset of the unconditional jump **jmp** allows access to the full segment. Offsets for the conditional jumps, and the short form of the unconditional jump, must be within -128 through **+127** bytes of the next instruction.

There is only one label defining word for these jumps. It does not directly assemble any machine code, but creates an entry in the jump table consisting of the label's length, name string, and absolute offset within the current **codeseg** segment.

If any jumps have previously used this label, their entries in the jump table are used to insert the correct offset in their code and the entry is removed. Future references to that label by other jumps immediately enter the correct offset and no further table entries are created. The jump table may be cleared by setting its first word, number of entries, to zero.

```
        0 jmp-tab !
```

The constant **jmp-lim** contains the number of bytes allotted for **jmp-tab.** It should be sufficient for defining IBCL primitives of any reasonable size, but the value in the file may be edited to provide sufficient room to use the assembler to create programs of any size.

To create a label, type:

```
        >>> label-name.
```

The **>>>** string (pronounced label) makes labels easy to spot. The same string can be reused for a new label only if the jump table has been cleared. To clear the jump table, set the first word of **jmp-tab** to zero. The jump table is not automatically cleared by terminating one code primitive or beginning another.

Since labels are just entries in the jump table, they cannot conflict with IBCL words. All labels defined using **>>>** should end with a period to make your programs more readable. This is a convention, not a requirement.

NOTE: The operator for these instructions does not end with a period. It is not the end of the instruction.

All direct intrasegment jumps and loops follow the form:

```
        jump-operator label-name
```

For instance, the primitive to return the absolute value of a number on the stack:

```
code abs
        bx bx or.  jns positive. bx neg. >>> positive. end-code
```

	clocks	
jmp	7	Unconditional direct jump. Only jump with

label that may be anywhere in segment. Long form of relative offset always used.

xxx	8	If jump is taken.
xxx	4	If jump is not taken.

The remaining jumps are conditional and the destination label must be within -128 through **+127** bytes of the byte after the jump. The conditions for taking a jump are given both as flag settings and relation of destination to source. The latter form is relevant following a subtract or compare operation.

Jump if ....., dest is .... source.

ja	cf=O and **zf=0**	above
jae	cf=O	above or equal to
jb	cf=l	below
jbe	cf=l or zf=l	below or equal to
jc	cf=l	below
jcxz	cx=o	(if count is zero)
je	zf= 1	equal to
jg	sf=of and zf=O	greater than
jge	sf=of	greater than or equal to
jl	sf of	less than
jle	sf of or zf=l	less than or equal to
jmps	unconditional, short	
jna	cf=l or zf=l	not above
jnae	cf= 1	not above or equal to
jnb	**cf=0**	not below
jnbe	**cf=zf=0**	not below or equal to
jnc	cf=O	
jne	**zf=0**	not equal to
jng	sf of or zf=l	not greater than
jnge	sf of	not greater than or equal to
jnl	sf=of	not less than
jnle	sf=of and zf=O	not less than or equal to
jno	**of=0**	(jump if no overflow)
jnp	pf=O	(parity odd)

jns	sf=0	(no sign)
jnz	zf=O	(not zero)
jo	of= 1	(jump if overflow set)
jp	pf= 1	(parity even)
jpe	pf= 1	(parity even)
jpo	pf=O	(parity odd)
js	sf=1	(negative)
jz	ZF= 1	(zero)

Above and below refer to unsigned numbers.
Greater than and less than refer to signed numbers.

loopcx	cx	Loop (jump) if cx is not zero.
loope	cx and zf=l	Loop if equal.
loopz	cx and zf=1	Loop if zero.
loopne	cx and zf=O	Loop if not equal.
loopnz	cx and zf=O	Loop if not zero.

Conditional Execution & Loops

The high-level flow control words parallel the high-level flow control words of IBCL. They are used by the assembler. They may be loaded from **hiflow.** Since the purpose of these packages is fast execution, we also include minor variations of the control words that are more efficient in some instances. The source code is provided on disk in file **hiflow,** enabling you to create control words tailored to your particular needs.

The primary variation is a set of words that does not pop the stack when testing a conditional. Recall that in IBCL the top of the stack is in the **bx** register. This is the register that the conditionals check. Its contents are usually lost when the next element is popped into it. Often the flag word is used within an **if** or a loop and the control word would have to be preceded by a **dup** to provide an extra copy. Another variation is easy to write but too numerous for including a complete set. For these the condition refers to the flags left by the previous instruction, for instance:

```
begin. cx bx cmp. while-below. [bx] 0 ib mov. bx inc. repeat.
```

The condition name is taken from the related conditional jump name.

All code class high-level flow control words end with a period to maintain consistency with the assembler. The flow control words that do not drop the word on top of the stack include a minus sign. These words use the stack for branch and target addresses and for tags to check matching errors. The compile time stack beyond the most recent flow control word should be considered inaccessible. The execute time stack is used as noted in the individual words. Nesting on all of these words is limited only by stack space.

First word	Terminators	NOTE: bx is the top stack word. sp points to the second.
if. tt	then.	Code tt executes if bx is nonzero. Next stack word pops into bx.
if. tt	else. ff then.	Code tt executes if bx is nonzero. Code ff executes if bx is zero. Next stack word pops into bx.
if-t. tt	then.	Code tt executes if bx is nonzero. bx unchanged.
if-t. tt	else. ff then.	Code tt executes if bx is nonzero. Code ff executes if bx is zero. bx unchanged.
begin. bb	again.	Infinite loop, stack not used.
begin. bb	until.	Code bb repeatedly executes until bx is nonzero entering until. Next stack word is pops into bx on every pass through until.

begin. bb	until-t.	Like until., but bx preserved.
begin. cc	while. bb repeat.	Loop exits when bx is zero entering while. Next stack word pops into bx on every pass through while.
begin. cc	while-t. bb repeat.	Like while. except bx preserved.
begin. cc	while-below. bb repeat.	Like while-t. except loop exits when carry flag is clear.
do.		Push si, then di, onto return stack. Pop initial value from data stack (bx) into di, pop limit into si, pop third stack word into bx.
do-.		Like do. but initial value remains on stack (in bx). Limit is still removed from stack.
		Either form of do. may be used with any of the following do loop terminators.
	loop.	Increment di. If di less than si, branch back to code after do. Otherwise pop the top of the return stack into di, and next word into si, and continue executing after loop.
	+loop.	Like loop. but the value in the bx register is added to di instead of 1. If the increment is negative, the loop is repeated until the count is less than the limit.

	The next data stack word is popped into bx.
+loop-.	Like **+loop**. but the increment is retained in bx.
/loop.	Add value in bx to di. If di is below limit (unsigned), branch back to code after do. Otherwise pop the top of the return stack into di, and next word into si, and continue executing after /loop. The next data stack word is popped into bx.
/loop-.	Like /loop. but the increment is retained in bx.

Words to use inside do loops.

leave.	Set di to si, terminating loop at next pass through loop terminator.
i.	Push count onto data stack. (Push bx, then copy di to bx)
j.	Push count of next outer do loop onto data stack. (Push bx, then copy top of return stack to bx)

Processor Control

These instructions have no operands, the operator is sufficient to generate the code.

	clocks	
hlt.	2	Cause cpu to enter halt state. This state may be cleared by an enabled external interrupt or by a system reset.
lock:	2	Bus lock prefix. The processor's bus-lock signal will be asserted for the duration of the operations caused by the following instruction. Used in multiprocessor systems to control access to shared resources. For instance, the following sequence will wait for some resource to become available, signaled by a non-zero value in the variable free?.

```
code  wait          >>> notyet.
                                        al  al  xor.
        lock:   free?                   +[] al  xchg.
                                        al  al  test.
                                        jz  notyet.end-code
code  release   free?                   +[I  0  ib  mov.end-code
```

nop.	3	No operation. Implemented as ax ax xchg.
clc.	2	Clear carry flag.
stc.	2	Set carry flag.
cmc.	2	Complement carry flag.

cld.	2	Clear direction flag, set to ascending.
std.	2	Set direction flag, set to descending.
cli.	2	Clear interrupt flag, disable maskable external interrupts.
sti.	2	Set interrupt flag, enable maskable external interrupts.

Interrupts

Interrupts may be generated by software to transfer control through any of 256 vectors at the bottom of memory. The address of the vector transferred through is four times the interrupt number. For instance, the vector for interrupt 3 is at location 12 decimal. The first word at this location holds the segment paragraph number into which control is transferred. The second word holds the offset into that segment of the entry of the interrupt handler. See the IBM Technical Reference Manual or its equivalent for your computer for system interrupt usage. Interrupts between 128 and 255 are listed in the BIOS memory map as available for independent use.

int. Push the flag registers on the stack, clear tf and if (trap and interrupt flags), and transfer control as if for indirect intersegment call through vector element.

```
                                 clocks
                                 -----
   interrupt-number     int.     50
                  3     int.     51      (breakpoint interrupt)
```

into. Generate interrupt 4 if overflow flag (of) is set.

```
                        into.    52      (overflow flag set)
                                  4      (overflow flag not set)
```

iret. Perform intersegment return, then pop next stack word into flags. This restores the flags to the values they had before the interrupt which caused entry into this routine.

```
                        iret.    24
```

CoProcessor Support

esc. This instruction forms the base for all NEC 72191 numeric coprocessor instructions. It requires a 6 bit number (0 through 63) on the stack to use as the opcode for the coprocessor instruction. The instruction also requires either a memory operand (**[xx] or +[xx])** or a second number on the stack. The second number selects either one of the eight NEC 72191 stack registers or one of eight specific commands for the given opcode. In both cases the possible values are 0 through 7. The top register of the NEC 72191 stack is 0, the one under it is 1.

```
      memory  0..63  esc.
or      0..7  0..63  esc.
```

You will not normally need to use this instruction since complete NEC 72191 instruction mnemonics are provided in the 72191asm file.

wait.
fwait. These instructions are the same. The fwait. version is just a reminder that the floating point processor is the cause of the delay. The primary processor must not access any memory location that the coprocessor is attempting to read or to write. The primary processor is free to read or write other memory locations. The coprocessor must not be given a new instruction until the previous one has been completed. This last requirement is met automatically by instructions using NEC 7219 1 mnemonics from the 72191asm file — they all code an fwait. preceding each instruction.

Omission of fwait. wherever needed to prevent memory conflicts, can produce random bugs which are extremely difficult to detect. The logic may be flawless, but if the result isn't ready it will be wrong.

NEC 72191 Architecture

The NEC 72191 is an arithmetic coprocessor similar to but more advanced than the Intel 8087. Your MacBus is socketed to accept this coprocessor, but it is not available at this time. IBCL, however, can use the coprocessor. Two IBCL source files on your distribution diskette (72191ibcl and 72191asm) provide high and low level support, respectively, for the coprocessor. The next section, NEC 72191 Instructions, describes the IBCL words contained in these two files for coprocessor support.

Data registers

The eight 80-bit data registers of the NEC 72191 are organized as a stack. If a ninth data word is forced onto this stack, both its value and the value of the bottom word on the stack are lost. This happens because both would occupy the same register, so the NEC 72 191 marks that register empty. Data in the NEC 72191 data registers is always in an internal temporary real format. The current top of **fstack** may occur in any of the registers, and all addresses are relative to the current top of stack.

Status Word

The status word may be stored in memory and examined to determine the current state of NEC 72191 operations including the result of comparisons.

bit	definition
15	Set if previous instruction still executing (busy).
14	c3 of condition code.
13.11	Absolute register address of top of stack.
10.8	c2 through c0 of condition code.
7	The interrupt request bit is set if any of the following exceptions are set.
5	precision
4	underflow

3	overflow
2	zerodivide
1	denormalized operand
0	invalid operation

Control Word

A new control word may be written from memory to the NEC 72191 to control its response to various conditions. Sometimes this word is first read from the NEC 72191 into memory where only the bits relevant to the required change are altered. It is then loaded from memory back into the NEC 72191.

In case of alternatives, the default is listed first.

bit	definition
12	infinity control
	0 projective closure (infinities equivalent)
	1 affine closure (-infinity < +infinity)
11,10	rounding
	00 round to nearest or even
	01 round toward -infinity
	10 round toward +infinity
	11 truncate toward zero
9,8	precision
	11 64 bits
	00 24 bits
	10 53 bits
7	interrupt mask
5	precision
4	underflow
3	overflow
2	zerodivide
1	denormalized operand
0	invalid operation

Tag Word

The tag word contains a 2-bit code for each of the eight data registers. The first 2 bits are for register 7 (absolute) and the last 2 bits for register 0.

00	valid nonzero
01	valid zero
10	special (zero, unnormal, infinity, indefinite, or not a number)
11	empty

Registers are empty when the coprocessor is initialized and are assigned or freed as numbers are pushed and popped from the stack.

Data Types

All numbers are stored in the NEC 72191 in the same Temporary Real format. Instructions that require a memory reference will read or write a number at the memory location using one of the following data types.

Note: The packed decimal, long integer, and temporary real data types are used only by **fld.** and **fstp.** load and store instructions.

Type	# bits	range
Word Integer	16	-32768 through 32767
Short Integer	32	-2,147,483,648 through 2,147,483,647
Long Integer	64	-9*10**18 through 9*10**18
Packed Decimal	80	18 digits (signed)

		exponent #bits	bias	
Short Real	32	8	127	6 or 7 digits 10**-37 to 10**38
Long Real	64	11	1023	15 or 16 digits 10**-307 to 10**308
Temporary Real	80	15	16383	19 digits 1 O**-4932 to 10**4932

The least significant byte is always at the lowest addressed memory location. All integers are in the usual 2's complement form. Only the high-order bit of highest addressed byte of a packed decimal is significant — it is set for negative numbers. The corresponding bit is also set to indicate negative real numbers, but it is immediately followed by an exponent. The leading bit of the magnitude is explicit in temporary real format, but left understood in the other real formats.

IBCL's scaled decimal package interfaces through the short integer data type. The precision of this interface is more than adequate for entering and printing most numbers, and the decimal point is handled automatically. Intermediate steps could possibly require more precision, and this is easily accomplished using the assembler extensions and longer data types. Be careful to **allot sufficient** space to variables for the various data types.

NEC 72191 Instructions

To use the 72191 from IBCL, you must download one of the IBCL source files (72191ibcl or 72191asm). 72191ibcl provides a set of high level IBCL words which you may use interactively or in colon definitions. You may download this file to MacBus using the interactive IBCLload utility or the language interface function IBCLload. Refer to SECTION ONE - IBCL Source Files for more details.

Before downloading 72191asm, you must download the IBCL assembler contained in the file microasm. 72191asm contains IBCL definitions which allow you to use 72191 operations in your IBCL assembly language definitions. This section documents the coprocessor support facilities provided by these two utility files.

A full complement of NEC 72191 high-level IBCL words are included in the file 72 19 1 ibcl. They include NEC 7219 1 versions of most stack control and memory reference words, as well as comparisons and math from addition and subtraction through trig and log functions. Format conversion to and from the scaled decimal accepted and printed by IBCL is also provided. The scaled decimal package is not needed by 72191ibcl, which duplicates those words which are useful. Duplicated words include:

sc-constant scaler SC! sc@ and sc .

Our stack notation will contain a double colon :: with the NEC 72191 stack always to the right. Remember that the NEC 72191 stack can only hold eight numbers. Some of the transcendental functions need two extra temporary register from the NEC 72191 stack for intermediate calculations.

Data Type Selection

These words set the data type assumed by memory reference instructions. The data type set remains in effect until changed.

i16	Word integer	
r32	Short real	
i32	Short integer	
r64	Long real	
i64	Long integer	fld. and fstp. only
bcd	Binary coded decimal	"
treal	Temporary real	"

Instruction Format

The NEC 72191 mnemonics follow the usual IBCL pattern, and always begin with **f.** They may be preceded by either a single operand or no operand. If an operand is present, it must be of the memory reference type or a single word integer from 0 through 7 addressing one of the 8 NEC 72191 stack registers relative to the top of the NEC 72191 on chip stack. Instructions needing two operands always use the top number on the NEC 72191 stack for one.

```
                Fccccc.
            n   Fccccc.
         [cc]   Fccccc.
offset +[cc]   Fccccc.

                fabs.
            1   fadd.
         [bx]   fsub.
  x +[bp+di]   fdiv.    (x is some variable.)
```

Data Tram fer

fld. Read using memory/fstack operand and push value on fstack. (NEC 72191 stack)

fst. Format number on top of fstack using current data type and store at memory/fstack location.

fstp. Format number on top of fstack using current data type and store at memory/fstack location. Then pop fstack (throw away top number).

fxch. Exchange top of fstack with number at fstack location.

	stack	clock count range i64	bcd	treal
memory fld.		60-68+	290-310+	53-65+
memory fstp.		94-105+	520-540+	52-58+
fstack fld.	17-22			
fstack fst.	15-22			
fstack fstp.	17-24			
fstack fxch.	10-15			

	i16	r32	i32	r64
memory fld.	46-54+	38-56+	52-60+	40-60+
memory	fst.	80-90+	84-90+	82-92+96-104+
memory	fstp.	82-92+	86-92+	84-94+98-106+

Comparison

The compare, test, and examine commands return their results by setting the condition code bits in the NEC 72191 Status word.

fcom.	Compare number on top of fstack with memory/fstack operand.
fcomp.	Compare as for **fcom.**, then pop top number from fstack.
fcompp.	Compare top two numbers on fstack and then pop both.
ftst.	Compare top fstack number to zero.

Set condition codes as follows:	c3	co
Top number > operand	0	0
Top number < operand	0	1
Top number = operand	1	0
not a number or projective infinity	1	1

fxam. Examine number on top of fstack and set condition codes **c3..c0** to indicate type.

0 + unnormal	1 invalid	2 - unnormal	3 invalid
4 + normal	5 + infinity	6 - normal	7 - infinity
8 + 0	9 empty	10 - 0	11 empty
12 invalid	13 empty	14 invalid	15 empty

Unnormal numbers are so small that normalizing them would give a negative (unrepresentable) biased exponent. Instead, the biased exponent is set to zero and magnitude bits are right shifted to compensate. If the number is small enough, all bits are shifted out the end of the register and it becomes an ordinary positive or negative zero. Slightly larger numbers retain some bits, but since the full register width is not being used some precision is lost.

				clock count range		
			i16	r32	i32	r64
memory	fcom.		72-86+	60-70+	78-91+	65-75+
memory	fcomp.		74-88+	63-73+	80-93+	67-77+
fstack	fcom.	40-50				
fstack	fcomp.	45-52				
	fcompp.	45-55				
	ftst.	38-48				
	fxam.	12-23				

Constants

Push the specified constant onto the fstack.

		clock count
fldz.	zero	11-17
fldl.	one	15-21
fldpi.	**pi**	16-22
fldl2t.	**log base 2** of 10	16-22
fldl2e.	**log base 2 of e**	15-21
fldlg2.	**log base 10 of 2**	18-24
fldln2.	**log base e of 2**	17-23

Arithmetic

fadd.	**fadd".**	faddp.			
fsub.	fsub".	fsubp.	fsubr.	fsubr".	fsubrp.
fmul.	fmul".	fmulp.			
fdiv.	fdiv".	fdivp.	fdivr.	fdivr".	fdivrp.

Add, subtract, multiply, or divide the number on top of the fstack and the memory/fstack operand. The **suffix** determines the destination, order of operands, and whether or not to pop the top of the stack.

	Subtract from/divide into number on top of fstack.
none	result returned in top of fstack
"	result returned to mem/fstack operand
p	result returned to mem/fstack operand fstack popped
	Subtract from/divide into mem/fstack operand
r	result returned in top of fstack
r"	result returned to mem/fstack operand
rp	result returned to mem/fstack operand and fstack popped

		stack	clock count range i16	r32	i32	r64
memory	fadd--.		102-137+	90-120+	108-143+	95-125+
memory	fsub--.		102-137+	90-120+	108-143+	95-125+
memory	fmul--.		124-138+	110-125+	130-144+	112-168+
memory	fdiv--.		224-238+	215-225+	230-243+	220-230+
fstack	fadd--.	70-100				
fstack	fsub--.	70-100				
fsteck	fmul--.	90-145				
fstack	fdiv--.	193-203				

The following words use the top one or two numbers on the fstack. The result replaces the top number (x), but any second number (y) remains on the stack unchanged.

	clock count	
fsqrt.	180-186	Replace top number with its square root. -0 <= x <= +infinity
fscale.	32-38	Scale top number by second. i.e., Add second (rounded toward zero to an integer) to the exponent of the first. -2**15 <= y <= 2**15, y integer
frndint.	16-50	Round top number to integer.
fxtract.	27-55	Replace top number with its exponent, then push significand onto fstack. (an fscale. at this point would put the number back together, except that a copy of the exponent would remain.)
fabs.	10-17	Absolute value of top number.
fchs.	10-17	Change sign of top number.
fprem.	15-190	Partial remainder of top/second.

fprem. is usually used to reduce the argument of a cyclic function to its fundamental range. (to reduce an angle in radians to the range of 0.0 through pi/4) To allow more rapid response to system interrupts, the maximum clock count of this instruction has been kept low by limiting the range reduction for one step to 2**64. If the top number was (33.9 * 2**64 * pi/4) and the second was pi/4, one execution would leave (33.9 * pi/4), which is still not in the range of the fptan. function. A second execution would leave the desired result, (.9 * pi/4). Bit c2 of the status word will be set if the operation must be repeated.

The three lowest order bits of the quotient are returned in status word bits c0, c3, and cl, where cl is the least significant bit and c0 the most significant. c3 occupies bit 14 of the status word, while c2 through c0 occupy bits 10, 9 and 8. If you store the status word out to memory to check these bits, remember to use an fwait. after the store and before trying to load the status word into an NEC V50 register or otherwise attempting to use it.

Transcendental

The transcendental instructions also use the top one or two numbers on the fstack. x represents the number on top of the fstack, and y the number under it.

In all cases, the arguments are destroyed and only the results are returned on the fstack.

ALL ARGUMENTS MUST BE VALID, NORMALIZED, AND IN RANGE.

	clocks	
fptan.	30-540	Tangent. Angle (x) must be in radians with $0 < x < \pi/4$. This function does not really return the tangent, but the length of the opposite and adjacent side of a triangle with radius about 1.5. The length of the adjacent side is stored on top so the tangent could be calculated with: 1 fdivrp.
fpatan.	250-800	Arctangent. Arguments are lengths of the opposite and adjacent sides as returned by fptan. 0 < y(opposite) < x(adjacent) < infinity
f2xm1.	310-630	Exponential. (2 ** x) - 1 x must be between 0 and 0.5 inclusive. Offsetting the result by minus one preserves precision when x is very near zero and the result would be very near one. To calculate other exponents, use the relation **x**y** = **2**(y * LOG2(x))**
fyl2x.	900-l 100	Y * log2(x) 0 < x < +infinity -infinity < y < +infinity Note: Neither x nor y can be infinite.

fyl2xp1.	$y * \log2(x+1)$ 700- 1000	$0 <= x <= 1-\mathrm{sqrt}(0.5)$ $-\mathrm{infinity} < Y < +\mathrm{infinity}$ The inverse of **f2xm1.**

Processor Control

finit.	2-8	Initialize NEC 72 19 1 Control word: projective infinity, round to nearest, 64 bits, interrupts disabled, all exceptions masked Status word: not busy, empty stack, no interrupt, no exceptions Tag word: all tags 11 for empty register Data registers unchanged
fnop.	IO-16	No operation.
fwait.	3+5n	Wait until NEC 72191 has completed instruction.
fldcw.	7-14+	Load control word into NEC 72 191. (2 bytes) memory fldcw.
fstcw.	12-18+	Store NEC 72191 control word into memory. (2 bytes) memory fstcw.
fstsw.	12-18+	Store NEC 72191 status word into memory. (2 bytes) memory fstsw.
feni.	2-8	Enable interrupts. Clear interrupt enable mask bit in Control word.
fdisi.	2-8	Disable interrupts. Set interrupt enable mask bit in Control word.

fclex.	2-8	Clear exception, interrupt and busy flags.
flncstp.	6-12	Increment fstack pointer. This makes a register within the fstack the temporary top of fstack. If you pop numbers while at this temporary top of fstack, you could leave empty registers within your fstack. Use with care.
fdecstp.	6-12	Decrement fstack pointer.
ffree.	9-16	Mark indicated register empty. fstack# ffree.
fstenv.	**40-50+**	Store environment. Control word, status word, tag word, instruction pointer, operand pointer. (14 bytes) memory fstenv.
fldenv.	**35-45+**	Load environment. memory fldenv.
fsave.	**197-207+**	Save complete NEC 72191 state in 94 byte area with environment at beginning followed by data registers. memory fsave.
frstor.	**197-207+**	Restore NEC 72191 state. memory fsave.

SEGMENT MANAGEMENT

IBCL provides two mechanisms for manipulating the full one megabyte address range of the NEC V50. This section documents the segment management system which is designed for use prior to compilation of your program. The heap manager, which is designed for use during execution of your program, is discussed in the section "HEAP MANAGEMENT WORDS."

IBCL reserves areas of memory for various purposes. For instance, IBCL reserves a large area for definitions of new words you define. In IBCL terminology we say that the definition segment is not completely full. It does, however, have a default size limit that is less than 64K. If you try to compile a large program you may run out of dictionary space.

