COMPREHENSIVE SERVICES

We offer competitive repair and calibration services, as well as easily accessible documentation and free downloadable resources.

SELL YOUR SURPLUS

We buy new, used, decommissioned, and surplus parts from every NI series. We work out the best solution to suit your individual needs.

Sell For Cash Get Credit Receive a Trade-In Deal

OBSOLETE NI HARDWARE IN STOCK & READY TO SHIP

We stock **New, New Surplus, Refurbished,** and **Reconditioned** NI Hardware.

APEX WAVES

Bridging the gap *between the manufacturer and your legacy test system.*

1-800-915-6216

www.apexwaves.com

sales@apexwaves.com

Request a Quote CLICK HERE *GPIB-232CT-A*

GPIB-CT IBCL
Reference Manual

December 1993 Edition

Part Number 320132-01

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (512) 794-5678

Branch Offices:
Australia 03 879 9422, Austria 0662 435986, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521,
Denmark 45 76 26 00, Finland 90 527 2321, France 1 48 65 33 70,
Germany 089 714 50 93, Italy 02 48301892, Japan 03 3788 1921,
Netherlands 01720 45761, Norway 03 846866, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 27 00 20, U.K. 0635 523545

Limited Warranty

The GPIB-232CT, GPIB-422CT, and the GPIB-232CT-A are warranted against defects in materials and workmanship for a period of two years from the date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Important Notice

The material in this manual is subject to change without notice. National Instruments assumes no responsibility for errors which may appear in this manual. National Instruments makes no commitment to update, nor to keep current, the information contained in this document.

Copyright

Trademarks

MicroGPIB™ is a trademark of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

Warning Regarding Medical and Clinical Use of National Instruments Products

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

Illustrations

List of Figures

List of Tables

About This Manual

This manual describes the National Instruments IBCL (Interface Bus Control Language) operating system for the GPIB-232CT, GPIB-422CT, and GPIB-232CT-A interface products. This manual describes the built-in IBCL commands and outlines techniques for adding new ones to the system.

This manual applies to the GPIB-232CT, GPIB-422CT, and GPIB-232CT-A interface products. Rather than mentioning all three products when a reference is made, this manual will use the notation GPIB-CT to indicate all products.

Assumption of Previous Knowledge

IBCL users include OEMs who have custom applications for the GPIB-CT and experienced users who wish to access the full power of the on-board processor.

To use this manual effectively, you should be somewhat familiar with microcomputers, computer devices, and the GPIB-CT default operating system. You should also have an understanding of the IEEE 488 functionality.

Organization of the Manual

The following discussion contains a description of each section of the *GPIB-CT IBCL Reference Manual*.

- Chapter 1, *Getting Started with IBCL*, contains a brief tutorial which demonstrates the operation of the IBCL language.
- Chapter 2, *IBCL Reference*, contains a formal description of the IBCL language.
- Chapter 3, *GPIB Extensions*, describes the IBCL extensions you can use to directly operate and control the GPIB.
- Chapter 4, *Programming Examples*, contains sample applications written in IBCL.

- Chapter 5, *Technical Information*, contains information for improving and customizing performance from the GPIB-CT.
- Appendix A, *Multiline Interface Messages*, contains an ASCII chart, and a list of the corresponding GPIB messages.
- Appendix B, *IBCL Status and Error Messages*, contains a table of the IBCL status and error messages.
- Appendix C, *Creating Permanent IBCL Words in EPROM*, describes the procedure for permanently adding new words and data to the IBCL operating system.
- Appendix D, *Using Extended Memory*, describes the extended memory of the GPIB-CT and gives guidlines for its use with IBCL.
- Appendix E, *Other Useful IBCL Words*, contains IBCL words that are application-specific.
- Appendix F, *Glossary of IBCL Functions*, contains a list of commonly used IBCL words and a description of each.
- Appendix G, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.

Conventions Used in This Manual

italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
monospace	Lowercase text in this font denotes text or characters that are to be literally input from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, variables, filenames, and extensions, and for statements and comments taken from program code.

bold monospace	Bold lowercase text in this font denotes the messages and responses that the computer automatically prints to the screen.
italic monospace	Italic lowercase text in this font denotes that you must supply the appropriate words or values in the place of these items.
<CR>	carriage return
<LF>	line feed

The period character (.) is referred to as the IBCL word "dot" in the programming examples.

A space ( ) appears in the examples wherever you should press the spacebar. It is very important that you notice where spaces are used in the examples, because a space is the separator operation for IBCL. In the programming examples of this manual, characters separated by spaces look like this:

```
0 1 2 3 4
```

The same string of characters that are not separated by spaces look like this:

```
01234
```

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

Related Documentation

For more information on the internal workings of IBCL or for more tutorial-style information, consult one of the Forth language books listed here:

Forth