The IBCL words discussed in this section allow you to enlarge IBCL's reserved areas prior to compiling your program. Include the necessary segment management word sequence at the beginning of your program.

These words conflict with the heap manager. Once you have initialized the heap, you may no longer use segment management words.

Relocation Tools

The basic segment relocation tool is **copy-seg. A** new segment for the destination must be created first unless an old one can be reused. The size of the destination segment must be sufficient to hold the portion of the source segment below its working end. The destination segment may be larger or smaller than the source. To create a new segment and copy an old one into it, type:

```
size-of-new-segment   segment   name-of-new-segment
name-of-source-segment   name-of-new-segment   copy-seg
```

Another word will automatically allocate space for a new segment, move the contents of the old segment to it, and reuse the old segment descriptor and name for the new location. The descriptor is relinked to maintain a sequential list corresponding to the physical memory. (First descriptor in the list is for the top segment, last descriptor is for the lowest.) The space occupied by the segment is added to the space allocated to the segment below it. This word is primarily used by the special system segment move words.

```
new-size segment-descriptor move-seg
```

The new size is left on the stack and the segment descriptor is replaced by the new first paragraph number.

System segments cannot be summarily moved. Other actions must take place to maintain continuity of control. These segments include the definition list segment lists, the stack segment stacks, and all vocabulary segments including ibcls. The code segment **codes** is never moved. Its size may be increased up to 64K bytes by moving the segments above it higher.

Additional definition list segments may be created by typing:

```
size-of-new-list-segment make-list name-of-new-segment
```

Control may be transferred to an alternate definition list segment by typing:

```
alternate-segment-descriptor use-list
```

Alternate list segments should only be created from the original system list segment lists. When in the alternate segment, its descriptor will use the name lists. Return to the system list segment is accomplished by again typing:

```
alternate-segment-descriptor use-list
```

The first action on entering an alternate list segment should be to create a new vocabulary for words created in it. Words from a single vocabulary should always refer to a single definition list segment. The task and forget-task pair will not work if used across multiple list segments.

The following words are used to move system segments and alter their size:

```
new-size new-voc       (moves context vocabulary segment)
new-size new-stack
new-size new-list
```

In each case the new size is checked against the working end of the segment and the operation aborts if it is too small. The contents of the old segment are moved to the new segment at the start of free memory, and the old segment descriptor and name are reused. The descriptor is relinked to maintain a sequential list corresponding to the physical memory. The first descriptor in the list is for the highest segment, last descriptor is for the lowest. The space formerly occupied by the segment is added to the space allocated to the segment below it.

Although segments may be moved at any time, it is advisable to plan each move carefully. Careless moving can leave gaps in the memory map that are unreachable by the segment management utilities. More general memory management operations are provided by the heap manager.

GPIB Control Words

IBCL provides a complete set of words for manipulation of the IEEE-488 (GPIB) bus. This section documents the IBCL GPIB word set. For a complete explanation of GPIB operations, see APPENDIX B.

IBCL has two types of words that manipulate the GPIB: device level words and MacBus GPIB port level words. When you use device level words IBCL automatically computes and sends the required command strings over the GPIB. IBCL maintains a table of configuration information for each device. This information defines the characteristics of a particular device, including its GPIB address, a time limit for data transfers, and any end-of-string modes.

When performing I/O operations, IBCL takes the device's GPIB address from the appropriate configuration table and its own GPIB address from the MacBus interface configuration table and sends these out as GPIB command bytes. The configuration tables may be changed with IBCL words documented later in this section.

To perform device level operations, MacBus must be CIC.

The following is a description of each GPIB-related IBCL word.

GPIB Status Variables

Some of the GPIB words documented so far use global variables to reflect their status. This technique allows easy access to operation results without cluttering up IBCL's stack. The global IBCL variables are:

ibret (-- a)	contains return value from last operation.
iberr (-- a)	contains error status of last operation.
ibcnt (-- a)	contains byte count of last transfer.
ibppr (-- a)	contains response from last parallel poll
ibspr (-- a)	contains response from last serial poll.

Associating Names with GPIB Devices

Device level IBCL words require a device number as a stack parameter. A device's configurations table stores its name, however, IBCL lets you use this name in device calls instead of having to remember the device number. The IBCL word ibfind scans the device table for a name and creates an IBCL constant of this name using the device number as the value. For instance, if you have used the IBCONF utility to give device 3 the name 'plotter', you may issue the following ibfind call:

```
ibfind plotter
```

Now you may use the word plotter anywhere in an IBCL program and the result will be the same as if you had used the number 3 instead.

Note that **ibfind** MAY NOT BE COMPILED INTO AN IBCL WORD! You MUST issue any ibfind commands outside of colon definitions. Once device names are defined, they may be used within colon definitions.

Device Level Words

The following pages describe the device level words.

NAME

clr - send selected device clear (SDC)

SYNOPSIS

d clr

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

clr sends the message selected device clear (SDC). SDC reinitializes all device functions. **clr** sends the following commands:

- Unlisten (UNL)
- Listen address of the device
- Secondary address of the device if applicable
- Selected Device Clear (SDC)
- Unlisten (UNL)

EXAMPLE

1. To clear device plotter:

```
plotter clr
```

NAME

110 - place all devices in local lockout state

SYNOPSIS

d 110

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

110 sends the message LLO which places all devices in the local lockout state. This usually inhibits recognition of front panel input.

llo sends the following command:

- Local Lockout (LLO)

EXAMPLE

1. To place device plotter in local lockout state:

```
plotter llo
```

SEE ALSO

loc.

NAME

loc - go to local mode

SYNOPSIS

d **loc**

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

loc is used to move devices temporarily from a remote program mode to a local mode. A device enters remote mode when the REN line is asserted and the device detects its listen address.

loc places the indicated device in local mode by sending the command sequence:

- Unlisten (UNL)
- Listen address of the device
- Secondary address of the device if applicable
- Go To Local (GTL)
- Unlisten (UNL)

EXAMPLE

1. To return device plotter to local state:

```
plotter loc
```

SEE ALSO

110.

NAME

onl - place device online or offline

SYNOPSIS

d f **onl**

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

f is a true/false value indicating online/offline.

onl reinitializes all software as though bringing the device online for the first time.

Call **onl** with **f** non-zero to reset a device software to its power-on state. Call **onl** with f zero only when finished with a device.

EXAMPLE

1. To disable device plotter:

```
plotter 0 onl
```

SEE ALSO

ibfind.

NAME

pct – pass control

SYNOPSIS

d **pct**

DESCRIPTION

d is a device number or device name used in an ibfind call.

pct passes Controller-In-Charge (CIC) authority to the specified device. The MacBus GPIB port automatically goes to an idle state. The function assumes that the device has Controller capability.

pct sends the following commands.

- Talk address of the device
- Secondary address of the device, if applicable
- Take Control (TCT)

EXAMPLE

1. To pass control to device pcxt:

```
pcxt pct
```

NAME

ppc - parallel poll configure

SYNOPSIS

d v **ppc**

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

v is a valid parallel poll enable/disable command.

ppc enables or disables the device from responding to parallel polls.

ppc sends the following commands:

- Unlisten (UNL)
- Listen address of the device
- Secondary address of the device, if applicable
- Parallel Poll Configure (PPC)
- Parallel Poll Enable (PPE) or Disable (PPD)
- Unlisten (UNL)

Each of the 16 PPE messages specifies the GPIB data line **(DIO1** through D108) and sense (one or zero) that the device must use when responding to the Identify (IDY) message during a parallel poll. The assigned message is interpreted by the device along with the current value of the individual status (ist) bit to determine if the selected line is driven true or false. For example, if **PPE=0x64, DIO5** is driven true if **ist=0** and false if ist=l. And if **PPE=0x68, DIO1** is driven true if ist=l and false if **ist=0**. Any PPD message or zero value cancels the PPE message in effect.

Which PPE and PPD messages are sent and the meaning of a particular parallel poll response are all system dependent protocol matters to be determined by the user.

EXAMPLES

1. To configure device dvm to respond to a parallel poll by sending data line D103 true if IST=O:

   ```
   dvm 62 ppc
   ```

2. To configure device dvm to respond to a parallel poll by sending data line **DIO1** true if IST= 1:

   ```
   dvm 68 ppc
   ```

3. To cancel the parallel poll configuration of device dvm:

   ```
   dvm 0 ppc
   ```

SEE ALSO

rpp.

NAME

rd – read data from GPIB

SYNOPSIS

d buf cnt rd

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

buf is the long address of the buffer to use (buf might have been created using **alloc**).

cnt specifies the number of bytes to read from the GPIB.

rd reads cnt bytes of data from a GPIB device. The following steps are performed:

1. The device is addressed to talk and the MacBus GPIB port to listen, if not already addressed to do so.
2. The MacBus GPIB port reads the data from the device.
3. Attention is reasserted.

When rd returns, ibcnt is the actual number of data bytes read from the device; and iberr is the first error detected if iberr is non-zero.

rd operation terminates on any of the following events:

- When cnt bytes have been read
- Error is detected
- Time limit is exceeded
- END message is detected
- eos character is detected (if this option is enabled)

After termination, ibcnt contains the number of bytes read.

EXAMPLES

1. To read hex 56 bytes of data from device tape:

   ```
   tape buf 56 rd
   ```

NAME

rpp - conduct a parallel poll

SYNOPSIS

d rpp

DESCRIPTION

d is a device number or device name used in an ibfind call.

rpp conducts a parallel poll of previously configured devices by sending the IDY message (ATN and EOI both asserted). If **ibret** contains a non-zero value upon completion, ibrpp contains a valid response.

EXAMPLE

1. To remotely configure device lcrmtr to respond positively on D103 if its individual status bit is 1, and then parallel poll all configured device:

```
lcrmtr 6A ppc
lcrmtr rpp
```

SEE ALSO

ppc.

NAME

rsp - conduct a serial poll

SYNOPSIS

d rsp

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

rsp is used to serially poll one device and obtain its status byte or to obtain a previously stored status byte. If the 0x40 bit of the response is set, the status response is positive, i.e., the device is requesting service.

If automatic serial polling is enabled (default configuration), devices are polled the instant they request service. Positive responses are saved in a queue. If a device has been polled, the RQS bit of that device's status word is set, and in this case a call to **rsp** returns the previously acquired status byte. If the RQS bit of the status word is not set when **rsp** is called, the device is serially polled. Upon completion, if **ibret** contains a non-zero value, **ibspr** contains a valid serial poll response.

The interpretation of the response in **ibspr,** other than the RQS bit, is device-specific. For example, the polled device might set a particular bit in the response byte to indicate that it has data to transfer, and another bit to indicate a need for reprogramming. Consult the manufacturer's documentation for the device for interpretation of the response byte.

EXAMPLE

1. To obtain the Serial Poll response byte from device tape:

```
tape rsp
```

SEE ALSO

wait, wrqs.

NAME

trg - Send device trigger

SYNOPSIS

d trg

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

trg addresses and triggers the specified device, then unaddresses all devices on the GPIB.

trg sends the following commands:

- Unlisten (UNL)
- Listen address of the device
- Secondary address of the device, if applicable
- Group Execute Trigger (GET)
- Unlisten (UNL)

The response to a trigger is device dependent.

EXAMPLE

1. To trigger device analyz:

```
analyz trg
```

NAME

wait - wait for selected events

SYNOPSIS

d mask wait

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

The mask bit is set to wait for the corresponding event to occur.

wait is used to monitor the events selected in **mask** and to delay processing until any of them occur. These events and bit assignments are shown below:

Device Wait Mask Layout			
Mnemonic	**Bit Pos**	**Hex Val**	**Description**
TIMO	**14**	**4000**	Time limit exceeded
END	13	2000	END detected
RQS	11	800	Device requesting service

Disabling timeouts should be done only when it is certain the selected event will occur. **wait** always returns when the time limit is exceeded regardless of whether the TIMO bit is specified. To disable timeouts, use the configuration control words (described later in this section) to set the time limit to zero.

EXAMPLES

1. To wait for zero:

```
logger 0 wait
```

2. To wait for device logger to request service or timeout:

```
logger 4800 wait
```

3. To wait indefinitely for device logger to request service:

```
0 logger d_tmo setd
logger 800 wait
```

SEE ALSO

rsp, wrqs.

NAME

wrqs - wait for specific status byte

SYNOPSIS

d mask val wrqs

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

mask defines the specific status bits of interest.

val defines the desired bit configuration.

wrqs will wait for a device to request service with a status byte that matches the mask and value. The serial poll queue is first examined to see if a match is found.

A match is made if a status byte, stb, is received such that:

$$(stb \mathbin{\&} mask) == value$$

This technique allows waiting for specific bits in a status byte. An exact match of the value is always required.

EXAMPLES

1. To wait for device rx3 to request service with the byte hex 45:

```
rx3 FF 45 wrqs
```

2. To wait for device rx3 to request service with any byte with either bit 2 or bit 3 asserted:

```
rx3 OC OC wrqs
```

SEE ALSO

rsp, wait.

NAME

wrt - write data to GPIB

SYNOPSIS

d buf cnt wrt

DESCRIPTION

d is a device number or device name used in an **ibfind** call.

buf contains the data to be sent over the GPIB (buf might have been allocated using **alloc**).

cnt specifies the number of bytes to be sent over the GPIB.

wrt writes cnt bytes of data to a GPIB device.

The following steps are performed:

1. The device is addressed to listen and the MacBus GPIB port to talk, if not already addressed to do so.
2. The MacBus GPIB port writes the data to the device.
3. Attention is reasserted.

When **wrt** returns, ibcnt is the actual number of data bytes written to the device; and iberr is the first error detected if iberr is non-zero.

wrt terminates on any of the following events:

- When cnt bytes have been written
- Error is detected
- Time limit is exceeded

After termination, ibcnt contains the number of bytes written.

The double quote **(")** places the text, up to the closing quote, in memory.

The double quote word leaves the long address and string length on the stack and is thus ideal for use with wrt. For instance, “abc” leaves the long address of the string and a count of 3 on the stack.

EXAMPLES

1. To write 10 bytes of instructions to device dvm:

```
dvm " F3R1X5P2G0" wrt
```

MacBus GPIB Port Level Words

MacBus GPIB Port level words are used to manipulate the MacBus GPIB port directly. These words are used in situations that require greater flexibility than the device functions provide.

Such situations include:

— Anytime MacBus is not CIC.

— A Group Execute Trigger (GET) involving more than one device.

— A data transfer between two devices without MacBus participating.

— Waiting for any device to request service (see **wait).**

To use these words, you must be familiar with GPIB protocol. It is your responsibility to perform all addressing and unaddressing of the GPIB.

NAME

bcac - become active controller

SYNOPSIS

v bcac

DESCRIPTION

If v is non-zero, MacBus takes control synchronously with respect to data transfer operations; otherwise, MacBus takes control immediately (and possibly asynchronously).

It is generally not necessary to use the **bcac** word. Words such as **bcmd** and **brpp,** which require that MacBus take control, do so automatically.

To take control synchronously, MacBus waits before asserting the ATN signal so that data being transferred on the GPIB will not be corrupted. If a data handshake is in progress, the take control action is postponed until the handshake is complete; if a handshake is not in progress, the take control action is done immediately. Synchronous take control is not guaranteed if an **brd** or **bwrt** operation completed with a timeout or error.

Asynchronous take control should be used in situations where it appears to be impossible to gain control synchronously (e.g., after a timeout error).

The ECIC error results if MacBus is not CIC.

EXAMPLES

1. To take control immediately without regard to any data handshake in progress:

```
1 bcac
```

2. To take control synchronously and assert ATN following a read operation:

```
1 bcac
```

NAME

bcmd - send command message to GPIB

SYNOPSIS

buf cnt **bcmd**

DESCRIPTION

buf is the long address of a buffer containing the commands to be sent over the GPIB.

cnt specifies the number of bytes to be sent over the GPIB.

bcmd is used to transmit interface messages (commands) over the GPIB. These commands include device talk and listen addresses, secondary addresses, serial and parallel poll configuration messages, and device clear and trigger instructions.

bcmd is NOT used to transmit programming instructions to devices; programming instructions and other device dependent information are transmitted with brd and bwrt.

bcmd terminates on any of the following events:

- All commands are successfully transferred
- Error is detected
- Time limit is exceeded
- Take Control (TCT) command is sent
- Interface Clear (IFC) message is received from the System Controller (not MacBus)

After termination, the **ibcnt variable** contains the number of command bytes sent.

An ECIC error results if the MacBus GPIB port is not CIC. If the MacBus GPIB port is not Active Controller, it asserts ATN prior to sending the command bytes. The MacBus GPIB port remains Active Controller afterward.

EXAMPLES

In the following examples, GPIB commands and addresses are coded as printable ASCII characters. When the hex values to be sent over the GPIB correspond to printable ASCII characters, this is the simplest means of specifying the values. APPENDIX A contains conversions of hex values to ASCII characters.

1. To unaddress all Listeners with the Unlisten command (ASCII ?) and address a Talker at 0x46 (ASCII F) and a Listener at 0x31 (ASCII 1):

```
" ?F1" bcmd
```

2. Same as Example 1 except the Listener has a secondary address of 0x6E (ASCII n):

```
" ?F1n" bcmd
```

NAME

bgts - go from active controller to standby

SYNOPSIS

v bgts

DESCRIPTION

v is the type of go-to-standby.

bgts causes MacBus to go to the Controller Standby state and to unassert the ATN signal if it is the Active Controller. **bgts** permits GPIB devices to transfer data without MacBus being a party to the transfer.

It is generally not necessary to use **bgts.** Functions such as **brd** and **bwrt,** which require that MacBus go to standby, do so automatically.

If v is non-zero, MacBus shadows data transfer handshakes as an Acceptor, and when the END message is detected, MacBus enters a Not Ready For Data (NRFD) handshake **holdoff** state on the GPIB. If v is zero, no shadow handshake or **holdoff** is done.

If the shadow handshake option is activated, MacBus participates in data handshake as an Acceptor without actually reading the data. It monitors the transfers for the END message and holds off subsequent transfers. This mechanism allows MacBus to take control synchronously on a subsequent operation such as **bcmd** or **brpp.** 1 bgts should always be followed by a wait for END (Example 2).

The ECIC error results if MacBus is not CIC.

EXAMPLES

1. To turn the ATN **line off:**

```
0 bgts
```

2. To turn the ATN line off with bgts after unaddressing all listeners with the Unlisten (ASCII ?) command, addressing a talker at 0x46 (ASCII F), and addressing a listener at 0x31 (ASCII 1) to allow the talker to send data messages:

```
" ?F1" bcmd
1 bgts
6000 bwait
```

SEE ALSO

bcac, bcmd, bwait.

NAME

bist **-** set or clear individual status bit (IST)

SYNOPSIS

v bist

DESCRIPTION

v is the sense of the IST bit.

If v is non-zero, the individual status bit is set. If v is zero, the bit is cleared.

bist is used when MacBus participates in a parallel poll that is conducted by another device that is the Active Controller. The Active Controller conducts a parallel poll by asserting the EOI and ATN signals which send the Identify (IDY) message. While this message is active, each device that has been configured to participate in the poll responds by asserting a predetermined GPIB data line either true or false, depending on the value of its local IST bit. MacBus, for example, can be assigned to drive the D103 data line true if IST=l and false if IST=O; conversely, it can be assigned to drive D103 true if IST=O and false if IST= 1.

The relationship between the value of IST, the line that is driven, and the sense at which the line is driven is determined by the Parallel Poll Enable (PPE) message in effect for each device. MacBus is capable of receiving this message either locally, via bppc, or remotely, via a command from the Active Controller. Once the PPE message is executed, **bist** changes the sense at which the line is driven during the parallel poll, and in this fashion MacBus can convey a one-bit, device dependent message to the Controller.

EXAMPLES

1. To set the individual status bit:

```
1 bist
```

2. To clear the individual status bit:

```
0 bist
```

SEE ALSO

bppc, brpp.

NAME

bloc – go to local mode

SYNOPSIS

bloc

DESCRIPTION

The MacBus GPIB port is placed in a local state by sending the local message Return To Local (rtl), provided it is not locked in remote mode. **bloc** is used to simulate a front panel Return-To-Local switch when the computer is used to simulate an instrument.

EXAMPLE

1. To return the MacBus GPIB port to local state:

```
bloc
```

SEE ALSO

bsre.

NAME

bonl - place MacBus GPIB port online or **offline**

SYNOPSIS

v **bonl**

DESCRIPTION

If v is non-zero, the MacBus GPIB port is enabled for operation (i.e., online). If v is zero, it is held in a reset, disabled mode (offline).

Taking the MacBus GPIB port **offline** may be thought of as disconnecting its GPIB cable from the other devices.

bonl can also be used to restore the default configuration settings of a MacBus GPIB port. Calling **bonl** with v non-zero when the device or MacBus GPIB port is already online simply has the effect of restoring all configuration settings to their defaults.

EXAMPLES

1. To reset the configuration settings to their defaults:

```
1 bonl
```

2. To disable MacBus GPIB port:

```
0 bonl
```

SEE ALSO

ibfind.

NAME

bppc - parallel poll configure

SYNOPSIS

v bppc

DESCRIPTION

v is a valid parallel poll enable/disable command.

The MacBus GPIB port itself is programmed to respond to a parallel poll by setting its local poll enable (lpe) message to the value of v.

Each of the 16 PPE messages specifies the GPIB data line (DIOI through D108) and sense (one or zero) that the device must use when responding to the Identify (IDY) message during a parallel poll. The assigned message is interpreted by the device along with the current value of the individual status (ist) bit to determine if the selected line is driven true or false. For example, if **PPE=0x64,** D105 is driven true if **ist=0** and false if ist=l. And if **PPE=0x68, DIO1** is driven true if ist=l and false if **ist=0.** Any PPD message or zero value cancels the PPE message in effect.

Which PPE and PPD messages are sent and the meaning of a particular parallel poll response are all system dependent protocol matters to be determined by the user.

The 16 valid PPE messages and the 16 valid PPD messages are found in APPENDIX A.

EXAMPLES

1. To cancel the parallel poll configuration of MacBus:

```
70 bppc
```

2. To configure MacBus GPIB port to respond to a parallel poll by sending data line **DIO8** true if IST=O:

```
67 bppc
```

SEE ALSO
bcmd, bist.

NAME

brd - read characters from GPIB

SYNOPSIS

buf cnt brd

DESCRIPTION

buf is the long address of the buffer to use.

cnt specifies the number of bytes to read from the GPIB.

brd reads **cnt** bytes of data from a GPIB device.

brd attempts to read from a GPIB device that is assumed to already be properly initialized and addressed.

If the MacBus GPIB port is CIC, **bcmd** must be called prior to **brd** to address a device to talk and the MacBus GPIB port to listen. If the MacBus GPIB port is not CIC, the device on the GPIB that is the CIC must perform the addressing.

If the MacBus GPIB port is Active Controller, the MacBus GPIB port is first placed in Standby Controller state (ATN **off)** and remains there after the read operation is completed.

An EADR error results if the MacBus GPIB port is CIC but has not been addressed to listen with bcmd. An EABO error results if the MacBus GPIB port is not CIC and is not addressed to listen within the time limit. An EABO error also results if the device that is to talk is not addressed and/or the operation does not complete for whatever reason within the time limit.

brd operation terminates on any of the following events:

- When cnt bytes have been read
- Error is detected
- Time limit is exceeded
- END message is detected
- eos character is detected (if this option is enabled)

- Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is CIC

After termination, ibcnt contains the number of bytes read. A short count can occur on any of the above events but the first.

EXAMPLE

1. To read 56 bytes of data from a device at talk address 0x4C (ASCII L) and then unaddress it (the MacBus GPIB port at listen address is 0x20 or ASCII blank):

```
" ?L " bcmd      ( address talker and listener )
buf 56 brd       ( read data )
"  ?" bcmd       ( unaddress talker and listener )
```

SEE ALSO
bcmd, beos.

NAME

brpp - conduct a parallel poll

SYNOPSIS

brpp

DESCRIPTION

brpp conducts a parallel poll of previously configured devices by sending the IDY message (ATN and EOI both asserted).

When done, if **ibret contains a non-zero value, ibppr contains a valid** poll response.

An ECIC error results if MacBus is not CIC. If MacBus is Standby Controller, it takes control and asserts ATN (becomes Active) prior to polling. It remains Active Controller afterward.

EXAMPLE

1. To remotely configure a device at listen address 0x23 to respond positively on D103 if its individual status bit is 1, and then parallel poll all configured devices:

```
23 cmd c!l                     ( listen address )
5  cmd 1 + c!l                 ( parallel poll configure )
68 cmd 2 + c!l                 ( parallel poll enable )
3F cmd 3 + c! l                ( unlisten )
cmd 4 bcmd                     ( send command string )
brpp                           ( response in ibppr )
```

SEE ALSO

bcmd, bist, bppc.

NAME

brsc **-** request or release system control (SC)

SYNOPSIS

v brsc

DESCRIPTION

v specifies request or release system control.

If v is non-zero, functions requiring System Controller capability are subsequently allowed. If v is zero, functions requiring System Controller capability are disallowed.

brsc is used to enable or disable the capability of MacBus to send the Interface Clear (IFC) and Remote Enable (REN) messages to GPIB devices using the **bsic** and bsre functions. The MacBus GPIB port must not be System Controller to respond to Interface Clear sent by another Controller.

In most applications, MacBus will always be the System Controller. In other applications, MacBus will never be the System Controller. In either case, brsc is used only if the Macintosh Plus is not going to be System Controller for the duration of the program execution. While the **IEEE-**488 standard does not specifically allow schemes in which System Control can be passed dynamically from one device to another, **brsc** would be used in such a scheme.

EXAMPLE

1. To request to be System Controller if the MacBus GPIB port is not currently so designated:

```
1 brsc
```

NAME

brsv – request service and/or set serial poll status byte

SYNOPSIS

v brsv

DESCRIPTION

v specifies the serial poll response byte.

If the 0x40 bit is set in v, MacBus additionally requests service from the Controller by asserting the GPIB SRQ line.

brsv is used to request service from the Controller using the Service Request (SRQ) signal and to provide a system dependent status byte when the Controller serially polls MacBus.

It is not an error to call **brsv** when MacBus is CIC, although this usage makes sense only if control will be passed later to another device. In this case, the call updates the status byte, but the SRQ signal is asserted only if the 0x40 bit is set and only when control is passed.

EXAMPLES

1. To set the Serial Poll status byte to 0x41, which simultaneously requests service from an external CIC:

```
41 brsv
```

2. To stop requesting service (unassert SRQ):

```
0 brsv
```

3. To change the status byte to 1 without requesting service:

```
1 brsv
```

NAME

bsic - send interface clear **(IFC)**

SYNOPSIS

bsic

DESCRIPTION

bsic causes MacBus to assert the IFC signal for at least 100 ms, provided MacBus has System Controller authority. This action initializes the GPIB and makes the MacBus GPIB port CIC. **bsic** is generally used when you want to become CIC or clear a bus fault condition.

The IFC signal is supposed to reset only the GPIB interface functions of bus devices and is not intended to reset internal device functions. Device functions are reset with the Device Clear (DCL) and Selected Device Clear (SDC) commands. To determine the effect of these messages, consult the device documentation.

EXAMPLE

1. To initialize the GPIB and become CIC at the beginning of a program:

```
bsic
```

SEE ALSO

brsc.

NAME

bsre - set or clear Remote Enable (REN)

SYNOPSIS

v bsre

DESCRIPTION

v specifies set or clear.

bsre turns the REN signal on and off. If v is non-zero, the Remote Enable (REN) signal is asserted. If v is zero, the signal is unasserted. REN is used by devices to select between local and remote modes of operation. REN enables the remote mode. A device does not actually enter remote mode until it receives its listen address.

The ESAC error occurs if MacBus is not System Controller.

EXAMPLES

1. To place a device at listen address 0x23 (ASCII #) in remote mode with local ability to return to local mode:

```
1 bsre          ( set REN to true )
" #" bcmd       ( LAD )
```

2. To exclude the local ability of the device to return to local mode, send the Local Lockout command (0x11), or include it in the command string in Example 1:

```
11 buf c!l      ( send LLO )
buf 1 bcmd

or

1 bsre          ( REN true )
ascii # buf c! l( LAD LLO )
11 buf 1 + c!l
buf2bcn-d
```

3. To return all devices to local mode:

```
0 bsre          ( set REN to false )
```

SEE ALSO
brsc, bsic.

NAME

bwait - wait for selected events

SYNOPSIS

mask bwait

DESCRIPTION

The mask bit is set to wait for the corresponding event to occur.

bwait is used to monitor the events selected in mask and to delay processing until any of them occur. These events and bit assignments are shown below:

Board Wait Mask Layout

Mnemonic	**Bit** Pos	Hex **Val**	**Description**
TIMO	14	4000	Time limit exceeded
END	13	2000	MacBus detected END or EOS
SRQI	12	1000	SRQ on
LOK	7	80	MacBus in Lockout State
REM	6	40	MacBus in Remote State
CIC	5	20	MacBus is CIC
TACS	3	8	MacBus is talker
LACS	2	4	MacBus is listener
DTAS	1	2	MacBus in Device Trigger State
DCAS	0	1	MacBus in Device Clear State

If **mask = 0,** the function returns immediately. This is used to report the current device or MacBus GPIB port state.

If the TIMO bit is 0 or the time limit is set to 0; timeouts are disabled. Disabling timeouts should be done only when it is certain the selected event will occur.

All activity on the MacBus GPIB port is suspended until the event occurs.

EXAMPLES

1. To wait for a service request or a timeout:

```
5000 bwait
```

2. To report the current status:

```
0 bwait
```

3. To wait indefinitely until control is passed from another CIC:

```
20 bwait
```

4. To wait indefinitely until addressed to talk or listen by another CIC:

```
C bwait
```

SEE ALSO

bgts.

NAME

bwrt – write data to GPIB

SYNOPSIS

buf cnt bwrt

DESCRIPTION

buf contains the data to be sent over the GPIB.

cnt specifies the number of bytes to be sent over the **GPIB.**

bwrt writes **cnt** bytes of data to a GPIB device.

bwrt attempts to write to a GPIB device that is assumed to already be properly initialized and addressed.

If the MacBus GPIB port is CIC, **bcmd** must be called prior to **bwrt** to address the device to listen and the MacBus GPIB port to talk. Otherwise, the device on the GPIB that is the CIC must perform the addressing.

If the MacBus GPIB port is Active Controller, the MacBus GPIB port is first placed in Standby Controller state with ATN off and remains there after the write operation is completed. Otherwise, the write operation commences immediately.

An EADR error results if the MacBus GPIB port is CIC but has not been addressed to talk with **bcmd.** An EABO error results if the MacBus GPIB port is not CIC and is not addressed to talk within the time limit. An EABO error also results if the operation does not complete for whatever reason within the time limit.

bwrt terminates on any of the following events:

- When cnt bytes have been written
- Error is detected
- Time limit is exceeded
- Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is CIC

After termination, **ibcnt** contains the number bytes written. A short count can occur on any of the above events but the first.

EXAMPLE

1. To write 10 instruction bytes to a device at listen address 0x35 (ASCII 5) and then unaddress it (the talk address of the MacBus GPIB port is 0x40 or ASCII @):

```
" ?@5" bcmd          ( UNL MTA LAD )
" F3R1X5P2G0" bwrt   ( send instruction bytes )
" _?" bcmd
```

SEE ALSO

bcmd.

Configuration Control Words

IBCL provides words to inspect and alter the MacBus GPIB port and device configuration tables. Using these words, you can change a devices primary or secondary address, the timeouts associated with various devices, and so on.

To inspect or change device configuration table entries, use the IBCL words getd and setd, respectively. These words are used in the following manner:

```
dev field-word getd          ( leaves value on top of stack)
new dev field_word setd      ( replaces old value with new)
```

In these examples dev is a device number or device name used in an **ibfind** call, and field word is one of the device field words defined below:

Field Word	Field Interpretation
d_eos	end of string field
d_eot	end termination mode
d_pad	device primary address
d_sad	device secondary address
d_tmo	device timeout setting

These meaning of each of these fields in the device configuration table is more thoroughly explained in the next few pages.

NAME

d_eos - end of string mode field word

DESCRIPTION

When used with setd, the new value specified selects the eos character and the data transfer termination method according to the following table.

The assignment made by this function remains in effect until beos is called again or **bonl** is called.

Data Transfer Termination Method		
Method	Value of v *	
	Byte 1	Byte 0
A. Terminate read when eos is detected (ibrd and ibrdf)	REOS 0x04	eos
B. Send END when eos is written 0x08 (ibwrt and ibwrtf)	XEOS 0x08	eos
C. Compare all 8 bits of eos byte rather than low 7 bits (all reads and writes)	BIN 0x10	eos

* Byte 0 is the least significant byte.

byte 3	byte 2	byte 1	byte 0

Methods A and C determine how read operations terminate. If Method A alone is chosen, reads terminate when the low 7 bits of the byte that is read match the low 7 bits of the eos character. If Methods A and C are chosen, a full 8-bit comparison is used.

Methods B and C together determine when write operations send the END message. If Method B alone is chosen, the END message is sent automatically when the low 7 bits of any byte match the low 7 bits of the eos character. The eos character should always be the last byte sent. If Methods B and C are chosen, a full 8-bit comparison is used.

EXAMPLES

1. To send END when the linefeed character is written for operations involving device dvm:

```
80A dvm d_eos setd
ascii 1 buf c!l      ( data bytes to )
ascii 2 buf 1 + c!l( be written )
ascii 3 buf 2 + c!l( EOS character )
OA buf 3 + c! l      ( is last byte )
dvm buf 4 wrt
```

2. To program device devl to terminate a read on detection of the linefeed character ('\n' == 0x0A) that is expected to be received within 512 bytes:

```
40A devl d_eos setd
dev1 buf 512 rd
( The END bit is set if the read terminated )
( on the eos character, with the actual number of )
( bytes received contained in ibcnt. )
```

SEE ALSO

d_eot.

NAME

d_eot - END termination mode field

DESCRIPTION

If the value of this field is non-zero, the **END** message is sent automatically with the last byte of each write operation. If the value is zero, END is not sent. **setd** is used to alter the value from its configuration setting.

The END message is sent when the GPIB EOI signal is asserted during a data transfer. It is used to identify the last byte of a data string without having to use an end-of-string character.

The value of this field changes only when it is explicitly set or when **bonl** is called.

EXAMPLES

1. To send the END message with the last byte of all subsequent writes to device plotter:

```
1 plotter d_eot setd          ( enable sending of EOI )
( It is assumed that wrt contains the data to be written )
( to the GPIB )
plotter buf 3 wrt             ( write 3 bytes )
```

2. **To** stop sending END with the last byte:

```
0 plotter d_eot setd          ( disable sending EOI )
```

SEE ALSO

d_eos.

NAME

d_pad - primary address field

DESCRIPTION

The valid primary addresses are 0 to 30. A device listen address is formed by adding 0x20 to the primary address; the talk address is formed by adding 0x40 to the primary address. This is done automatically by the GPIB firmware on the **GPIB-V50.**

This field retains its value until it is explicitly set or **bonl** is called.

A devices primary address determines the talk and listen addresses for use in all I/O directed to that device. The actual GPIB address of any device is set within that device either with hardware switches or a software program. Refer to the manufacturer's device documentation for instructions.

EXAMPLE

1. To change the primary GPIB listen and talk address of device plotter from the configuration setting to **0x2A** and **0x4A** respectively:

```
A plotter d_pad setd
```

SEE ALSO

d_sad.

NAME

d_sad – change or disable secondary address

DESCRIPTION

The valid secondary addresses are 60 to 7E. Values of 0 to 7F disable secondary addressing.

This field retains its value until it is explicitly changed or **bonl** is called.

When secondary addressing is enabled, this field records the secondary GPIB address of that device to be used in subsequent device level I/O function calls.

EXAMPLES

1. To change the secondary GPIB address of device dvm from its current value to **0x6A**:

```
6A dvm d_sad setd
```

2. To disable secondary addressing for device dvm:

```
0 dvm d_sad setd
```

NAME

d_tmo - change or disable timeout limit

DESCRIPTION

The value of this field specifies the timeout limit as follows:

Mnemonic	Field Value	Minimum Timeout
TNONE	0	disabled
T10us	1	10 usec.
T30us	2	30 usec.
T100us	3	100 usec.
T300us	4	300 usec.
T1ms	5	1 msec.
T3ms	6	3 msec.
T10ms	7	10 msec.
T30ms	8	30 msec.
T100ms	9	100 msec.
T300ms	10	300 msec.
T1s	11	1 sec.
T3s	12	3 sec.
T10s	13	10 sec.
T30s	14	30 sec.
T100s	15	100 sec.
T300s	16	300 sec.
T1000s	17	1000 sec.

Note that if the **field** value is zero, no limit is in effect.

The time limit is an escape mechanism used to exit gracefully from a *hung bus* condition. Since the GPIB is an asynchronous bus, read and write operations can be held up indefinitely.

Timeout values are approximate, though never less than indicated.

EXAMPLES

1. To change the time limit for device level I/O operations involving device tape to approximately 300 us:

   ```
   4 tape d_tmo setd
   ```

2. To perform I/O operations with no timeout in effect:

   ```
   0 tape d_tmo setd
   ```

GPIB Port and Device Configuration Tables

To inspect or change MacBus GPIB port configuration table, use the IBCL words getb and setb, respectively. These words have the following syntax:

```
field-word getb          ( current value on stack )
new field_word setb      ( replace old value with new )
```

The valid MacBus GPIB port level field words are:

Field Word	**Field Interpretation**
b_uflags	contains a set of bits used to track information pertinent to the MacBus GPIB port. You may set and monitor the value of this field using setb and getb, just like the other port level configuration words. The bit definition of the **b_uflags** field is as follows:

Bit #	Meaning When Set
0	Assert EOI with last byte of each write
1	Hold off handshake at the end of each read
2	Terminate read when end of string received
3	Assert EOI with end of string byte
4	Use 8-bit compare on the end of string byte
5	Terminate I/O operation on device clear
6	Terminate I/O on address status change
7	reserved
8	reserved
9	GPIB port individual status (IST) bit
10	MacBus will assert remote enable when it is System Controller
11	MacBus is System Controller

12	Tri-state timing is enabled (faster transfers)
13	Repeat addressing is disabled
14	Automatic serial polling is disabled

b_eos	end of string field
b_eot	end termination mode
b_pad	MacBus GPIB port primary address
b_sad	MacBus GPIB port secondary address
b_tmo	MacBus GPIB port timeout value

All of these fields are very similar to their device counterparts except the primary and secondary addresses. The PAD and SAD fields in the device tables specify the address MacBus uses when it tries to communicate with the devices in question. The PAD and SAD fields in the MacBus GPIB port table specify the addresses that MacBus RESPONDS to as its own addresses.

The MacBus GPIB port configuration table also has a field determining whether or not the board uses DMA for GPIB transfers. Switching between DMA and program controlled I/O is more difficult than simply changing a configuration field, so setb does not work on this field. The IBCL word bdma achieves the transition. For example, to use DMA for GPIB transfers, use:

```
1 bdma
```

To use programmed I/O for GPIB transfers, use:

```
0 bdma
```

Advanced Features

IBCL offers one other powerful GPIB management tool. Using IBCL, you may install handlers that IBCL will automatically execute when a GPIB interrupt occurs or a device requests service. The words supporting this ability are:

onirq (irq --) When used in the form

irq onirq xxxxx

onirq will request that the IBCL word xxxxx be executed when the GPIB interrupt irq occurs. irq is one of the following:

SRQI	- service request
DCAS	- device clear
DTAS	- device trigger
ADSC	- address status change
TACS	- talker status change
LACS	- listener status change
CIC	- controller-in-charge status change
REMC	- remote/local status change
LOKC	- lockout status change

If you use the digit 0 for xxxxx, onirq recognizes this as a special value and turns OFF handling of that interrupt. For instance, to disable automatic serial polling, use the command string

SRQI onirq 0

This disables automatic serial polling.

onrqs (d m v --) When used in the form:

d m v **onrqs xxxxx**

onrqs will request that the IBCL word xxxxx be called when GPIB device d requests service with a status byte (stb) which satisfies the test (m & spr)==v. In order for onrqs to operate, the default onirq for SRQ must not be changed.

Additional GPIB Words

IBCL contains several other words for GPIB management that were implemented specifically to ease development of the Megamax C MacBus Support Library. You will probably never have cause to use these words, but this section documents them anyway, just in case.

Configuration Control

The following words change a MacBus GPIB port or device configuration table field and return the old value in the variable ibret. Words beginning with b are MacBus GPIB port words and have the syntax:

```
new-val bxxxxxx
```

Words starting with i are device words and have the syntax:

```
device new_val ixxxxx
```

Field Changing Word	Field Affected
bidma	When **new_val** is zero, MacBus uses programmed control I/O for GPIB transfers; when non-zero, MacBus uses DMA
bieos	MacBus GPIB port end of string field
ieos	device end of string field
bieot	MacBus GPIB port timeout field
ieot	device timeout field
bipad	MacBus GPIB port primary address
ipad	device primary address

bisad	MacBus GPIB port secondary address
isad	device secondary address

GPIB-Macintosh Direct Transfer Words

The MacBus Support Library provides functions which transfer GPIB data and commands directly to/from Macintosh memory. These are synchronous functions; however, control doesn't return until the transfer is 100% complete. In some applications you might be able to exploit parallel processing if you could continue to run your Macintosh program while MacBus handled the GPIB. Using the following IBCL words, you can construct your own asynchronous GPIB functions.

bprd (n --)	**bprd** instructs IBCL to read n bytes from the GPIB using **brd** and then to upload the memory to the host by executing **ulm.**
prd (d n --)	**prd** instructs IBCL to read n bytes from GPIB device d using **rd** and then to upload the memory to the host by executing **ulm.**
bpwt (n --)	**bpwt instructs IBCL to obtain n bytes** of data from the host by executing **dlm** and then to write the data to the GPIB using bwrt.
pwt (d n --)	**pwt** instructs IBCL to obtain n bytes of data from the host by executing dlm and then to write the data to GPIB device d using wrt.

Data moving between the GPIB and the Macintosh must "stop off" in MacBus memory temporarily; the possibility of memory exhaustion exists. The IBCL word **avail?** leaves in ibret the number of bytes available as a temporary transfer buffer. You should never need to use this word; it is built into bprd, prd,

bpwt, and **pwt.**

IBCL **Heap Management Words**

IBCL provides a sophisticated heap manager to manage memory resources on MacBus. The heap manager interface is provided by the following words:

<alloc> (n -- s)	Allocate an n-byte buffer and return its segment address.
alloc (n --)	When used in the form: n **alloc** buf **alloc** allocates an n byte buffer and creates an IBCL word buf that will leave the buffer's segment address on the stack when executed.
free (s --)	Free the buffer starting at segment address s.

The heap manager automatically reclaims free space and merges adjacent free blocks into larger, single free blocks.

The heap is initialized by the ibcl word **init-heap**. Once you have initialized the heap, you MUST NOT use any segment management words, nor even create a new IBCL vocabulary. All segment management should be part of the application compilation process, and the heap should be initialized after the application begins execution. init-heap has no stack arguments.

The skeleton of a sample MacBus session might look like this:

```
vocabulary application
( define application words and data here)
: final-word init-heap word1 word2 etc... ;
final-word( run application)
```

Note that the application program should not use segment management words as part of its algorithm.

Memory Dumping Words

Memory dumps can aid the program debugging process considerably. IBCL provides two dump words, one for near addresses and one for segment addresses:

dump **(a n --)**	Dumps n bytes starting at offset a in the default data segment.
ldump (s a **n** --)	Dumps n bytes starting at long address **s:a.**

SECTION FOUR - TECHNICAL INFORMATION

This section provides technical information about MacBus. This information is provided to allow knowledgeable users to interface compatible interface cards to their systems. This section describes the physical, environmental, and electrical specifications for MacBus as well as a technical description of the major system components.

PHYSICAL SPECIFICATIONS

MacBus measures 6 inches high, by 11 3/4 inches wide, by 15 1/4 inches deep and weighs approximately 15 pounds. The unit can accommodate up to three optional IBM PC or IBM PC AT compatible interface cards in addition to the two standard cards which come with the unit.

ENVIRONMENTAL SPECIFICATIONS

The GPIB-V50 is designed for use under the following environmental conditions:

Storage Temperature:	0 to 70 degrees C
Operating Temperature:	10 to 40 degrees C
Humidity:	10% to 95% non-condensing

ELECTRICAL SPECIFICATIONS

The electrical specifications for the MacBus power supply are as follows:

Input Voltages:	90V to 130V (115V nominal) or 180V to 260V (230V nominal)
Input Frequency:	47-440 HZ
Inrush Current:	20A maximum peak at cold start
Fuse on Input Line:	2.5A, 250V
+5VDC Current:	13A max *
+12VDC Current:	5A max *
-12VDC Current:	1A max *
Max Output Wattage:	100W

* Note: Maximum current cannot be drawn from all outputs simultaneously. At no time should the average power (excluding transients) exceed the maximum output wattage.

The electrical specifications for the GPIB-V50 and SCSI-PC boards are as follows:

GPIB-V50:	+5VDC	1.45A typical
SCSI-PC:	+5VDC	380mA typical

DETAILED DESCRIPTION

MacBus consists of three basic components: an enclosure, a SCSI-PC interface, and a GPIB-V50 microprocessor card. The MacBus operating system is contained in ROM on the GPIB-V50 card and is called the Interface Bus Interactive Control program (IBCL). IBCL handles all system interaction including SCSI I/O and all IEEE-488 functions.

Enclosure

The enclosure is an "AT style" box complete with a switching power supply, room for up to three disk drives, and a motherboard with connectors for either IBM PC or IBM PC AT style cards. The switching power supply supplies +12VDC, -12VDC, and +5VDC to the I/O connectors. Because of the nature of switching power supplies, it is necessary to load the outputs so that a minimum amount of current will flow from the supply. The MacBus interface cards, as shipped from the factory, do not draw enough current to properly load down the +5V or +12V supply outputs. For dependable operation these lines have been loaded with power resistors which are connected to the inside of the chassis case.

If optional interface cards are added, it is possible that the cards will load down the power supply outputs so that the power resistors are no longer needed. If this is the case, remove the resistors from the circuit by disconnecting the connector between the resistors and the supply.

Caution: The power resistors get extremely hot after only a few minutes of operation. Never touch the resistors after operation and avoid touching the area around where the resistors are connected.

Use the following guidelines to help you determine when the resistors should be disconnected.

1. If, when adding optional interface cards, the total current draw (typical) from all the boards in the system exceeds 6.0A

on the +5V line, then disconnect the power resistor with the red and black wires.

2. If, when adding optional interface cards, the total current draw (typical) from all the boards in the system exceeds 2A on the +12V line, then disconnect the power resistor with the red and blue wires.

The typical current draw for the GPIB-V50 and SCSI-PC cards is given on page 4-2.

Also, remember if you remove a board or boards from your system and the total current draw falls under the thresholds given above, then you need to reconnect the resistors.

SCSI-PC Interface

The factory configurations for the SCSI-PC are shown below:

	Factory Setting
SCSI-PC I/O Addresses:	330-33F hex
SCSI ID:	6
Interrupt level:	5
DMA channel:	3
Terminating Resistors:	present

The SCSI-PC I/O address, interrupt level, and DMA channel should not be changed. If any of these parameters conflict with an optional adapter card, the optional adapter card must be changed so that the conflict is removed. The SCSI ID and the presence of the terminating resistors can and should be changed under certain circumstances. To change the SCSI PC configuration, open the unit as described in SECTION TWO, **Install Internal Options paragraph, and** remove the SCSI-PC card. Figure 4-l shows a parts locator diagram for the SCSI-PC card. Use this figure to locate the SCSI ID switch and the terminating resistors. Refer to the following paragraphs for descriptions of these options.

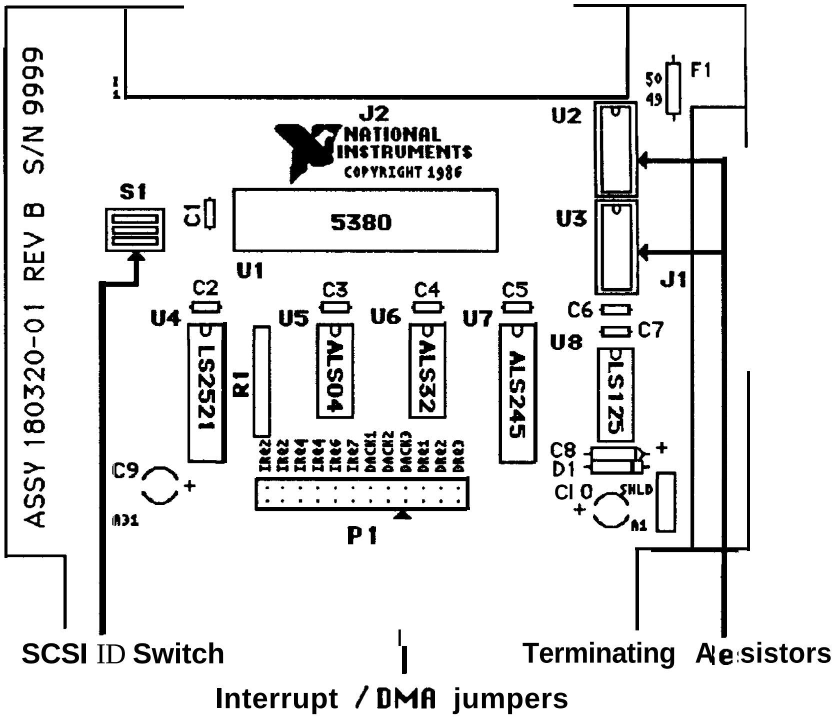


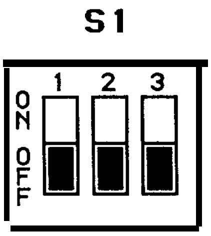
Figure 4-l - SCSI-PC Parts Locator Diagram

SCSI-PC I/O Addresses

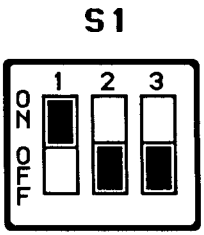
The SCSI-PC interface uses 16 I/O registers from 330 hex to 33F hex. These I/O registers are hardwired so their locations cannot be changed. When adding I/O interface adapters, make sure their I/O addresses do not conflict with those of the SCSI adapter or any other adapter which has been added to the system.

SCSI ID Switch Settings

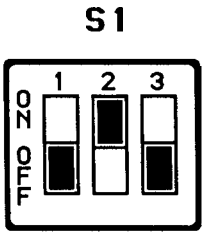
Each device on the SCSI bus must have its own unique device ID. The Macintosh Plus is set to use SCSI ID 7; the MacBus is shipped from the factory with its SCSI ID set to 6. If the MacBus SCSI ID conflicts with another SCSI device, one of the devices SCSI IDs must be changed to remove the conflict. To change the SCSI ID used by the SCSI-PC, set the switches as shown below for the desired SCSI ID.



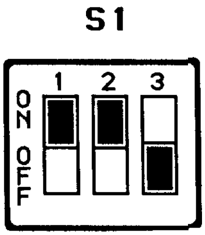
SCSI ID = 7
Do not use.
Used by Macintosh Plus



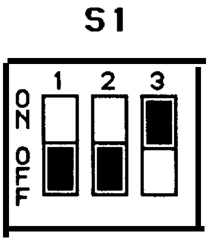
SCSI ID = 6
(Factory Default)



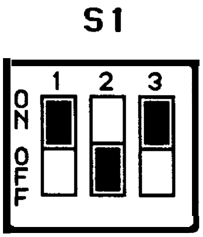
SCSI ID = 5



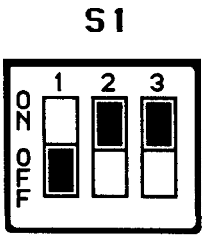
SCSI ID = 4



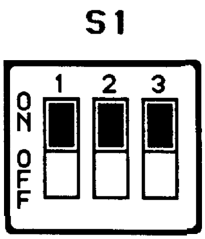
SCSI ID = 3



SCSI ID = 2



SCSI ID = 1



SCSI ID = 0

Figure 4-2 - SCSI ID Switch Settings

If the SCSI ID is changed from its factory default setting (6), it will be necessary to run SCSI_conf to notify the software of the SCSI ID setting change.

DMA Settings

The SCSI-PC interface uses DMA channel 3 for high-speed transfers between the SCSI bus and GPIB-V50 memory. Optional interface boards can use any DMA channel other than channel 3 (or 7 since it is connected to channel 3). The jumpers on the SCSI-PC must be jumpered as shown in Figure 4-3 before running any software applications from the Macintosh Plus.

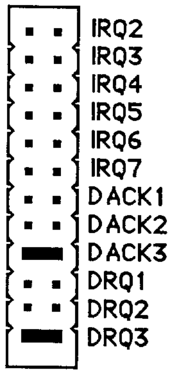


Figure 4-3 - SCSI-PC DMA Channel 3 Setting

Interrupt Settings

The SCSI-PC uses interrupt level 5. Optional interface cards can use any interrupt level other than level 5. The interrupt jumper on the SCSI-PC must be jumpered as shown in Figure 4-4 before running any software applications from the Macintosh Plus.

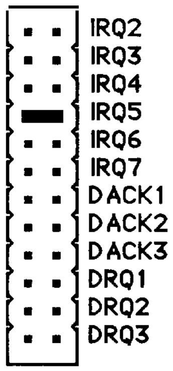


Figure 4-4 - SCSI-PC Interrupt Level 5 Setting

Terminating Resistors

The SCSI standard dictates the use of termination resistors on all SCSI signal lines for the first and last device on the SCSI bus. The first and last devices are defined as the two devices which are connected to only one other SCSI device. If, for example, MacBus and the Macintosh Plus are the only two SCSI devices, then they both will use terminating resistors. The SCSI-PC has been designed to have the ability of terminating the SCSI lines on the card. If MacBus is the first or last device on the SCSI bus, make sure that the terminating resistor modules are plugged into their sockets (U2 and U3 - Figure 4-l). If MacBus is not the first or last SCSI device, carefully remove these modules and store them in a safe place for future use.

The GPIB-VSO

The system's RAM is located on the GPIB-VSO card and starts at hex 00000 of the 1 Mb address space. The RAM is socketed in the locations designated U1 through U8 and U16 through U23 on the GPIB-VSO card.

The RAM subsystem consists of either 16 65,536x1 or 16 262,144x1 dynamic RAM modules for a total address space of either 128K or 512K, respectively. If the GPIB-VSO card is socketed with 128K bytes of RAM, you can increase your RAM space by changing the RAM modules; however, the following precautions should be made:

- The RAM modules, as well as other system components, can be damaged by improper handling techniques. Make sure you use a grounded wrist strap and work on a grounded work area when changing the RAM modules.
- Memory chips should be change only with the same or compatible type of RAM modules with a maximum access time of 120 nanoseconds.
- All 16 RAM locations must be socketed with the same type of RAM modules — either 64Kx1 or 256Kx1.

There are no user configurable jumpers or switches on the GPIB-V50 card. The system dynamically determines the amount of RAM on power-up and all other available options are configurable in software through IBCL function calls.

Although the GPIB-VSO was designed to allow interfacing to IBM PC and IBM PC AT compatible interfaces, there are a few notable differences between the signals supported on the GPIB-VSO I/O connector and those provided on an IBM PC AT. These differences are listed below:

- LA17 through LA23 are not supported on the GPIB-VSO
- CLK is an 8 MHz signal
- OWS is not supported on the GPIB-VSO

. DRQO and -DACKO are connected to the on-board GPIB circuitry but can be disconnected in software if they are needed by another board

• 16-bit bus operations to 8-bit devices are not supported on the GPIB-V50

. The seven IBM PC AT DMA channels are mapped into the four GPIB-V50 channels as shown below (DMA Channel 0 is the highest priority). The DMA acknowledge signals are mapped accordingly.

DMA Channel	62-pin Connector	36-pin Connector	Transfer Type Supported
0	not bussed	DRQO(D09)	8-bit
1	DRQ1	DRQ5(D11)	8- or 16-bit (programmable)
2	DRQ2	DRQ6(D13)	8- or 16-bit (programmable)
3	DRQ3	DRQ7(D15)	used by the SCSI-PC card

Notice that even though the I/O bus provides seven separate channels for DMA, only a maximum of four can be used at a time (including the DMA channel used for GPIB operation).

. The 11 IBM PC AT interrupt levels are mapped into the six available V50 interrupt levels as shown below (V50 interrupt level 2 has the highest priority).

V50 Level	62-pin Connector	36-pin Connector	Function
2	IRQ2(B04)	IRQ10(D03)	**Available** to I/O card
3	IRQ3(B25)	IRQ1 1(D04)	Available to I/O card
4	IRQ4(B24)	IRQ12(D05)	Available to I/O card
5	IRQ5(B23)	-	Used by the SCSI-PC card
6	IRQ6(B22)	IRQ15(D06)	Available to I/O card
7	IRQ7(B21)	IRQ14(D07)	Available to I/O card

Notice that even though there are 11 interrupt channels on the I/O bus, only a maximum of six may be used at a time.

SECTION FIVE - DIAGNOSTICS & TROUBLESHOOTING

This section is designed to help you determine the cause of any problem you may be having with your MacBus setup. A checklist is provided to help you troubleshoot your system. If for some reason you cannot solve the problem with the procedures outlined below, call National Instruments for further instructions.

WHERE TO BEGIN

The following checklist should be read and followed before you try more involved troubleshooting procedures:

Is the AC power cord attached properly and plugged into an outlet of the correct voltage (the voltage is noted on the label to the right of the fan outlet)?

Is the power switch to MacBus and all other SCSI devices in the ON position?

Is the SCSI cable attached and seated correctly on MacBus, the Macintosh Plus, and any optional SCSI devices?

Do all SCSI devices have their own unique SCSI ID?

Did you power all SCSI devices on well in advance (10 seconds) of powering on your Macintosh Plus?

Did you make sure any optional plug in cards are compatible with MacBus and have their own unique address space, interrupt lines, and DMA channels?

If the above checklist has been read and followed and you are still experiencing a problem, it will be necessary to narrow the problem down to a specific type. The problems most likely to occur are problems caused by incorrect programming techniques or optional system components such as plug-in cards or SCSI devices. By following the procedures below you should be able to determine the cause of the problem and correct it yourself.

CHECK ELECTRICAL CONNECTIONS

1. Turn the system unit switch to the OFF position.
2. Unplug the system unit's power cord from the wall outlet.
3. Check the wall outlet by plugging in a working lamp.
4. Disconnect and reconnect each cable on MacBus, including the power cord, to ensure proper electrical connection.
5. Plug the MacBus power cord back into the wall outlet and turn the system unit switch to the ON position. Listen for the sound of the fan in the unit. If the fan does not operate when the unit is powered on, the MacBus fuse may be blown. The fuse may be replaced by opening the unit and removing the cover of the power supply. Remove power to MacBus before opening the unit as dangerous voltages are present inside the power supply unit. Replace only with the same type and rating of fuse.

ISOLATE THE SYSTEM

If the system was configured to have other SCSI devices or plug-in cards, there is a chance that these options could be causing a problem because 1) they themselves are not working properly, 2) they are not compatible with the MacBus system interface, or 3) the devices are set up improperly or have conflicting interface parameters. In any event, you will be able to determine if these devices are causing your problem by removing them from your system as described below.

1. Turn the power switch of the Macintosh Plus, MacBus, and all SCSI peripherals to OFF.
2. Remove all external SCSI peripherals by disconnecting their cables from the SCSI daisy chain.
3. Remove all plug-in cards, except the GPIB-V50 card and the SCSI-PC card, from the MacBus backplane.

4. Make sure the SCSI card is set to the factory default settings. These settings are given on page 4-4. Also, make sure the SCSI terminating resistors are installed in the SCSI-PC board.
5. Connect the SCSI cable to the Macintosh Plus and to MacBus.

RUN DIAGNOSTIC SOFTWARE

The diagnostic software sends a command to MacBus which tells it to return the results from the power-on self-test. The results from the test will be posted on the Macintosh Plus screen to aid you in determining the problem. If MacBus is unable to send any information back to the Macintosh Plus, the diagnostic software will post a SCSI bus error.

1. Check all electrical connections then power up MacBus. Wait 10 seconds and then power on your Macintosh Plus.
2. Insert the distribution disk, which came with your MacBus, in the Macintosh Plus disk drive and double click on the IBDIAG icon when it appears. This will start the on-board diagnostic routines on MacBus.
3. The results from the diagnostics are displayed on the Macintosh screen. The following is a list of possible error messages.

Error Code	Description
0	No errors
1	ROM checksum failure
2	CPU internal register failure
3	CPU on-board peripheral failure
4	RAM failure
5	GPIB failure
6	SCSI bus error

4. If the diagnostics pass, then one or more of the options you were using in your system may be causing MacBus to malfunction. Try reinstalling the options one-by-one until you determine which option is causing the system to malfunction. If the diagnostics fail then there is a possibility that MacBus is not functioning properly. Check that the boards are plugged in correctly and that all electrical connections are made. If the diagnostics still do not pass, call National Instruments to get a Return Material Authorization (RMA) number before returning the product for service.

APPENDIX A - MULTILINE INTERFACE MESSAGES

Multiline interface messages are sent and received with ATN TRUE.

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL		20	040	32	SP	MLAO
01	001	1	SOH	GTL	21	041	33	!	MLA1
02	002	2	STX		22	042	34	"	MLA2
03	003	3	ETX		23	043	35	#	MLA3
04	004	4	EOT	SDC	24	044	36	$	MLA4
05	005	5	ENQ	SDC	25	045	37	%	MLA5
06	006	6	ACK		26	046	38	&	MLA6
07	007	7	BEL		27	047	39	'	MLA7
08	008	8	BS	GET	28	050	40	(	MLA8
09	009	9	HT	TCT	29	051	41	)	MLA9
OA	012	10	LF		2A	052	42	*	MLA10
OB	013	11	VT		2B	053	43	+	MLA11
o c	014	12	FF		2c	054	44	,	MLA12
OD	015	13	CR		2D	055	45		MLA13
OE	016	14	s o		2E	056	46	.	MLA14
OF	017	15	SI		2F	057	47	/	MLA15
10	020	16	DLE		30	060	48	0	MLA16
11	021	17	DC1	LLO	31	061	49	1	MLA17
12	022	18	DC2		32	062	50	2	MLA18
13	023	19	DC3		33	063	51	3	MLA19
14	024	20	DC4	DCL	34	064	52	4	MLA20
15	025	21	NAK	PPU	35	065	53	5	MLA21
16	026	22	SYN		36	066	54	6	MLA22
17	027	23	ETB		37	067	55	7	MLA23
18	030	24	CAN	SPE	38	070	56	8	MLA24
19	031	25	EM	SPD	39	071	57	9	MLA25
1A	032	26	SUB		3A	072	58		MLA26
1B	033	27	ESC		3B	073	59	,	MLA27
1C	034	28	FS		3c	074	60	<	MLA28
1D	035	29	GS		3D	075	61	=	MLA29
1E	036	30	RS		3E	076	62	>	MLA30
1F	037	31	u s		3F	077	63	?	UNL

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
40	100	64	@	MTA0	60	140	96	‘	MSA0,PPE
41	101	65	A	MTA1	61	141	97	a	MSA1,PPE
42	102	66	B	MTA2	62	142	98	b	MSA2,PPE
43	103	67	C	MTA3	63	143	99	c	MSA3,PPE
44	104	68	D	MTA4	64	144	100	d	MSA4,PPE
45	105	69	E	MTA5	65	145	101	e	MSA5,PPE
46	106	70	F	MTA6	66	146	102	f	MSA6,PPE
47	107	71	G	MTA7	67	147	103	g	MSA7,PPE
48	110	72	H	MTA8	68	150	104	h	MSA8,PPE
49	111	73	I	MTA9	69	151	105	i	MSA9,PPE
4A	112	74	J	MTA10	6A	152	106	j	MSA10,PPE
4B	113	75	K	MTA11	6B	153	107	k	MSA11,PPE
4c	114	76	L	MTA12	6C	154	108	l	MSA12,PPE
4D	115	77	M	MTA13	6D	155	109	m	MSA13,PPE
4E	116	78	N	MTA14	6E	156	110	n	MSA14,PPE
4F	117	79	0	MTA15	6F	157	111	0	MSA15,PPE
50	120	80	P	MTA16	70	160	112	p	MSA16,PPD
51	121	81	Q	MTA17	71	161	113	q	MSA17,PPD
52	122	82	R	MTA18	72	162	114	r	MSA18,PPD
53	123	83	S	MTA19	73	163	115	s	MSA19,PPD
54	124	84	T	MTA20	74	164	116	t	MSA20,PPD
55	125	85	U	MTA21	75	165	117	u	MSA21,PPD
56	126	86	V	MTA22	76	166	118	v	MSA22,PPD
57	127	87	W	MTA23	77	167	119	w	MSA23,PPD
58	130	88	X	MTA24	78	170	120	x	MSA24,PPD
59	131	89	Y	MTA25	79	171	121	y	MSA25,PPD
5A	132	90	Z	MTA26	7A	172	122	z	MSA26,PPD
5B	133	91	[	MTA27	7B	173	123	(	MSA27,PPD
5c	134	92	\	MTA28	7c	174	124	|	MSA28,PPD
5D	135	93	]	MTA29	7D	175	125	}	MSA29,PPD
5E	136	94	^	MTA30	7E	176	126	~	MSA30,PPD
5F	137	95	_	UNT	7F	177	127	DEL	

APPENDIX B - THE GPIB

The IEEE-488, also known as the General Purpose Interface Bus or GPIB, is a high speed parallel bus structure originally designed by Hewlett-Packard. It is generally used to connect and control programmable instruments, but has gained popularity in other applications, such as intercomputer communication and peripheral control.

TYPES OF MESSAGES

The GPIB carries device-dependent messages and interface messages.

- Device-dependent messages, often called data or data messages, contain device-specific information such as programming instructions, measurement results, machine status, and data files.
- Interface messages manage the bus itself. They are usually called commands or command messages. Interface messages perform such functions as initializing the bus, addressing and unaddressing devices, and setting device modes for remote or local programming.

The term command as used here should not be confused with some device instructions which can also be called commands. Such device-specific instructions are actually data messages.

TALKERS, LISTENERS, AND CONTROLLERS

A Talker sends data messages to one or more Listeners. The Controller manages the flow of information on the GPIB by sending commands to all devices.

Devices can be Listeners, Talkers, and/or Controllers. A digital voltmeter, for example, is a Talker and may be a Listener as well.

The GPIB is a bus like an ordinary computer bus except that the computer has its circuit cards interconnected via a backplane bus whereas the GPIB has standalone devices interconnected via a cable bus.

The role of the GPIB Controller can also be compared to the role of the computer's CPU, but a better analogy is to the switching center of a city telephone system.

The switching center (Controller) monitors the communications network (GPIB). When the center (Controller) notices that a party (device) wants to make a call (send a data message), it connects the caller (Talker) to the receiver (Listener).

The Controller usually addresses a Talker and a Listener before the Talker can send its message to the Listener. After the message is transmitted, the Controller usually unaddresses both devices.

Some bus configurations do not require a Controller. For example, one device may always be a Talker (called a Talk-only device) and there may be one or more Listen-only devices.

A Controller is necessary when the active or addressed Talker or Listener must be changed. The Controller function is usually handled by a computer.

The MacBus system permits your Macintosh Plus to play all three roles.

- Controller - to manage the GPIB
- Talker - to send data
- Listener - to receive data

THE CIC AND SYSTEM CONTROLLER

Although there can be multiple Controllers on the GPIB, only one Controller at a time is active or Controller-In-Charge (CIC). Active control can be passed from the current CIC to an idle Controller. Only one device on the bus, the System Controller, can make itself the CIC. MacBus is usually the System Controller.

GPIB SIGNALS AND LINES

The interface system consists of 16 signal lines and 8 ground return or shield drain lines.

The 16 signal lines are divided into the following three groups.

- 8 data lines
- 3 handshake lines
- 5 interface management lines

Data Lines

The eight data lines, DI01 through D108, carry both data and command messages. All commands and most data use the 7-bit ASCII or IS0 code set, in which case the 8th bit, D108, is unused or used for parity.

Handshake Lines

Three lines asynchronously control the transfer of message bytes among devices. The process is called a three-wire interlocked handshake and it guarantees that message bytes on the data lines are sent and received without transmission error.

NRFD (not ready for data)

NRFD indicates when a device is ready or not ready to receive a message byte. The line is driven by all devices when receiving commands and by Listeners when receiving data messages.

NDAC (not data accepted)

NDAC indicates when a device has or has not accepted a message byte. The line is driven by all devices when receiving commands and by Listeners when receiving data messages.

DAV (data valid)

DAV tells when the signals on the data lines are stable (valid) and can be accepted safely by devices. The Controller drives DAV when sending commands and the Talker drives it when sending data messages.

Interface Management Lines

Five lines are used to manage the flow of information across the interface.

ATN (attention)

The Controller drives ATN true when it uses the data lines to send commands and false when it allows a Talker to send data messages.

IFC (interface clear)

The System Controller drives the IFC line to initialize the bus and become CIC.

REN (remote enable)

The System Controller drives the REN line, which is used to place devices in remote or local program mode.

SRQ (service request)

Any device can drive the SRQ line to asynchronously request service from the Controller with the SRQ line.

EOI (end or identify)

The EOI line has two purposes. The Talker uses it to mark the end of a message string. The Controller uses it to tell devices to identify their response in a parallel poll.

PHYSICAL AND ELECTRICAL CHARACTERISTICS

Devices are usually connected with a cable assembly consisting of a shielded 24 conductor cable with both a plug and receptacle connector at each end. This design allows devices to be linked in either a linear or a star configuration, or a combination of the two. See Figures B-l, B-2, and B-3.

The standard connector is the Amphenol or Cinch Series 57 MICRORIBBON or AMP CHAMP type. An adapter cable using non-standard cable and/or connector is used for special interconnect applications.

The GPIB uses negative logic with standard TTL logic level. When DAV is true, for example, it is a TTL low level ($\leq 0.8V$), and when DAV is false, it is a TTL high level ($\geq 2.0V$).

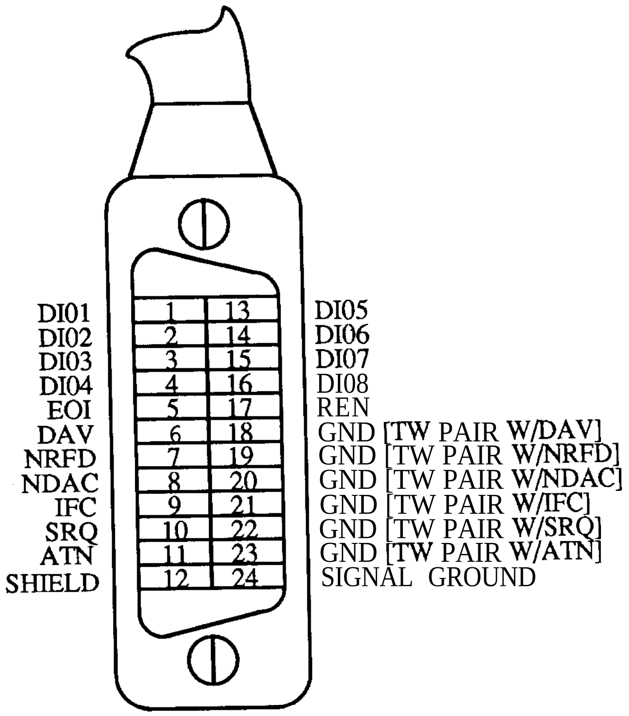


Figure B-l - GPIB Connector and the Signal Assignment

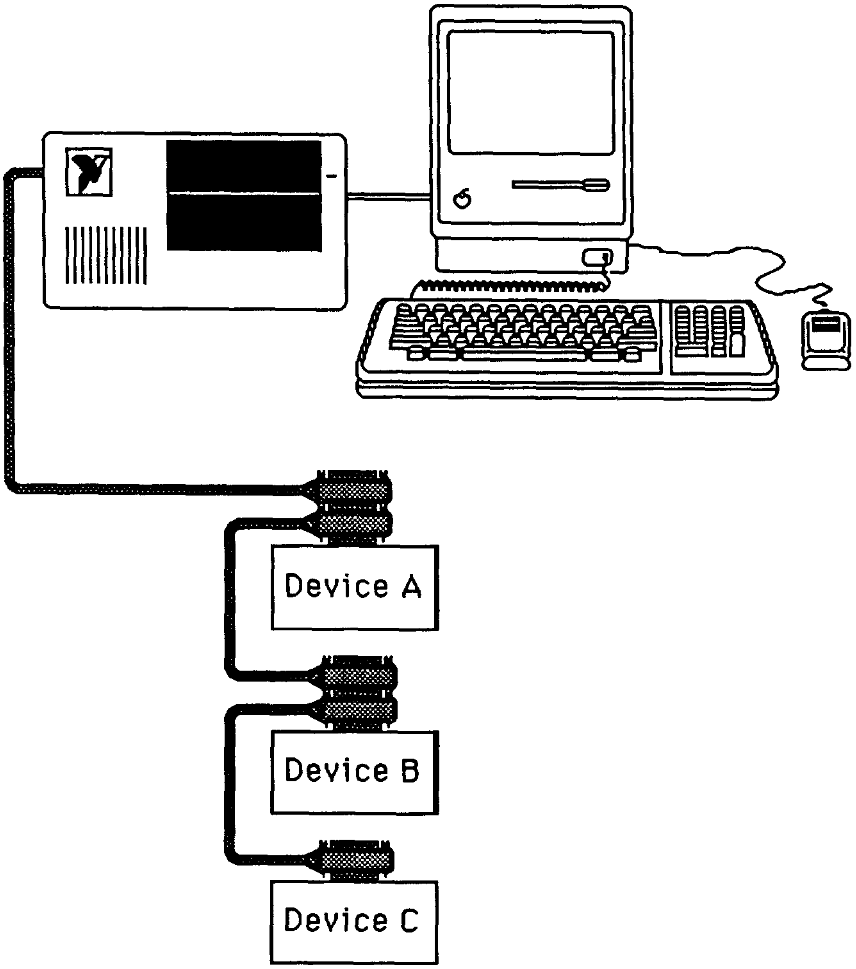


Figure B-2 - Linear Configuration

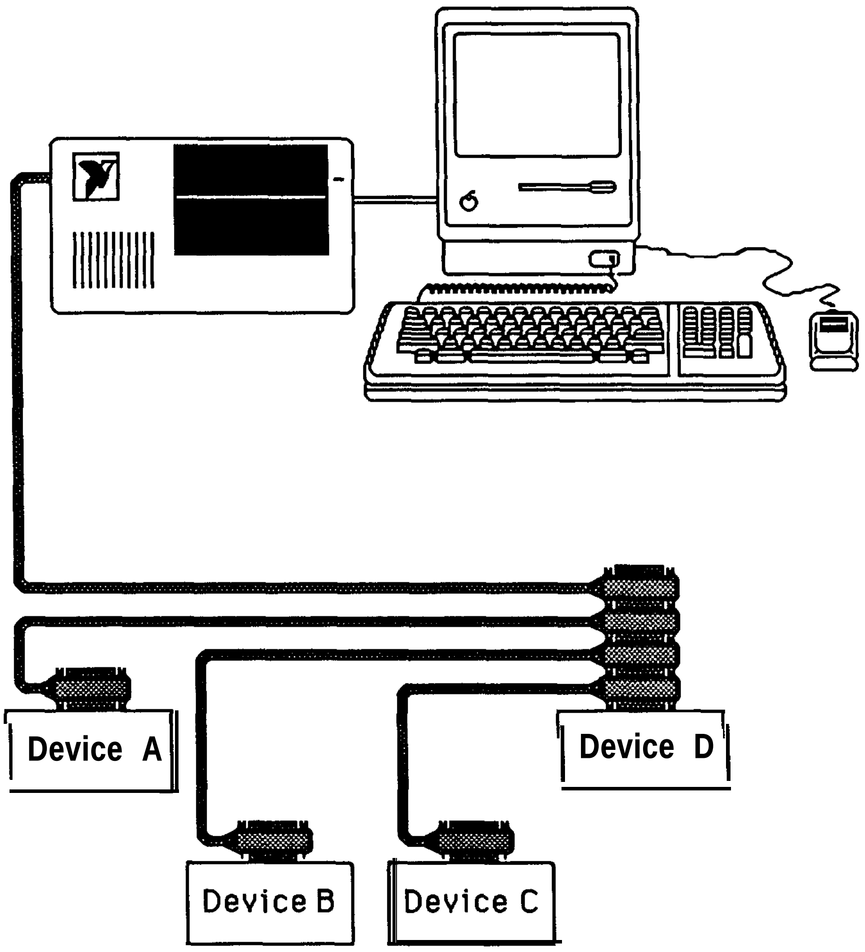


Figure B-3 - Star Configuration

CONFIGURATION REQUIREMENTS

To achieve the high data transfer rate that the GPIB was designed for, the physical distance between devices and the number of devices on the bus are limited.

The following restrictions are typical.

- A maximum separation of four meters between any two devices and an average separation of two meters over the entire bus.
- A maximum total cable length of 20 meters.
- No more than 15 devices connected to each bus, with at least two-thirds powered on.

Bus extenders are available from National Instruments and other manufacturers for use when these limits must be exceeded.

RELATED DOCUMENTS

For more information on topics covered in this section consult the following related documents.

- IEEE Std. 488-1978, “IEEE Standard Digital Interface for Programmable Instrumentation.”
- MacBus Technical Reference Manual.

APPENDIX C - IBCL COMMAND SUMMARY

key:	n	16 bit integer
	u	16 bit unsigned integer
	i	16 bit signed integer
	d	32 bit integer
	du	32 bit unsigned integer
	di	32 bit signed integer
	s	16 bit segment paragraph number
	a	16 bit ds relative address
	e	16 bit es relative address
	uv	16 bit ds relative address of user variable
	sd	16 bit ds relative address of segment descriptor
	fcb	16 bit ds relative address of file control block
	#	16 bit character count
	c	use only lower byte of 16 bit number
	xxx	a name to be entered into or found in a vocabulary, or a filename
	i	immediate, executes from compile mode
	c	use this word only while compiling

Note: top of stack is to the right.

word	stack before - after	description
!	n a -	store word n at ds:a
!%	UC - u+l	store byte c at cs:u and return u+1
! e	n e -	store word n at es:e
! l	n s a -	store word n at s:a
#	dul - du2	extract next digit of dul and send to output string
#>	du - a c	end pictured numeric conversion, return address and count for type
#s	du - 0 0	extract remaining digits from du and send to

		output string
$!	al a2 -	copy string from al to a2
$!l	sl al s2 a2 -	copy string from s1:a1 to s2:a2
$=	sl al s2 a2 - f	compare strings
$+	al a2 - pad	concatenate strings
$+l	sl al s2 a2 - pad	concatenate strings
$constant xxx string”		initialize string constant
Semit	n c -	write block of length n of c
$find	a0 sl al a2 - sl a3 f	find string a0 in s1:a1...s1:a2
$line		read line from keyboard to pad
$type	a -	block write of ($) string at a
$variable xxx	n -	allot n byte string variable
$var-far xxx	sd n -	allot n byte string variable in segment sd
$xchg	al a2 -	exchange string at sl:al with s2:a2
%>		terminate machine code byte list
´ xxx	- a ic	return parameter field address of next word, error if not found

The following are user variables which contain the code field address currently assigned to the word left after dropping the leading tick.

‘-find	'cr	‘vocabulary
‘word		
´abort	‘expect	'number
‘interpret		

"	-- s 0 n	take text from the input stream up to the closing " and place n bytes at segment s, offset 0
(comment)	-	comment, ignore characters until)
*	il i2 - i3	signed product of il and i2
*/	il i2 i3 - i4	signed 32 bit product of il and i2 divided by i3 8 rounded toward 0
*/mod	il i2 i3 - i4 i5	like */ i4=remainder i5=quotient
+	nl n2 - n3	signed or unsigned sum of n1& n2
+!	n a -	add n onto word value at ds:a
+-	il i2 - i3	return il, negated if i2 is negative
+>r	n -	add n to top number on return stack
+buf	al - a2 f	return address of next disk buffer f is false if a2 is prev buffer
+code xxx		name entry for overlaid code
+is name	n l n 2 -	store nl at pfa+2+(2*n2)
+loop	i -	do loop terminator using signed index with step i

,	n -	append n to definition lists allot 2 bytes
, words	n -	append n to current vocabulary allot 2 bytes
	nl n2 - n3	nl minus n2 signed or unsigned
-1	- -1	constant
-2	- -2	constant
-find xxx	- pfa lb t - f	return pfa, length byte and true if next word in input stream is found, otherwise return false
-text	al u a2 - f	compare strings at al & a2 over length u, return false if equal
-trailing	a #1 - a #2	reduce character count to omit trailing blanks
	i -	type signed integer i with one trailing blank using base
." msg"	ic	type following message until "
.code' xxx		dump machine code for word xxx
.head		dump code for word currently being defined
.index	s# -	type s# and line 0 of screen s#
.line	l# s# -	type line l# of screen s#
.memory		display segment names and ranges
.r	i u -	type i right aligned in a field of u characters
.s	stack - stack	list data stack in ascending or descending order
.seg	sd - prevsd	type name and range of segment sd then link to previous segment
.sl		set .ss to f for ascending order
.sr		set .ss to t for descending order
.ss	- f	constant, data stack list order
. word	cfa -	type word with this cfa from context vocabulary
/	il i2 - i3	quotient of il divided by i2 rounded toward zero
/loop	u - ic	do loop terminator using unsigned index and step u
/mod	il i2 - i3 i4	i1/i2 remainder=i3 quotient=i4
0	- 0	constant
0!	a -	store 0 at word address ds:a
O!e	e -	store 0 at word address es:e
0<	i - f	true if i negative
o=	n - f	true if n is 0
0>	i - f	true if i greater than 0

0branch	f -		run time procedure interpreter branches if flag false
Oc!e	e -		store 0 at byte address es:e
1	- 1		constant
1+	n - n+l		increment top word on stack
1+!	a -		increment nunber at word addr ds:a
1-	n - n-l		decrement top word on stack
l-!	a -		decrement number at word addr ds:a
2	- 2		constant
2*	i - i*2		multiply top word on stack by two
2+	n - n+2		add two to top word on stack
2+!	a -		increment number at word address ds:a twice
2-	n - n-2		subtract two form top word on stack
2-!	a -		decrement number at word address ds:a twice
2/	i - i/2		divide by two, round towards zero
3	- 3		constant
4	- 4		constant
4+	n - n+4		add four to top word on stack
6+	n - n+6		add six to top word on stack
			begin high-level colon definition
;		ic	end colon definition
;' xxx		ic	create new defining word which will use word xxx for execution time routine
;case		ic	terminate case: vector table
;code		ic	create new defining word which will use address of following code for execution time routine
<	il i2 - f		true if il is less than i2
<#	du - du		initialize pictured numeric output
<+loop>	i -	c	run time procedure for +loop
<-find> xxx	- pfa lb t - f		find pfa and length byte of next word, return false if not found
<-findw> xxx	- lpa lb t - f		as above, but return es relative list pointer address, leave vocabulary segment pointer in es
<.">		c	type following inline text
</loop>	u -	c	run time procedure for /loop
<;code>		c	replace code field of most recently defined word with address of following machine code
<abort">		c	run time procedure for abort"
<abort>			clear data & return stacks, quit
<cmovel	sl a s2 e # -		copy # bytes from block at sl:a to block at s2:b, beginning with highest address
<cmove>	a e # -		copy # bytes from block at ds:a to block at

			ds:e, beginning with lowest address
<cr>			type carriage return, linefeed
<do>		c	run time procedure for do
<docon>	- n		run time code for constants
<dovar>	- pfa		run time code for variables
<doarrf>	- s a		run time code for array-fars
<emit>	c -		type character c and increment out
<expect>	a # -		move characters from terminal to string at ds:a until cr received or count of # reached
<fill>	a u c -		fill u bytes after ds:a with c
<find>	al a2 - lpa lb t - f		search vocabulary for al beginning at a2, return false if not found
<interpret>			interpret input stream beginning at blk, >in
<line>	1# s# - a #		return address & line length of line 1# screen s#
<loop>		c	run time procedure for loop
<number>	a - di		convert string at ds:a to number
<voc79>			primitive for FORTH79 vocabulary
<word>	c - a		transfer characters from input stream to string at ds:a until c is found or stream exhausted
=	nl n2 - f		true if nl equals n2
>	il i2 - f		true if il greater than i2
>in	- uv		offset to present character in input stream
>r	n -	c	transfer n to return stack
	a -		type number at ds:a
?comp			error if not compiling
?csp			error if stack position not csp
?dup	n - n n n - 0		duplicate n if not zero
?(	f -		ignore until next) only if f false
?pai rs	nl n2 -		error if nl not equal to n2
?seg-size	# sd - # sd		error if destination segment is too small
?stack			error if stack out of bounds
?stream	f -		error if input stream exhausted indicated by true flag
@	a - n		place number at ds:a on stack
@e	e - n		place number at es:e on stack
@l	s a - n		place number at s:a on stack
abort			clear data and return stack return control to terminal
abort" msg"	f -	c	type message and abort if flag true
abs	i - u		absolute value of integer i
again	a l -	ic	compile branch to start of begin - again loop
al lot	u -		assign u more bytes to parameter field of

		most recent word
al lot%	# -	allot u bytes in code segment
allot-seg	u - u s	allot u bytes from free memory for new segment
and	ul u2 - u3	ul and u2
array-far xxx	sd # -	define long variable xxx of length # bytes using segment from segment descriptor at ds:sd
ascii c	- c	return ASCII code of following char
	c	or compile ASCII code as literal
avai l	- -	scratch area of memory used for GPIB to Macintosh data transfers. Execution leaves the byte size of the scratch area in the return variable, ibret.
b->w	c - i	sign extend 8 bit value to 16 bits
base	- uv	base for numeric i/o conversion
bcac	b --	become active controller. send local message tcs if b=O else send tca.
bcmd	L c --	L is long address of command string which is c bytes long.
begin	- a I	return address and identifier for begin type loop control words
behead' xxx		delete header
behead" xxx yyy		delete range of headers
bgts	b --	go to standby. send local message gts if b=O or listen in continuous mode if b<>0.
bist	v bist	if v is non-zero, the individual status bit is set. if v=0, the bit is cleared
bl	- 20h	constant: ASCII blank
blank	a u -	fill u bytes after ds:a with blanks
bloc	—	send local message return to local.
bonl	b --	if b=1 power on reset. if b=O place GPIB offline
bppc	v bppc	if v is a valid parallel poll enable/disable command
branch		run time procedure to unconditionally branch
brd	L c --	read c bytes into long address L
brpp	--	conduct parallel poll. If 'ibret' shows no error a valid poll response is stored in 'ibppr'
brsc	v brsc	if v is non-zero, functions requiring System Controller capability are allowed
brsv	c --	request service with byte c
bsic	--	pulse Interface Clear (IFC) for 100msec
bsre	b --	if b=0 unassert Remote Enable else assert it
bstat	—	store GPIB interface status in the variable

		ibret
bwait	mask --	wait for any event specified by mask to occur
bwrt	L c --	write c bytes from long address L to GPIB
c!	c a -	store byte c at ds:a
c!e	c e -	store byte c at es:e
c!l	c s a -	store byte c at s:a
c%	c -	append byte c to latest machine code definition
c,	c -	append byte c to latest high-level definition
c, words	c -	append byte c to latest vocabulary entry
c/l	- uv	characters per line
c@	a - c	place character at ds:a on stack
c@e	e - c	place character at es:e on stack
c@l	s a - c	place character at s:a on stack
cfa	pfa - cfa	replace parameter field address with code field address
clear	u -	clear screen u to blanks
clr	dev --	send message Selected Device Clear to device dev
cmove	al a2 i -	move i bytes from block at ds:al to ds:a2, begin at lowest address
cmovel	sl al s2 a2 i -	move i bytes from block at sl:al to s2:a2, begin at lowest address
code xxx		initiate machine code, assembler, and/or optimizer definition
codes	- a	address of code segment descriptor
compile	c	append 16 bit value which follows compilation address of compile to latest definition
constant xxx	n -	define constant xxx which returns n
context	- uv	address of vocabulary for searches
convert	dil al - di a2	accumulate number string at al into di, leaving address of first nonconvertable character
copy	ul u2 -	copy screen ul onto screen u2
copy-seg	sdl sd2 -	copy segment for descriptor at sdl to segment defined at sd2
count	a - a+1 #	get character count for string at ds:a
countl	s u - s u+1 #	get character count for string at s:u
cr		emit carriage return linefeed
create xxx		enter xxx into current vocabulary initialize code field as variable
csp	- uv	data stack position marker
cstep	a - a+1 c	increment address and return character at ds:a+l
current	- uv	address of vocabulary in which to enter new definitions

d!	d a -		store double word d at address ds:a
d!l	d s a -		store double word d at address s:a
d+	dl d2 - d3		sun double nunbers dl and d2
d+-	dil i - di2		return dil, negated if i was negative
d-	dl d2 - d3		subtract double number dl from d2
d.	di -		type double integer di according to base with one trailing blank
d.r	di u -		type di right aligned in field of u characters
d0=	d - f		true if double word d is zero
d<	dil di2 - f		true if dil less than di2
d=	dl d2 - f		true if dl equals d2
d>	dil di2 - f		true if dil greater than di2
d@	a - d		return double number at ds:a
d@l	s a - d		return double number at s:a
d_eos	n -- d		arg for 'get'. End of String character.
d_eot	n/a		a field in the device configuration table. if non-zero, the END message is sent automatically with the last byte of each write operation. if the value is zero, END is not sent
d_pad	n -- d		erg for 'get'. primary GPIB address.
d_sad	n -- d		erg for 'get'. secondary GPIB address.
d_tmo	n -- d		erg for 'get'. time limit.
dabs	di - du		absolute value of double integer di
dconstant xxx	d -		define double constant xxx which will return d
ddrop	d -		drop top two words from stack
ddup	d - d d		duplicate double number
decimal			set base to decimal
definitions	-		change current vocabulary to context
depth	- u		return count of words on data stack
digit	c n - u t f		return binary digit corresponding to ASCII c using base n or false flag if invalid
dliteral	d -	ic	compile double nunber d on stack as a literal
dmax	dil di2 - di3		keep larger of two double numbers
dmin	dil di2 - di3		keep smaller of two double nunbers
dnegate	di --di		two's complement of double nunber
do	il i2 -	c	begin do loop with index limit i1 index starting at i2
does>		ic	define run time action of defining word (high-level)
dover	dl d2 - dl d2 dl		copy second double nunber on stack
dp	- n		short for dpl @
dpl	- a		user variable, number of digits after decimal for last number
drop	n -		drop top number from stack
drop- seg	sd - sd		extract segment from chain, enlarging one

			below
ds!	u -		load ds register to use alternate definition list segment
dswap	dl d2 - d2 d1		swap two double nunbers on stack
du<	dul du2 - f		true if dul below du2
dump	s u # -		dunp # bytes after address s:u
dup	n - n n		duplicate top nunber on stack
dup@	a - a n		return nunber at address ds:a preserve address
dup>r	n - n		copy n to return stack
dupc@	a - a t		return character at ds:a preserve address
dvariable	xxx -		define double variable xxx
else		ic	follows true part and precedes false part of if
emit	c -		type character c
empty			clear dictionary of user defined words and reclaim segments
enclose	c s a - a nl n2 n3		for text at s:a and delimiter c find offsets first non-delimiter character nl first delimiter after text n2 first char excluded n3
endc	a		address of code segment working end
end- code			terminate machine code, assembler and/or optimizer definition
endl	a		address of working end of definition list segment
endw	a		address of working end of current vocabulary
es!	s -		store segment paragraph numbers into es register
execute	a -		execute compilation address a
execute@	a -		execute compilation address at ds:a
exit		c	terminate execution of colon definition
expect	a u -		input u characters or until cr received to ds:a
fence	a		address of word below which forget is disabled
fill	a u c -		fill u bytes after ds:a with byte c
fill1	s a u c -		fill u bytes after s:a with byte c
find xxx	a		return code field address of xxx or 0 if xxx is not found
find-lpa	xxx - a		return link pointer address of xxx or 0 if xxx is not found
flags>	flags		return current processor flags
forget xxx			delete xxx and beyond from current vocabulary
forget-task	a -		forget words and segments defined after task descriptor at a
ibcl		i	set context to ibcl vocabulary
ibcls	a		address of segment descriptor for ibcl

				vocabulary segment
here		- a		working end of definition lists
here-c		- a		working end of code segment
here-w		- a		working end of current vocabulary
hex				set base to hexadecimal
hld		- uv		address of latest character of text for numeric output
hold	c -			insert c in pictured numeric string
hop.	pfa -			macro, execute high-level word
i		- n	c	return loop index
i'		- n	c	return loop index buried one deep
i!	n a u -			store word n at ds:a+u
i!e	n e u -			store word n at es:e+u
i!l	n s a u -			store word n at s:a+u
i+!	n a u -			add word n to word at ds:a+u
i+!e	n e u -			add word n to word at es:e+u
i+!1	n s a u -			add word n to word at s:a+u
i@	a u - n			place word at ds:a+u on stack
i@e	e u - n			place word at es:e+u on stack
i@l	s a u - n			place word at s:a+u on stack
ic!	c a u -			store char c at ds:a+u
ic!e	c e u -			store char c at es:e+u
ic!1	c s e u -			store char c at s:a+u
ic@	a u -			place char at ds:a+u on stack
ic@e	e u -			place char at es:e+u on stack
ic@l	s a u -			place char at s:a+u on stack
id!	d a u -			store dword d at ds:a+u
id!e	d e u -			store dword d at es:e+u
id!1	d s a u -			store dword d at s:a+u
id@	a u - d			place dword at ds:a+u on stack
id@e	e u - d			place dword at es:e+u on stack
id@l	s a u - d			place dword at s:a+u on stack
if	f -		ic	if flag true, execute following words until else or then
immediate				make last word defined imnediate
in$		- pad		prompted string input
in#		- d		prompted double number input
intcallf	a b c d #i - a' t f			like intcall, for interrupts that indicate success by setting carry flag to zero
interpret				interpret input stream at offset >in characters into blk
is name	n -		i	set parameter field to n
j		- n	c	return index of next to innermost do loop
label-far xxx	s a -			define label xxx to return

			long address s:a
last-seg	- a		pointer to first word of segment descriptor most recently defined
latest	- e		pointer to name field address of last word in current vocabulary
leave		c	set do loop limit and index equal forcing exit
limit	- a		highest address in ds segment
lists	- a		address of descriptor for definition list segment
lit	- n	c	return contents of next list address
literal	n -	i	compile n as 16 bit literal
110	dev --		send the message Local Lockout (LLO).
load	n -		interpret screen n
loc	dev --		send the message Go To Local (GTL).
locate-word	cfa - lpa t f		find word in context vocabulary with lpa pointing to this cfa else return false
loop		ic	terminate do Loop index has positive unit step
lpa	al - a2		convert name field address to list pointer address
m*	il i2 - di		double signed integer product of signed single integers
m*/	dil il i2 - di2		triple precision product of dil & il divided by i2, rounded toward 0
m+	d l n - d 2		mixed precision sum
m/	di nl - n2 n3		di/n1 signed remainder n2 signed quotient n3
m/mod	dul u1 - u2 du2		dul/ul remainder u2, quotient du2
make-list	xxx #		create alternate list segment with # bytes
match			skip next higher level list nesting
max	il i2 - i3		return greater of il or i2
memory	- a		address of memory descriptor
min	il i2 - i3		return lesser of il or i2
mod	il i2 - i3		remainder of n1/n2 with sign of nl
move	al a2 n -		move n words from block at ds:al to block at ds:a2
move- seg	# sd - # s		create new segment with # bytes for segment defined at sd and relink
negate	i - -i		two's complement of i
new- list	# -		redefine list segment with # bytes
new - voc	# -		redefine context vocabulary segment with # bytes available
next			noop
nextw	- s a		return segment and address of first free word in current vocabulary
nfa	lpa - nfa		convert list pointer address to name field

			address
nop		i	noop
not	n - f		true only if n is zero
number	a - di		convert string at ds:a to di
offset	- uv		base to add to block number to get physical block number
okc			accept code allocation
oklw			accept list and vocab allocation
oldc	- a		verified end of code segment
oldl	- a		verified end of definition lists
oldw	- a		verified end of current vocabulary segment
onl	dev b --		if b=O take dev offline else reset software associated with device dev to its original online state
onldev	b --		if b=O take dev offline else reset software associated with device dev to its original online state
option name	-		create conditional comment
or	nl n2 - n3		nl or n2
out	- uv		value incremented by emit
over	nl n2 - nl n2 nl		copy second element to top of stack
p!	c u -		output byte c to port u
p@	u - c		input byte c from port u
pad	- a		address of scratch area
pct	dev --		pass control to device dev.
pfa	nfa - pfe		convert name field addr to parameter field addr
pick	u - n		return u'th stack element to top
pp xxxxxxxx	u -		put text on line u of latest screen listed
ppc	dev n --		send parallel poll configure message n to dev
prev	- a		return address of disk buffer most recently used
prev- seg	sdl - sd2		link back to previous segment descriptor
pw!	n u -		output word n to port u
pw@	u - n		input word n from port u
query			transfer line from terminal to input buffer
quit			clear return stack, set execution mode
r>	- n		transfer top of return stack to top of data stack
r@	- n	c	copy top of return stack to data stack
rd	dev L c --		read c bytes from dev into long address L
repeat		ic	terminates begin -- while -- repeat
roll	u -		transfer u'th word in stack to top
rot	n1 n2 n3 - n2 n3 n1		transfer 3'rd word in stack to top
rp!			clear return stack
rp@	- u		return the return stack pointer

rpp	dev --	request parallel poll. if 'ibret' shows no error a valid poll response is stored in 'ibppr'
rsp	dev --	request serial poll from device dev. if 'ibret' shows no error a valid serial poll response is stored in 'ibspr'
s->d	i - di	sign extend single precision integer to double
s*s	sc1 sc2 - sc3	scaler multiplication
s/s	sc1 sc2 - sc3	scaler division
s/drv	- a	address of table of number of sectors per drive
so	- u	return address of bottom of stack
sc-sqrt	sc1 - sc2	scaler square root
screen-open	-	open screen file using fcb #0 and
seg-size	- uv	default vocabulary segment size
seg	- u v	temporary storage for segment descriptor address
segment xxx	u -	create segment xxx & allot u bytes
sign	i -	if i is negative, insert minus sign in pictured numeric output
sp!		clear data stack
sp0	- a	address of initial value of data stack pointer
sp@	- as	return address of top of stack
space		type ASCII blank
spaces	u -	type u ASCII blanks
ss!	u -	use alternate stack segment
stacks	- a	address of stack segment descriptor
state	- uv	compile/execute flag
step	a - a+2 n	add 2 to address and return word at ds:a+2
swap	nl n2 - n2 nl	swap top two words on stack
task xxx		define lower boundary of task xxx
text	c -	transfer c/l characters or until c from input stream to pad
then		ic terminate if -- else -- then or if -- then
tib	- uv	address of terminal input buffer
trg	dev --	address device dev and send Group Execute Trigger
type	a# -	type # characters beginning at ds:a
typel	s u # -	type # characters beginning at s:u
u*	ul u2 - du3	unsigned product of ul and u2
U.	u -	type unsigned integer u converted using base
u.8r	n -	type n right justified in field of 8 characters
u.r	u # -	type u right justified in field of # characters
u/mod	du ul - u2 u3	divide du by ul return remainder u2,

			quotient u3
u<	ul u2 - f		true if ul is below u2
until	f -	ic	terminate begin -- until loop repeat until flag is true
user-base	- a		constant: address of base of user variable area
update			mark most recently referenced block altered
use- list	sd -		use alternate list segment
user xxx	u -		define new user variable xxx at offset u
variable xxx	-		define variable xxx
vectors name n	-		create execution vector table
voc- link	- uv		address of voc-link field in most recently defined vocabulary
vocabulary xxx	-		create vocabulary xxx and allot segment xxx-seg of size seg-size to contain it
w%	n -		append word n to latest machine code definition
wait	dev mask		wait for selected GPIB events to occur
wait	mask --		wait for selected GPIB events to occur
warning	- uv		flag to suppress non-fatal errors
while	f -		continue begin -- while -- repeat loop until f becomes false then continue after repeat
width	- uv		nunber of valid characters for name
word	c - a		accept characters from input stream until c or end of line transfer to ds:a
word- seg	al - a2		return segment descriptor address for vocabulary with vocabulary descriptor at ds:al
wrqs	dev mask n -		wait for device dev to request service with a positive response such that the logical AND of the response and maskbd = nbd. if 'ibret' shows no error a valid poll is stored in 'ibspr'
wrt	dev L c --		write c bytes from long address L to device dev
xor	ul u2 - u3		exclusive or of u1 and u2
[		i	end compilation mode, begin execution mode
[compi lel xxx	-	ic	force compilation of following word
			remainder of line (to cr) is comnent
1			set compi tat ion mode

APPENDIX D - IBCL TUTORIAL

INTRODUCTION

IBCL is the programming language that programs the NEC-V50 in the MacBus expansion box. The IBCL interpreter resides in the MacBus and serves as both the native language and the operating system of the box. The users of IBCL include OEMs who have custom applications for MacBus, and experienced users who wish to access the full power of the onboard NEC-V50. Most users will use the language interface libraries for their programming.

THE LANGUAGE

Tutorial, is a brief tutorial which takes you through most of the highlights of the language. It uses the MacBus utility, IBCL Window.

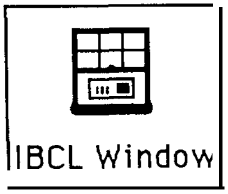
IBCL opens up all the resources of the MacBus interface to a single user. For this reason it is recommended that when developing applications with IBCL, no other GPIB processes should be active.

TUTORIAL

Starting the IBCL Window

In the following examples, what you type is in **boldface,** what the computer displays is in ***italics.*** The character ' ❑ ' means press the spacebar. The symbol '**<CR>**' means press the carriage-return.

Start the **IBCL Window** by double-clicking the IBCL Window icon.



Refer to SECTION TEN - IBCL WINDOW for more information on the IBCL Window utility.

The **IBCL** operating system responds with *ok* after a successful operation. Type **<CR>** a few times to make sure IBCL is there. IBCL should respond:

```
ok
ok
ok
ok
```

IBCL uses a push-down stack to store the numbers entered. Try entering the following. Be sure to put a space after every character, including the dots (.).

```
1 □ 2 □ 3 □ 4 □ . □ . □ . □ .<CR>
```

After you enter a **<CR>** the line should look like:

```
1 2 3 4 . . . .
4 3 2 1 o k
```

IBCL uses the dot character **(.)** (referred to as dot), to pop the top of the stack and print its value. Four dots will print the top four numbers on the stack. The numbers are printed 4 3 2 I, not *1* 2 3 4, because the numbers are pushed on the stack in the order they are entered. IBCL waits to echo the **<CR>** until after it reports *ok.*

To add the two numbers 9 and 5 together enter:

```
9❑ 5❑ +❑ .<CR>
```

The answer E will be displayed. The answer is E not 14 because the default base is 16 and E is 14 in hexadecimal. To change the base to decimal enter:

```
decimal<CR>
```

Try entering the following, with no space between 5 and the plus sign **(+)**.

```
9❑ 5+ ❑ .<CR>
```

IBCL will respond:

```
5+ : Not recognized
```

IBCL thinks in terms of *words,* where a *word* is any sequence of characters separated by a space or a <CR>. Since there is no space between the 5 and the +, IBCL thinks **5+** is one *word.* Note that since there was an error, IBCL did not report *ok.*

If you type **.** *(dot) you* will get the message:

```
. : Short stack
```

Short stack means that there are no numbers on the stack. The reason is that IBCL clears the stack after an error.

Defining New Words

The real power of IBCL is its ability to quickly add new *words* to its *dictionary.* The *dictionary* is the list of IBCL functions.

Define a new *word* called **3add,** which will add together the top three numbers on the stack together. Enter:

```
:❑ 3add❑ +❑ +❑ ;<CR>
```

Now, you try it:

```
5❑ 6❑ 7❑ 3add❑ .<CR>
18 ok
```

IBCL returns the answer 18 (decimal).

A new word is defined with a sequence starting with a colon (:). The first word after the colon is the name of the new word. The remaining *words*, up to the semicolon (;) are the body of the new *word.*

Now make a new *word,* **3addshow,** which adds the top three numbers on the stack and prints out the result. Enter:

```
: □3addshow□."□The□answer□is□"□3add□.□;<CR>
```

Now, you try it:

```
3 ❑  4 ❑  5 ❑  3addshow<CR>
The answer is 12 ok
```

Notice that **3addshow uses 3add, which is now part of the** *dictionary.* The *word* **."** will print out the characters up to the next **"** exactly as they are entered.

Loops and Conditionals

Make a new *word* **doline which will loop 5 times and print out the message** *line i,* **where** *i* is the loop count. Enter:

```
: □ doline□5□0□do□cr□."□line"□i□.□loop□;<CR>
```

Now you try it:

```
doline<CR>
line0
line1
line2
line3
line4 ok
```

do takes two arguments, a terminal count and an initial count. The word loop increments the index and loops back to do if the index is less than the terminal count. The word **i** pushes the current value of the index onto the stack. The word cr performs a <CR>.

Looping and conditional constructs only work within a new ***word*** definition.

In IBCL, a TRUE value is any non-zero value; a FALSE value is a zero value.

The word **if** checks the top number on the stack and conditionally executes ***words*** based on the TRUE/FALSE value of the top number. Define a new ***word*** called tf which will determine whether a number is TRUE or FALSE. Enter:

```
: □tf□if□."□TRUE□"□else□."□FALSE□"□endif□;<CR>
```

Now you try it a few times:

```
1❑ tf<CR>
TRUE ok
0❑ tf<CR>
FALSE ok
-1 •I tf<CR>
TRUE ok
9❑ 9❑ +❑ tf<CR>
TRUE ok
9❑ 9❑ - ❑ tf<CR>
FALSE ok
7❑ 4❑ =❑ tf<CR>
FALSE ok
7❑ 4❑ >□tf<CR>
TRUE ok
```

In the second to last example, the equal sign (=) tests the equality of the top two numbers on the stack. In the last example, the greater than sign **(>),** tests whether the second number on the stack is greater than the top number on the stack.

Stack Words

Sometimes the order of numbers on the stack is not what you want, or sometimes you need to check a value on the stack without losing it. There are several stack words that fix these problems.

Enter:

```
1❑ 2❑ 3❑ 4❑ swap❑ . ❑ . ❑ . ❑ .<CR>
```

The result will be 3 4 2 *1* because swap swapped the top two numbers on the stack. Enter:

```
2 •I dup •I . ❑ .<CR>
```

The result will be 2 2 because dup duplicated the top number on the stack.

Another stack word, drop, drops (pops) the top number from the stack.

More Looping

With the words encountered so far, we can explore some of the more complicated looping structures.

Enter the following three lines:

```
: □eq4□dup□4□=□;<CR>
:   □ ndec□."□going□"□1□-□eq4□;<CR>
:   □ beg1□7□begin□ndec□until□."□gone□"□drop□;<CR>
```

Now you try it:

```
beg1<CR>
going going going gone ok
```

The word begin marks the start of a non-iterated loop. until checks the first number on the stack, which is the result from ndec, and if it is FALSE loops back to the begin statement-that is, it loops until ndec returns TRUE.

ndec prints out the string "going " then subtracts one from the top number on the stack. It then calls eq4 which will return a TRUE/FALSE value indicating whether the top number of the stack is equal to 4.

eq4 duplicates the top number on the stack and compares it to 4. The number must be duplicated because the compare with 4 will pop the top two numbers off the stack when it does the compare.

After the begin . . . until the number 4 was still on the stack; the drop gets rid of it.

Forgetting

If you have tried reentering a word definition, you have seen the message:

```
xxxx : Isn't unique
```

IBCL does not replace a previous word with the new one, it remembers both, but uses the most recently defined. To revert back to the previous version use forget. Try entering the following sequence:

```
:□ ver □." ❑ version ❑ 1❑ " ❑ ;<CR>
ok
:□ ver □." ❑ version •I 2 ❑ " ❑ ;<CR>
ver : Isn't unique ok
ver<CR>
version 2 ok
forget ver<CR>
ok
ver<CR>
version I ok
forget ver<CR>
ok
ver<CR>
ver : Not recognized
```

The last forget removed the original definition. forget removes the requested *word* and all *words* that were defined since that *word.*

GPIB **Functions**

If you do not have GPIB devices handy omit this section. To use a GPIB, device you must have previously configured one using the utility IBCONF. For this example, assume there is a digitizer named *dig* already configured. Before you can use *dig you* must open it:

```
ibfind ❑ dig<CR>
```

If no error was reported dig is now ready for use. To clear device dig, enter:

```
dig ❑ clr<CR>
```

All device functions require the name of a device as their first argument. To write a command to device *dig* enter:

```
dig ❑ " ❑ cap13;25" ❑ wrt<CR>
```

The double quote character (") creates a buffer with the text **'cap13;25'** in it. It leaves on the stack the long address of the buffer and its count. These arguments are in the correct order for the **wrt** function.

Exiting the IBCL Window

To exit the **IBCL Window,** type "stop". This returns you to the Macintosh Plus Finder.

APPENDIX E - SCALED NUMBERS

The NEC 72191 floating point processor is an optional feature on MacBus, so every user will not have one. IBCL provides a software floating point capability for the user who does not have a coprocessor. This section documents this capability.

To use the software floating point facility, the utility file SCALER must be downloaded to MacBus. Do this using the IBCLload utility provided on your distribution disk or using the IBCLload function in the Megamax C MacBus Support Library (purchased separately). The file contains the definitions of words to support software floating point. This section describes these words.

An IBCL software floating point number is stored on the stack in an easy-to-manage format. The mantissa is stored as a double precision (i.e., 32 bit) number and the exponent is stored as a single precision (16 bit) number. When such a number is on top of the stack, the exponent is on top and the mantissa is beneath it. For example, if you give IBCL the following input:

```
32. 5
```

a floating point number representing 32.0E+5 will be on the stack. Note that the decimal point forces this to be a double precision number.

Addition and subtraction using floating point is significantly slower than integer double word addition. If the magnitude of numbers is limited, a fixed scaling factor and normal double word arithmetic will be much faster. For instance, financial calculations might need to be done in tenths of a cent. If the maximum dollar value computed at any one time can be kept below $2,147,483.64 there is no need for floating point of any kind. This range can be doubled for unsigned numbers.

We call this data type scaled because it is not a true binary floating point. True binary would require a base two exponent.

Addition and subtraction using base 10 floating point instead of true binary is slower. However, converting numbers to true binary and back to display format is even slower. Using scaled decimal is a good compromise if the average number of additions and subtractions per number is not large.

A scaler is entered onto the stack from the keyboard by first entering a double word number and then retrieving the exponent from dpl. dp can be used to retrieve the exponent. If used in a colon definition, it also normalizes the number.

```
12345.678 dp            . d.

: test 12345.678 dp ;    test . d.
```

The number entered must fit on one line but may have any number of digits and the decimal point may be located anywhere. The number stored in dpl is the power of ten by which the double word integer must be multiplied to produce the original value. Therefore, if the number is too large or too small to be easily entered, it can be entered in "scientific" notation. Suppose we want to enter 5.76 * 10 ** 339 :

```
5.76 dp 339 +
```

This produces the same number that would have been produced if we could have typed 576 followed by 337 zeroes. Similarly, 5.76 * 10 ** -339 can be entered as:

```
5.76 dp 339 -
```

Very large and very small numbers can be displayed in a similar notation:

```
rot rot d.  ." * 10** "  .
```

This displays 576 *10** 337 or 576 * 10** -341. Usually we would use ." e" instead of ." * 10 ** " to be consistent with conventional computer abbreviations.

The usual variable, constant, store, and at words are provided, and assume the stack items covered above.

```
nnn.nnn dp      x-constant  name
                scaler      name
                SC!
                sc@
```

Given two scaled values (6 words) on the stack, we can perform addition and subtraction.

```
sc+    ( scaler1 scaler2 -- scaler3 )
sc-    ( scaler1 scaler2 -- scaler3 )
```

Multiplication and division require a scaled value on the stack with a simple signed integer on top of the stack. This provides an efficient multiplication and division for cases such as some dollar amount being multiplied by the total number of units, or an average of a sum of entries.

```
sc*    ( scaler1 integer -- scaler2 )
sc/    ( scaler1 integer -- scaler2 )
```

A pair of scaled values on the stack can also be multiplied or divided.

```
s*s    ( scaler1 scaler2 -- scaler3 )
s/s    ( scaler1 scaler2 -- scaler3 )
```

The square root of a scaled value can also be calculated. Only four decimal digits will be significant. If the scaled value is negative, the square root of its absolute value will be returned.

```
sc-sqrt        ( scaler1 -- scaler2 )
```

MEMORY

ccc@ ccc!	At and Store are prefixed by the desired in memory data format. The address is on the IBCL data stack

as usual, and is dropped after the command. The source or destination of the data is the IBCL stack. Valid forms are:

```
i16@ i32@ i64@ r32@ r64@ r80@ bcd@ ( addr - :: - x )
i16! i32! i64! r32! r64! r80! bcd! ( addr - :: x - )
```

Be certain that the memory allotted for the variable names used is sufficient. **bcd@ bcd! r80@** and **r80!** all require five words for each memory location. (10 bytes)

```
variable r80-var 8 allot    (2 allotted automatically)
```

STACK

```
fdup    ( :: r80 -- r80 r80 )
fdrop   ( :: r80 -- )
fswap   ( :: y x -- x y )
fover   ( :: yx -- y x y )
frot    ( :: z Y x -- y x z )
```

COMPARISON

Comparisons follow the usual IBCL convention of comparing the second number on the stack to the one on top, popping both, and returning a flag on the data stack. The numbers being compared are both on the NEC 72191 stack.

```
f<    f=    f>         ( -- flag :: y x -- )
f0<   f0=   f0>        ( -- flag ::   x -- )

fmax                   ( :: y x -- maximum )
fmin                   ( :: y x -- minimum )

fxam                   ( -- status :: x -- x )
```

Refer to assembler for definition of Status word.

CONSTANTS

The following words place the value of a constant on the fstack.

```
f=0              ( :: -- 0               )
f=1              ( :: -- 1               )
f=pi             ( :: -- pi              )
f=l2(10)         ( :: -- log of 10 base 2  )
f=l2(e)          ( :: -- log of e base 2   )
f=log(2)         ( :: -- log of 2 base 10  )
f=ln(2)          ( :: -- log of 2 base e   )
```

MATH

Math functions follow the usual IBCL conventions where the top number on the stack is subtracted from or divided into the second, both arguments are dropped and the result is returned. For unary functions, the result replaces the argument.

Note: Stack order of arguments is determined by standard IBCL conventions, not NEC 72 19 1 conventions.

Unary

```
fsqrt   ( :: x -- square root of x )
          -0 <= x <= +infinity
f**2    ( :: x -- x * x )
fabs    ( :: x -- absolute value of x )
fnegate( :: x -- -x )

frndint ( :: x -- x rounded to integer toward 0 )
```

This word can be used to convert very small (unnormal) numbers to 0. The high level transcendental functions included in IBCL perform this function whenever necessary.

Binary

```
f+      ( :: y x -- y+x )
```

```
f-     ( :: y x -- y-x )
f*     ( :: y x -- y*x )
f/     ( :: y x -- y/x )
fmod ( :: y x -- remainder(y/x) )
         fmod is a floating point modulo operation where
         x is divided into y an integral number of times.
         The remainder is returned on the fstack, while
         the lowest order 3 bits of the quotient are
         returned in the NEC 72191 Status word.
         ( see fprem. )
```

TRANSCENDENTAL

The acceptable range for arguments for transcendental functions has generally been extended to the full number line with four exceptions:

1. Never use infinity as an argument.
2. The absolute value of an effective power must be less than 2**15.
3. Do not use zero as the argument of a log function.
4. Do not use an unnormal as an argument.

Exponentials

```
fscale ( :: y sf -- y*2**sf )
         Scale y by scale factor sf, which is rounded to an integer
         toward 0. If sf is between plus and minus 1, it must be
         exactly 0.
         -32768 <= sf < 32768

f2**   ·( ·    x --  2**x )  abs(x)<2** 15
f10**  ( II    x - - 10**x )  abs(x) < 2**15/log2(10)
f**    ( :: y  x --   y**x )  abs(x) < 2**15/log2(y)
                              0 < y < infinity
```

```
f*l2t  ( ::     x  --  x  * log2(10) )
f*l2x  ( ::  y  z  --  y  * log2(z) )
         0 < z < infinity     -infinity < y < infinity
```

Logarithms

```
f2log  ( ::  x  --  log2 (x) )   0 < x < infinity
fln    ( ::  x  --  ln.e (x) )   0 < x < infinity
flog   ( ::  x  --  log10(x) )   0 < x < infinity
```

Trigonometric

The trigonometric functions expect or return an angle in radians. The magnitude of the angle should be a normal number less than infinity but is not otherwise restricted.

ftriangle (:: x -- adjacent opposite)

This is the root word from which tangents, sines, and cosines are calculated. It returns the signed lengths of the opposite and adjacent sides of a right triangle with radius approximately 1.5 generated by the angle. The remaining forward trigonometric functions can be easily determined from these sides.

```
ftan   ( ::  x  --  tan(x) )
fsin   ( ::  x  --  sin(x) )
fcos   ( ::  x  --  cos(x) )
```

cosec x = 1 / sin x
sec x = 1 / cos x
cot x = 1 / tan x

farctan (:: x -- angle with tangent equal to x)

CONVERSIONS

These words move either double word integers or scaled decimal numbers between the IBCL data stack and the NEC 72191 floating point stack. If the top of the NEC 72191 stack does not contain a normalized number or zero for conversion to a scaler, a meaningless number will be produced. You can check for a valid number with **ftype** if presence of a valid number cannot be guaranteed.

```
>>f     ( double-nmber -- :: -- treal )
f>>     ( -- double-nunber :: treal -- )
sc>>fI  ( double-number scale -- :: -- treal )
f>>scI  ( -- double-nmber scale :: treal -- )
```

TYPE CHECKING

ftype (-- type :: x -- x)
Places a word on the IBCL data stack summarizing the characteristics of the top word on the 72191 stack. A bit is set in this word for every applicable characteristic.

bit	mask(hex)	characteristic
7	80	negative (sign bit set)
6	40	empty
5	20	not a number
4	10	infinity
3	08	normal, **nonzero**
2	04	zero
1	02	unnormal
0	01	normal or zero

INPUT/OUTPUT

Numbers are input in scaled decimal format. Enter an integer containing an appropriately placed decimal point, and follow with dpl **@** to give the base 10 scale factor. This will enter the scaled decimal number onto the IBCL data stack. It may then be moved to the **fstack** with **f>>sc** or stored in a **scaler** variable.

Integers can also be entered as either single or double integers and then be moved to the **fstack** with **f>>**. Single integers must first be converted to double with **s->d**.

Four output words are included for scaled numbers. They expect a signed double integer on the IBCL data stack, with a decimal scale factor above it. The sc. version prints the number with the decimal point in position. The se. version always places the decimal point to the right of the first significant digit and follows the number with e and the appropriate power of 10.

The **sc.r** and **se.r** version print a maximum number of digits right justified in a field of some minimum width. The number given for digits should be large enough to include the decimal point and the sign. It does not include the e or the exponent for the **se.r** version. The number is truncated to the maximum number of digits. If the field width is insufficient for the number, it is expanded to hold the complete number.

```
double-number scale sc.
double-number scale se.
double-number scale max#digits min-field-width sc.r
double-number scale max#digits min-field-width se.r

examples:   .000123456 dpl @         sc. 0.000123456
            .000123456 dpl @         se. 1.23456E-4
            .000123456 dpl @ 7 14 sc.r       0.00012
            .000123456 dpl @ 5 14 se.r        1.234e-4

            f=pi f>>sc          sc. 3.14159265
```

NEC 72191 CONTROL

finit	Initialize NEC 72191. (see **finit.**)
fldcw	(control-word --) Load control word from data stack into NEC 72191.
fstcw	(-- control-word) Push current NEC 72191 control word onto data stack.

fstsw (-- status-word)
Push current NEC 72191 status word onto data stack.

NEC 72191 Notes

If you overflow the NEC 72191 stack, you will lose more than just the bottom elements of the stack. The NEC 72191 will mark the top conflicting words of the stack empty as well. You cannot drop words from the bottom of the stack by writing over them as the stack wraps around.

```
1. >>f 2. >>f 3. >>f 4. >>f 5. >>f 6. >>f
7. >>f 8. >>f 9. >>f

f>> d. f>> d. f>> d. f>> d. f>> d. } (empty) 8 7 6 5
f>> d. f>> d. f>> d. f>> d. } 4 3 2 (empty)
```

Several high level words need NEC 72191 registers as scratch registers, reducing the total available for temporary storage.

f>>sc uses 4 NEC 72191 registers. The stack must contain no more than five numbers when invoking **f>>sc**.

```
sc>>f      f2**   f10**  f**
ftriangle  ftan   fsin   fcos   farctan    use 3

f2log fln  flog          dg>>rd rd>>dg     use 2
```

HIGH PRECISION I/O

If you are performing a significant amount of floating point I/O or if you need more than nine decimal digits of precision, the scaled decimal I/O package will not meet your needs. The binary coded decimal package, however, provides efficient I/O for floating point numbers through 18 decimal digits. Significant floating point functionality demands a NEC 72191, and the binary coded decimal package works using the NEC 72191. It is meant to be used in conjunction with the 72191ibcl high level support package or the 72191asm assembler extensions.

Do not be confused into thinking that this package limits you to use of binary coded decimal numbers. Binary coded decimal is simply an intermediate format for efficiently translating ASCII number strings and loading them onto the NEC 72191, or moving numbers from the NEC 72191 to ASCII string format for output. Our BCD format also includes provision for a base ten exponent over the full NEC 72191 range. All numbers on the NEC 7219 1 itself are in temporary real format. They may be transferred to and from system memory in any convenient format that is appropriate to the given number.

This package includes words to read and write binary coded decimal strings and move up to 18 digits with a base 10 exponent to and from the NEC 72191. It requires the NEC 72191 chip and some words from the 72191ibcl high level support package. Have the host download the files **72191ibcl** and **bcd,** respectively. See your host software reference manual for details.

Note: If you are just moving numbers off the NEC 72191 for storage, the most time efficient conversion is integer, followed by real and temporary real. The packed BCD form takes significantly longer to convert and should be used primarily for I/O. Even data files can often be written in one of the internal integer or real formats if they contain data for further processing and are not meant for direct human reading.

The NEC 72191 is an efficient means for processing floating point numbers — but if a job can done with single and double integers and simple arithmetic, use of the NEC 72191 will result in a decrease in performance.

The format for numbers to be read by the BCD package is fairly typical, except that it does allow for the optional use of commas.

The input string format is: sdd,ddd.ddd,ddesdddd

s	signs, optional
e	and exponent are optional
.	optional, right of number assumed if absent
,	all commas are ignored

d decimal digit, leading zeroes permitted
significant digits can be far before or after decimal

Some acceptable numbers are:

```
0        +0       -0       123      0123     12E5
-12.3E6         .00012
+123.456E-700
000,000,123,456
123456789123456789123456789.
1230000000000000000000000000000000.E100
.00000000000000000000000000000001234E123
.000,000,000,000,000,012,345,678,912,345,678,9
-000,,,,00,0,,0123.456,,,E+97
```

Numbers are written in a similar format, except that commas are never written and the sign of the exponent is always included explicitly. For positive numbers, a space is written instead of the plus sign.

The following pages describe the words which read, translate, and write floating point numbers.

bed-in (-- #digits)

> Read numeric string from input stream.
> Return number of significant digits read.
> The string is initially read to the **pad** buffer and the NEC 72191 format packed BCD representation is built below the pad buffer. The base 10 exponent is stored at pad.
> The 10 bytes below pad and the two at pad may be stored in a variable, but are usually loaded directly onto the NEC 72191 by **bcd>>f**.

Execution example: bed-in -123456789.12e7

First text reads the string to pad, then **$->bcd** converts it. Note that the base 10 exponent stored at pad is -2, since the packed BCD representation is viewed as an 18 digit integer. Stack return #digits (number of significant digits) is 11.

```
                 pad
text:            Of 31 32 33 34 35 36 37 38 39 2e 31 32 45 35

                                                pad
$->bcd:          00 00 00 20 91 78 56 34 12 80 fe ff
```

bcd>>f	(-- :: -- **r80**)
	Load the packed BCD number stored below pad and the base 10 exponent stored at pad onto the top of the NEC 72191 stack. This is usually done immediately after placing the packed BCD number with bed-in or moving it to the pad region from a variable. If you have loaded pad from a variable that retained fewer than all 18 digits, be sure to set the remainder of the pad transfer area to zeroes.
Note:	Execution of **bcd>>f** uses five NEC 72191 stack registers. Using more than eight stack registers will destroy both top and bottom word of the NEC 72191 stack.
f>>bcd	(-- :: r80 --)
	Convert the top number on the NEC 72191 stack to packed BCD representation below the pad buffer, and a base ten exponent stored at pad. Be sure to reformat and write the number, or move it elsewhere, before using the pad buffer for anything else.
Note:	Execution of **f>>BCD** uses 5 NEC 72191 stack registers. Using more than 8 stack registers will destroy both top and bottom word of the NEC 7219 1 stack.

bcd->$. (#digits -- pad)
bcd->$e (#digits -- pad)

Reformat the packed BCD representation below pad and the base 10 exponent at pad to an ASCII string and leave the address of the **pad** on the stack. Only #digits of the packed string are used. Zeroes will be used if more digits are required before the decimal appears using **bcd->$**

For **bcd->$e** the decimal always follows the first digit, and an e followed by the exponent follows after #digit digits.

The preceding words are most often used in pairs:

bed-in **bcd>>f**	and	**f>>bcd bcd->$.** count type
	or	**f>>bcd bcd->$e** count type

APPENDIX F - USING LABVIEW WITH MACBUS

This appendix provides examples that show you how to use MacBus with LabVIEW running on the Macintosh Plus. LabVIEW is a Laboratory Virtual Instrument Engineering Workbench for the Macintosh Plus. LabVIEW provides a complete integrated environment for scientific test and measurement applications involving instrument control, data acquisition, data analysis, data display, data management, and report generation. Because LabVIEW is a complete programming environment, it is also well suited to stand-alone applications involving scientific computation, modeling, and experimentation. LabVIEW provides you with interactive graphics, and a new graphical programming language resulting in a visual environment that is intuitive and easy to use, yet powerful and flexible enough for scientific applications. LabVIEW is functionally complete. Any application possible in a conventional programming language is possible in LabVIEW.

WHY USE LABVIEW WITH MACBUS?

As a programming environment, LabVIEW is far more intuitive, and far easier to use than a program written in conventional programs built from lines of textual code. In addition, LabVIEW is an interactive programming environment that allows you to build applications "bottom-up" in testing each piece along the way. With LabVIEW, you can test and build MacBus IBCL applications that can later be sent from a conventional type of program.

WHY USE **MACBUS** WITH **LABVIEW?**

MacBus provides the Macintosh Plus user with three key assets:

- A high-speed, intelligent IEEE-488 interface that can be programmed in a high-level language.
- Since MacBus is a complete, stand-alone computer, MacBus provides an attached processor to the Macintosh Plus that can run applications in IBCL **concurently** with the Macintosh Plus, and thus concurrently with **LabVIEW**. You can offload computational tasks from **LabVIEW** to the MacBus.
- MacBus is PC-AT hardware compatible, and provides you with three unused PC-AT slots in which PC/XT/AT compatible hardware may be placed. This hardware can then be controlled by IBCL and **LabVIEW.**

All of this means that you may interface the scientific programming environment of **LabVIEW,** to PC/XT/AT compatible hardware and operate the hardware concurrently with **LabVIEW**. For a given application, you have the flexibility to select the hardware, decide how much of the application will run on MacBus under the IBCL interpreter, and how much of the application will be run in **LabVIEW.** In addition, the MacBus processor is able to interpret 80186 machine code and thus applications can be developed on the IBM PC/XT/AT with the appropriate IBCL language interface, downloaded into the MacBus from the Macintosh Plus and called by IBCL. This means that existing assembly language drivers and routines can be ported to MacBus. (See MacBus Technical Reference Manual for specifications).

THE **LABVIEW** MACBUS INTERFACE

LabVIEW provides a built in set of instruments (**LabVIEW** terminology for graphical programs or subroutines) that allow you to access MacBus from **LabVIEW** with the same low-level support as supplied by the advanced function routines presented in the MacBus Software Reference Manual (SECTION SIX A - Microsoft BASIC or SECTION SIX B - Megamax C). You are able to send IBCL command strings to MacBus and receive IBCL responses from MacBus. You can also upload and download memory buffers to MacBus and poll the status of MacBus. The icons of the instruments provided by **LabVIEW** are shown below.

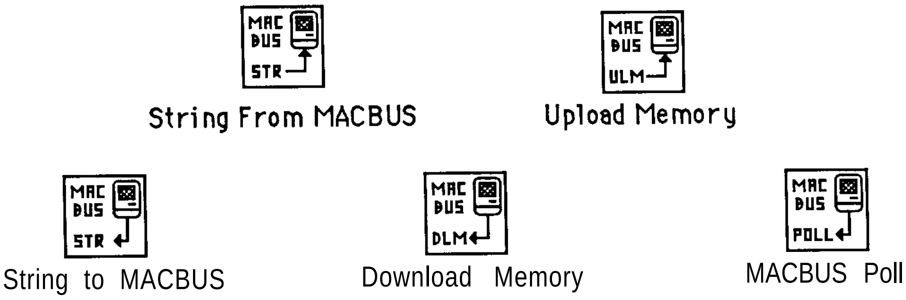


Figure F-l - LabVIEW/MacBus Interface Instrument Icons

String To MACBUS, String From MACBUS, Download Memory, Upload Memory, and MACBUS Poll perform the same functions as string out, **fout,** dlm, ulm, and poll in the MacBus Software Reference Manual. These instruments are explained in detail in the **LabVIEW** User Manual.

LABVIEW/MACBUS EXAMPLES

The remainder of this appendix assumes that you are familiar with the use of **LabVIEW** and its terminology. The remainder of this appendix shows instruments that National Instruments has developed for in-house use to run MacBus/LabVIEW applications. After reviewing these instruments, you should be able to build your own LabVIEW/MacBus applications.

Key IBCL Words - **alloc,** ulm, dlm

Key **LabVIEW** Instruments - String To MACBUS
String From MACBUS
Download Memory
Upload Memory
MACBUS Poll

The following three instruments are the building blocks of the remaining LabVIEW/MacBus interface examples. These instruments are called respectively: done poll, ulm poll, and dlm poll. They are built on top of the MacBus Poll instrument provided by **LabVIEW.** MacBus protocol requires that prior to executing any of the instruments, String To MACBUS, String From MACBUS, Download Memory, and Upload Memory, you must poll MacBus using the Poll MACBUS instrument in order to synchronize with MacBus. The instruments presented below perform the necessary polls.

Done Poll

Done poll repeatedly executes the MacBus Poll instrument and examines the MacBus Status bit vector array (which corresponds to **fstat** in the MacBus Software Reference Manual), until bit 0 (done) or bit 5 (error) is set, at which time the instrument completes and returns the error value (0 or 1). Subsequent to executing this instrument, you may execute the String To MACBUS or the String From MACBUS instrument. The done poll icon, connector pane, and connector pane controls are shown in Figure F-2. The done poll panel is shown in Figure F-3 and a diagram of the done poll instrument is shown in Figure F-4.

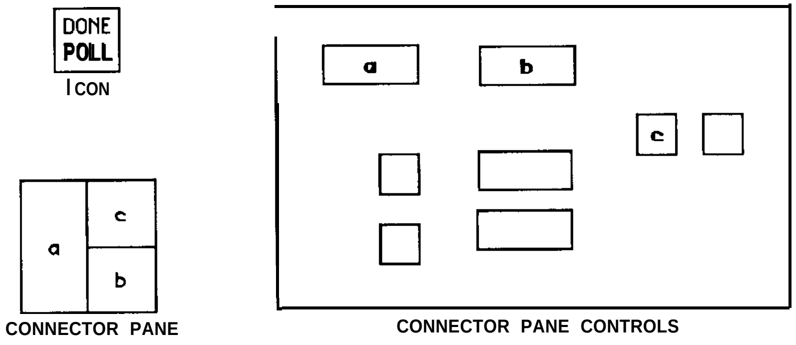


Figure F-2 - Done Poll Icon, Connector Pane, and Connector Pane Controls

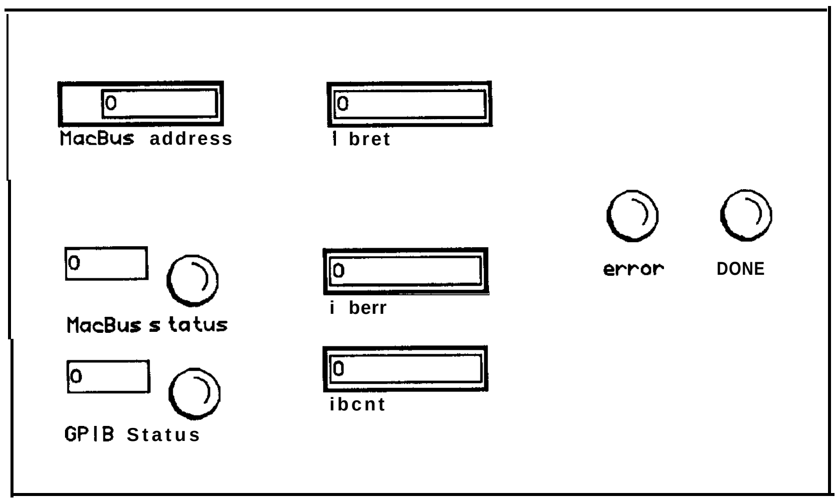


Figure F-3 - Done Poll Panel

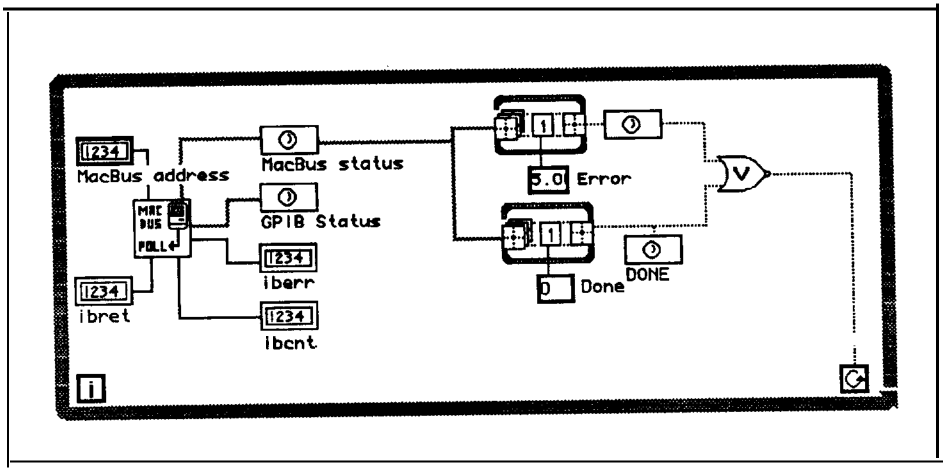


Figure F-4 - Done Poll Diagram

Ulm Poll

Ulm poll repeatedly executes the MacBus poll instrument and examines the MacBus Status bit vector array (which corresponds to **fstat** in the MacBus Software Reference Manual), until bit 2 (MacBus waiting for ulm) or bit 5 (error) is set, at which time the instrument completes and returns the error value (0 or 1). Subsequent to executing this instrument, you may execute the Upload Memory instrument. The ulm poll icon, connector pane, and connector pane controls are shown in Figure F-5. The ulm poll panel is shown in Figure F-6 and a diagram of the ulm poll instrument is shown in Figure F-7.

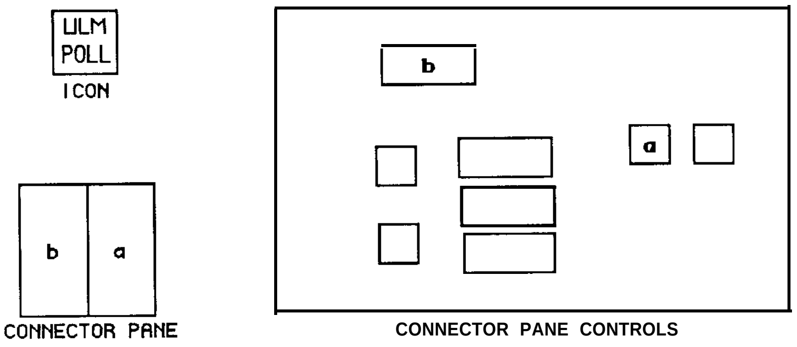


Figure F-5 - Ulm Poll Icon, Connector Pane, and Connector Pane Controls

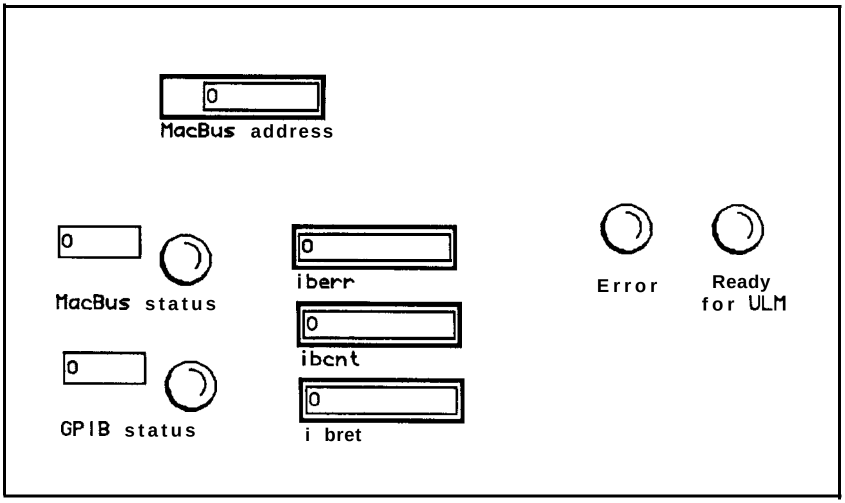


Figure F-6 - Ulm Poll Panel

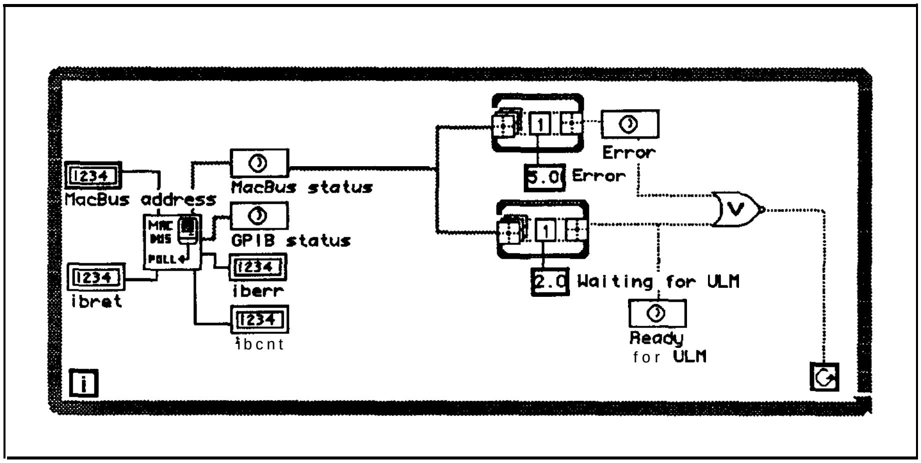


Figure F-7 - Ulm Poll Diagram

Dlm Poll

Dlm poll repeatedly executes the MacBus poll instrument and examines the MacBus Status bit vector array (which corresponds to fstat in the MacBus Software Reference Manual), until bit 3 (MacBus is waiting for dlm) or bit 5 (error) is set, at which time the instrument completes and returns the error value (0 or 1). Subsequent to executing this instrument, you may execute the Download Memory instrument. The dlm poll icon, connector pane, and connector pane controls are shown in Figure F-8. The dlm poll panel is shown in Figure F-9 and a diagram of the dlm poll instrument is shown in Figure F-10.

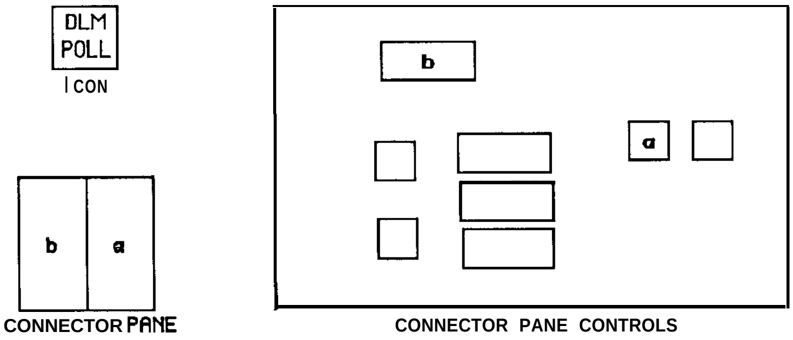


Figure F-8 - Dlm Poll Icon, Connector Pane, and Connector Pane Controls

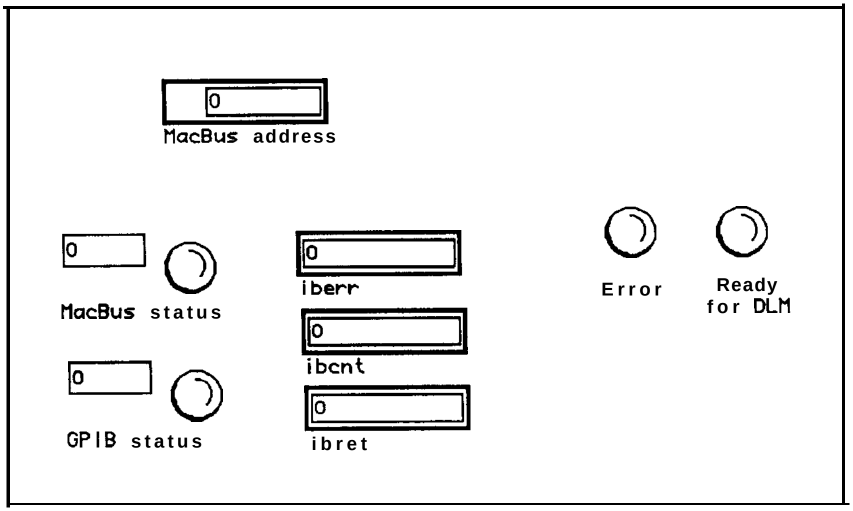


Figure F-9 - Dlm Poll Panel

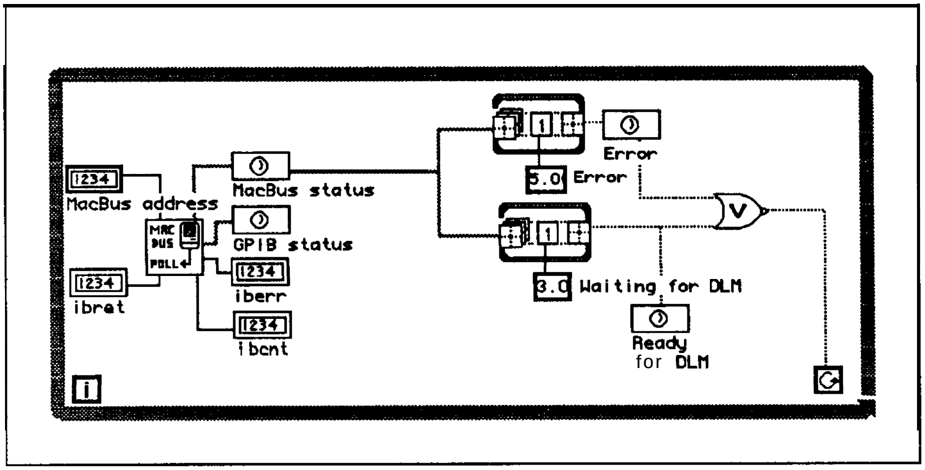


Figure F-10 - Dlm Poll Diagram

Higher Level MacBus Interface Instruments

The following five instruments are examples of higher level MacBus interface instruments, i.e., these instruments include the necessary poll before executing the MacBus input/output functions listed above. The instruments are: String to MacBus, String from MacBus, MacBus String I/O, MacBus DLM, and MacBus ULM.

String to ***MacBus***

String to MacBus accepts the MacBus address and a string as input. If the MacBus address is not supplied (connected), it uses the default MacBus SCSI address of 6. The instrument appends a NULL ASCII character to the string, executes the done poll instrument, and when completed, executes the String To MACBUS instrument. The instrument returns error information from String to MACBUS. The string to MacBus icon, connector pane, and connector pane controls are shown in Figure F- 11. The string to MacBus panel is shown in Figure F- 12 and a diagram of the string to MacBus instrument is shown in Figure F- 13.

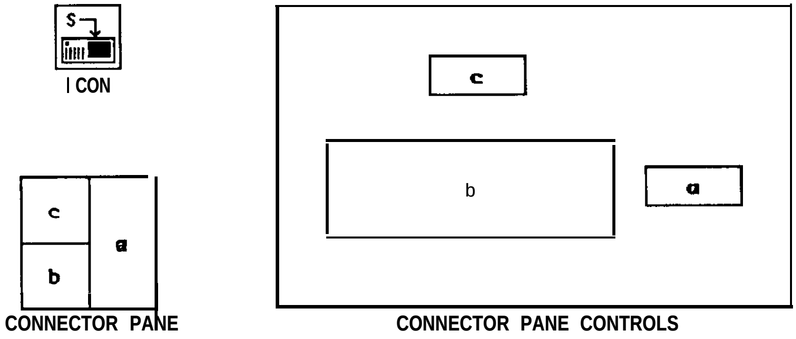


Figure F-11 - String to MacBus Icon, Connector Pane, and Connector Pane Controls

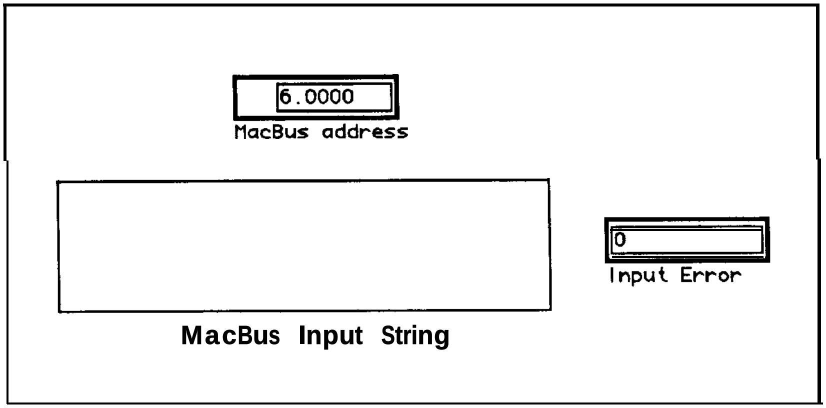
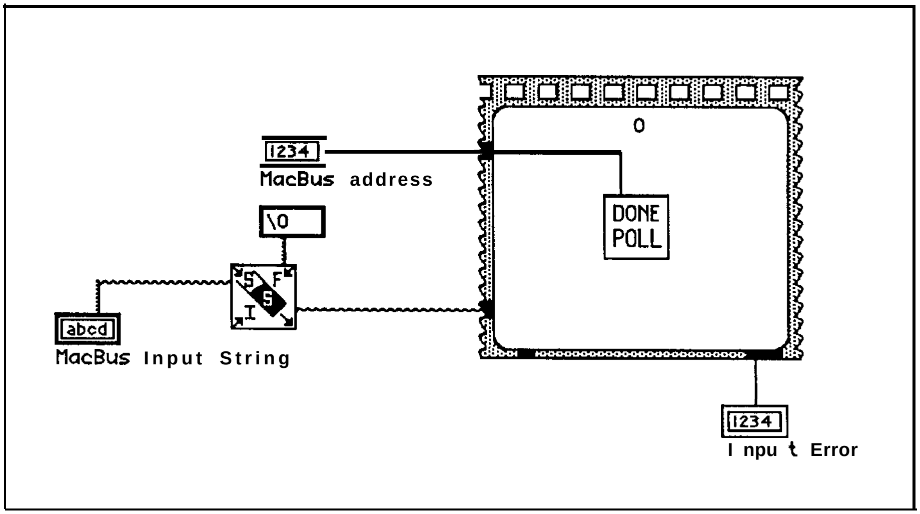


Figure F-12 - String to MacBus Panel



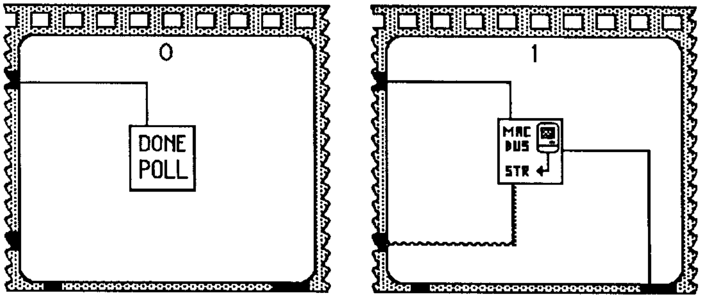


Figure F-13 - String to MacBus Diagram

String from MacBus

String from MacBus accepts the MacBus address as input. If the MacBus address is not supplied (connected), it uses the default MacBus address of 6. The instrument executes the done poll instrument, and when completed, executes the String From MACBUS instrument. The instrument returns the string and error information from String to MACBUS. The string from MacBus icon, connector pane, and connector pane controls are shown in Figure F-14. The String From MACBUS panel is shown in Figure F-15 and a diagram of the String from MacBus instrument is shown in Figure F- 16.

Note: IBCL will not return a string unless the following IBCL command string is placed at the beginning of the first string sent to MacBus:

```
find iaemit ‘emit !
```

This string instructs IBCL to place its responses in the IBCL output buffer that is read by the String From MACBUS instrument. It is only necessary to send this string once after MacBus is powered on or reset. Strings returned from IBCL are terminated by the phrase “ok” and a carriage return, linefeed, and space character.

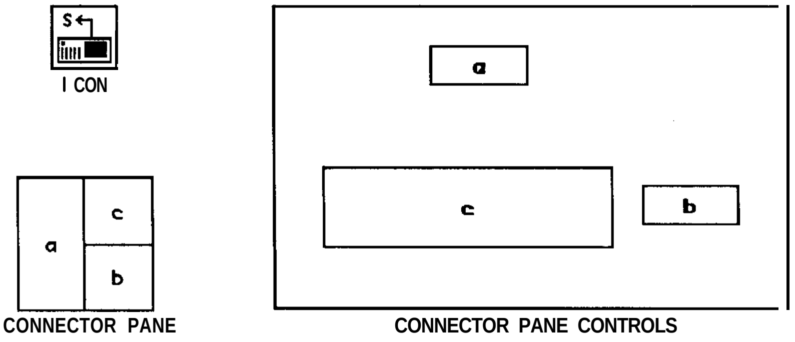


Figure F-14 - String From MacBus Icon, Connector Pane, and Connector Pane Controls

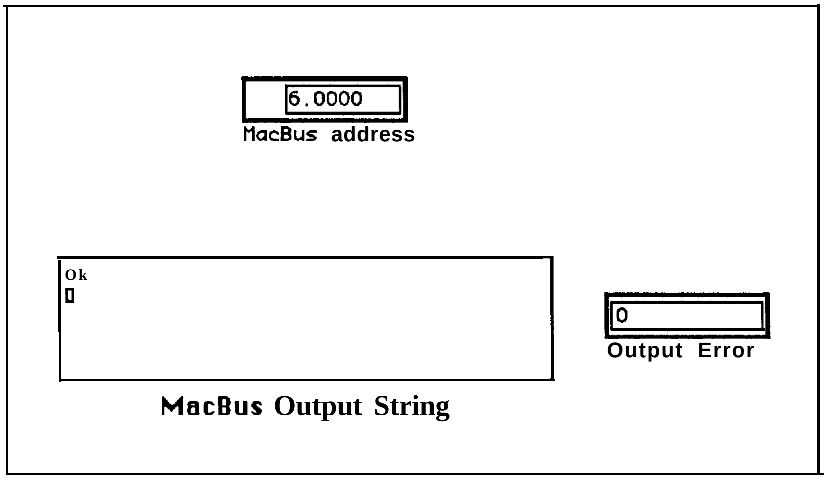
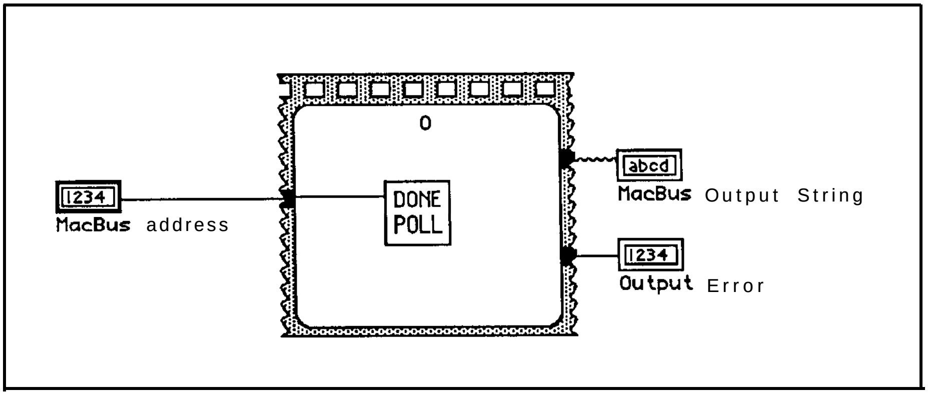


Figure F-15 - String From MacBus Panel



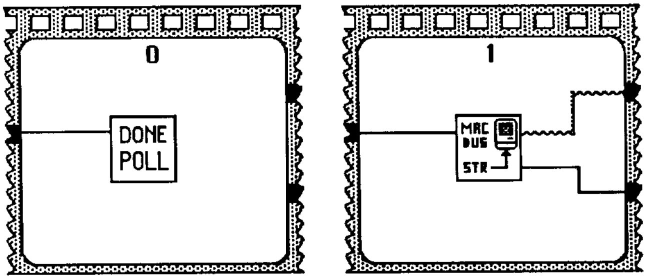


Figure F-16 - String From MacBus Diagram

MacBus String I/O

MacBus String I/O is a consolidation of the two above instruments. It uses the same input and output variables as String to MacBus and String from MacBus and executes the sequence of events as if String to MacBus was executed followed by the execution of String from MacBus. MacBus String I/O provides you with a window to the IBCL interpreter in MacBus and thereby is useful for interactive programming in IBCL. You may enter a string and when the Go icon is clicked, the string is sent to IBCL and any response returned.

Note: IBCL will not return a string unless the following IBCL command string is placed at the beginning of the first string sent to MacBus.

```
find iaemit 'emit !
```

This string instructs IBCL to place its responses in the output buffer that is read by the String From MACBUS instrument. It is only necessary to send this string once after MacBus is powered on or reset. The MacBus string I/O icon, connector pane, and connector pane controls are shown in Figure F-17. The MacBus string I/O panel is shown in Figure F-19 and a diagram of the MacBus string I/O instrument is shown in Figure F-20.

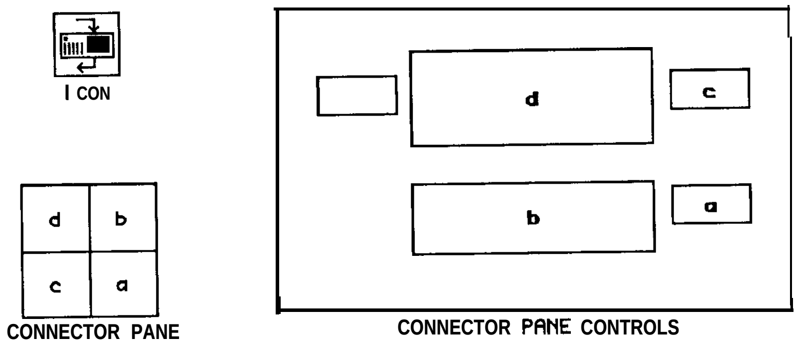


Figure F- 17 - MacBus String I/O Icon, Connector Pane, and Connector Pane Controls

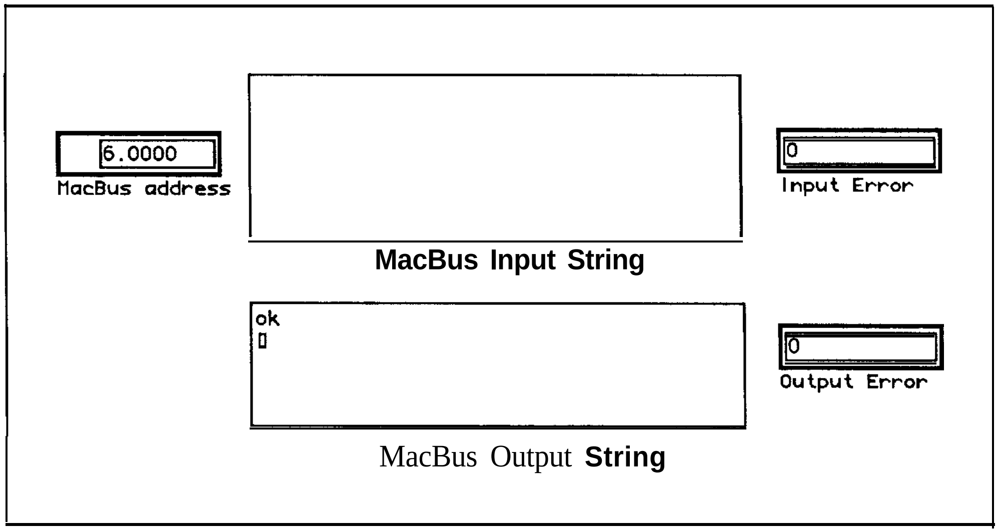


Figure F- 18 - MacBus String I/O Panel

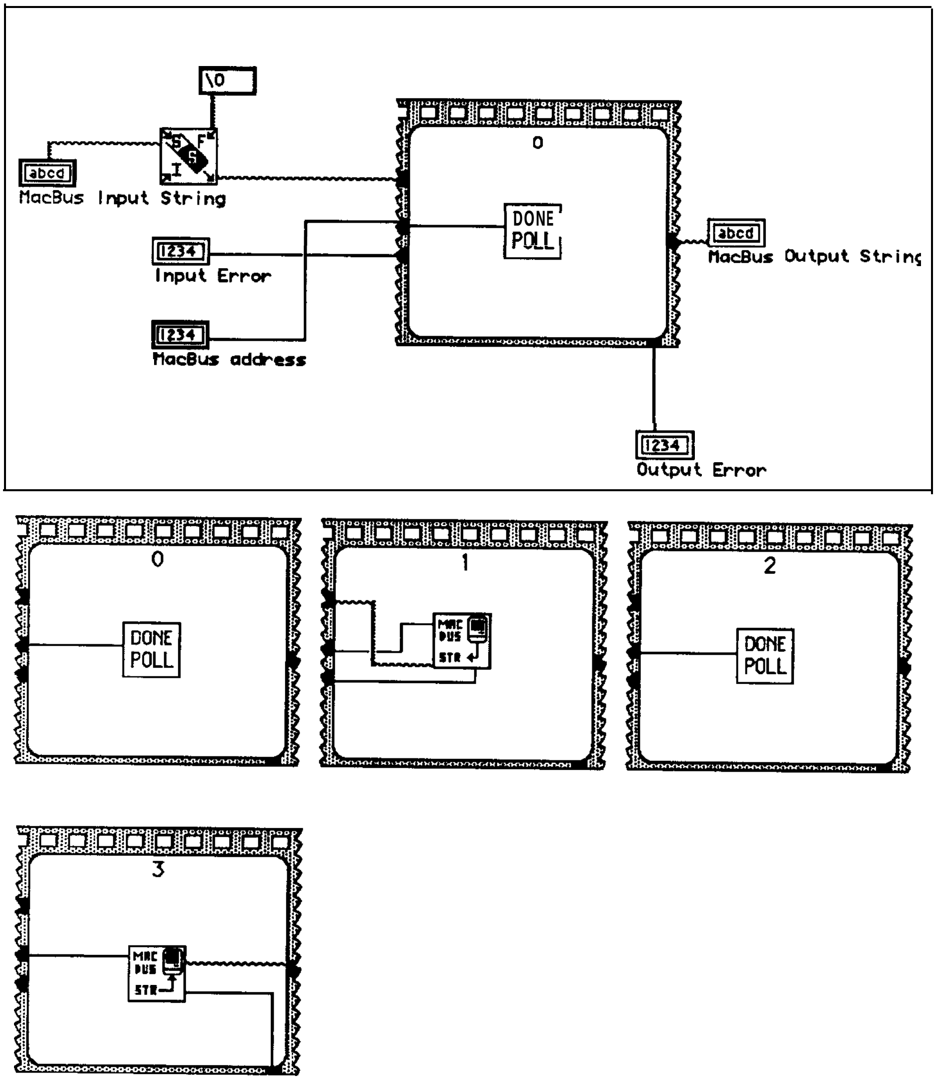


Figure **F-19 -** MacBus String I/O Diagram

MacBus DLM

MacBus DLM accepts the MacBus address, the information to be downloaded, and a binary control as input. If the MacBus address is not supplied (connected), it uses the default MacBus address of 6. The information to be downloaded is handled in **LabVIEW** as if it were a string, however it is treated as binary information in MacBus. This "string" may contain any type of binary information such as the contents of a file, or a previously uploaded buffer. The binary control controls whether or not the buffer is reversed as it is downloaded. Prior to downloading to MacBus, you must allocate a buffer in MacBus to download to. This is done in IBCL by using the **alloc** word or related words. The syntax of **alloc** is:

n **alloc** bufname

where **n** is the size of the buffer to be allocated, and bufname, when subsequently used, will cause the buffer segment (i.e., paragraph number) to be pushed onto the stack. (See IBCL Heap Management Words paragraph in Section Three of this manual). Before using MacBus DLM, a string must be sent to the MacBus that executes the word dlm which instructs IBCL to prepare for a download (See IBCL SCSI Interface Words paragraph in Section Three of this manual). The word dlm requires that the segment, offset and length of the buffer be on the stack. For example:

20 **alloc** MyownBuff MyownBuff 0 20 dlm

allocates a 20 byte buffer named MyownBuff and instructs IBCL to download 20 bytes and store them at the address **MyownBuff:0000.** After this string is sent, the MacBus DLM instrument may be executed.

The MacBus DLM instrument executes the dlm poll instrument, and when completed, executes the Download Memory instrument. The instrument returns error information from Download Memory. The MacBus DLM icon, connector pane, and connector pane controls are shown in Figure F-20. The MacBus DLM panel is shown in Figure F-21 and a diagram of the MacBus DLM instrument is shown in Figure F-22.

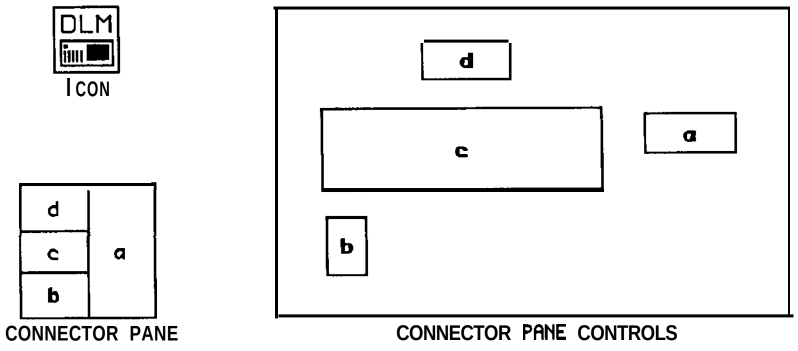


Figure F-20 - MacBus DLM Icon, Connector Pane, and Connector Pane Controls

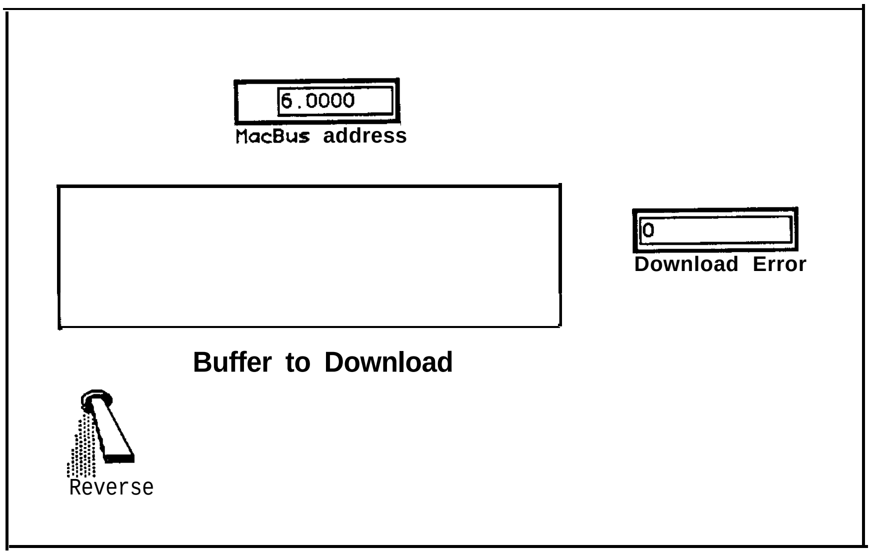
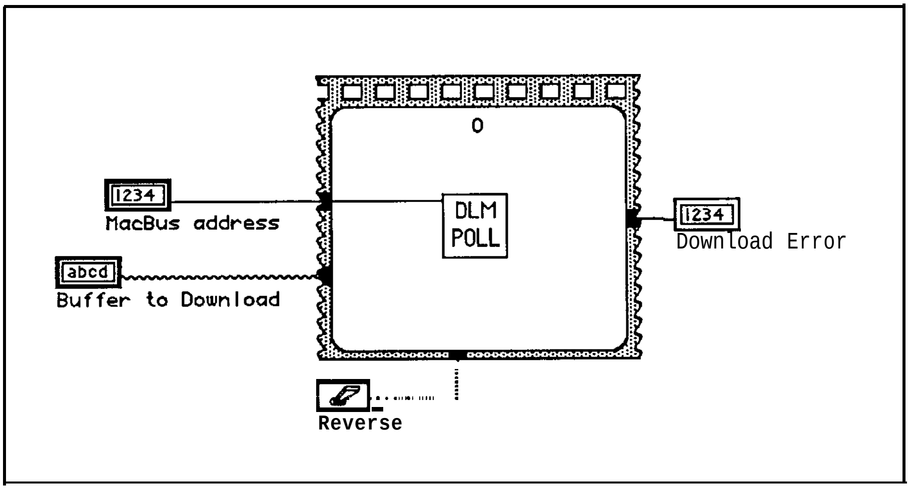


Figure F-21 - MacBus DLM Panel



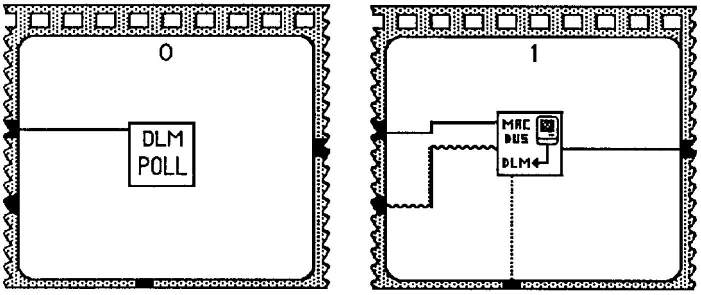


Figure F-22 - MacBus DLM Diagram

MacBus ULM

MacBus ULM accepts the MacBus address, the upload count and a binary control as input. If the MacBus address is not supplied (connected), it uses the default MacBus address of 6. The upload count specifies how may bytes will be uploaded from MacBus Memory. The binary control controls whether or not the buffer is reversed as it is uploaded.

Before using MacBus ULM, a string must be sent to the MacBus that executes the word ulm which instructs IBCL to prepare for an upload (See IBCL SCSI Interface Words paragraph in Section Three of this manual). The word ulm requires that the segment, offset, and length of the memory buffer to be uploaded be on the stack. For example:

```
MyownBuff 0 20 ulm
```

instructs IBCL to upload 20 bytes from MacBus memory beginning at address MyownBuff:0000. After this string is sent, the MacBus ULM instrument may be executed.

The MacBus ULM instrument executes the ulm poll instrument, and when completed, executes the Upload Memory instrument. The instrument returns the uploaded information from MacBus in a string and error information from Upload Memory. The uploaded information is handled in LabVIEW as if it were an ASCII string even though it is actually a sequence of binary bytes. The string may be subsequently parsed and converted to a sequence of numbers by LabVIEW. The MacBus ULM icon, connector pane, and connector pane controls are shown in Figure F-23. The MacBus ULM panel is shown in Figure F-24 and a diagram of the MacBus ULM instrument is shown in Figure F-25.

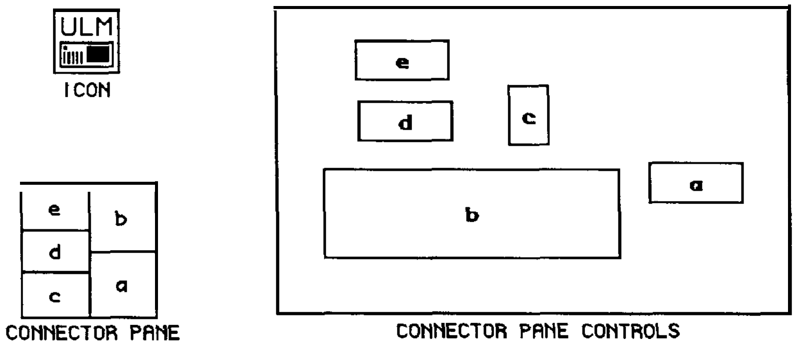


Figure F-23 - MacBus ULM Icon, Connector Pane, and Connector Pane Controls

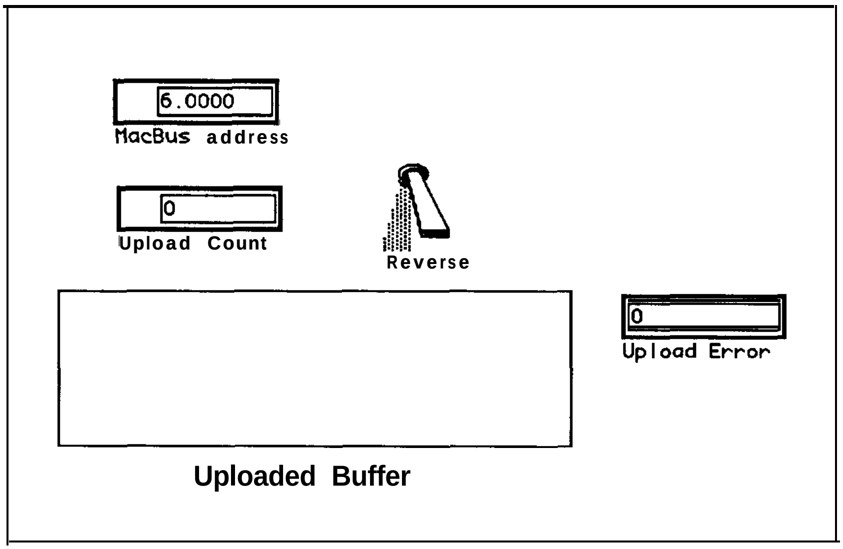
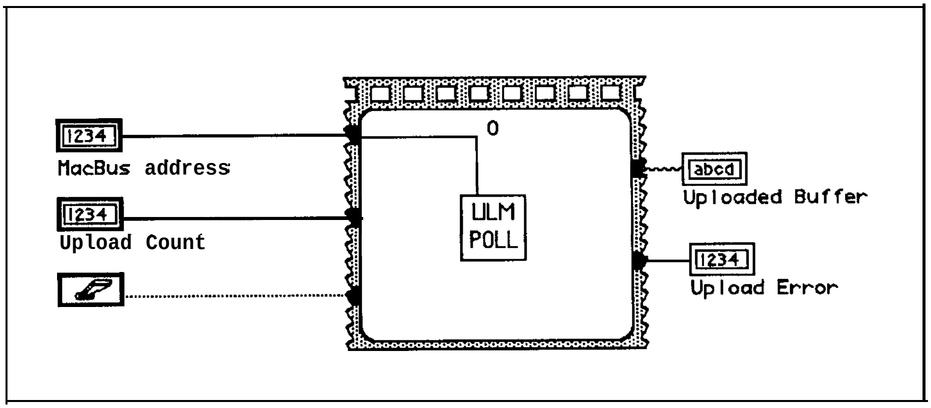


Figure F-24 - MacBus ULM Panel



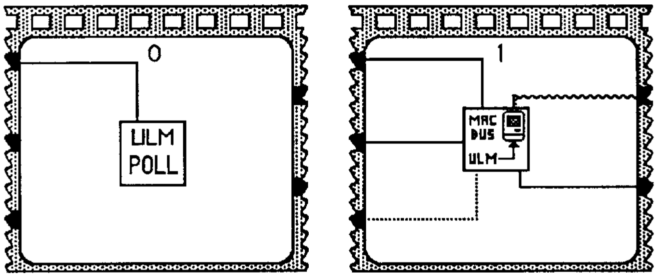


Figure F-25 - MacBus ULM Diagram

Applications of These Example Instruments

The instruments described above have been used in-house at National Instruments to run LabVIEW/MacBus applications. Some of our specific applications are presented below.

MacBus String I/O was frequently used to interactively test IBCL command string applications. The strings were then automatically constructed and sent by LabVIEW using LabVIEW's string processing instruments and the String to MACBUS instrument inside a higher level instrument.

The MacBus DLM instrument was used to download an 8086 machine code binary executable file which had been executed and linked on an IBM PC into MacBus memory. This file contained subroutines and drivers to operate a PC compatible card. The drivers and subroutines had been developed previously for an IBM PC/XT/AT application. All that was added was an assembly language interface to IBCL (See the MacBus Technical Reference Manual). The binary file was copied from an IBM PC onto a Macintosh diskette by one of the PC to Macintosh file transfer packages available on the market. The binary file was opened and read into a string by the LabVIEW file I/O instruments and then passed to the MacBus DLM instrument. Once the machine code was downloaded, the routines were called by executing IBCL words that transferred execution to the beginning of the code's buffer.

The MacBus ULM instrument was used to upload the contents of a buffer of data that had been generated by a card running in MacBus. Once the buffer was uploaded, the string was converted into an array by LabVIEW for subsequent scientific processing in LabVIEW.

The examples and applications presented in this appendix are only a few of the possibilities provided by LabVIEW and MacBus. You should now be able to take these examples and application ideas and develop applications that will suit your needs.

Product User Comment Form

National Instruments encourages you to give us comments on the documentation supplied with its products. This information helps us provide quality products to meet your needs.

Did you find deficiencies? Please comment on the completeness, clarity, and organization.

____

____

____

Did you find errors? If so, please give page number, and a description of the error.

____

____

____

Thank you for your help.

Name__________________________Title ________________

Company Name ____________________________________

Number & Street ___________________________________

City_____________________State & Zip _________________

Phone ( )-____________________

Mail to: Technical Publications
National Instruments
12109 Technology Boulevard
Austin, TX 78727-6204

MacBus UM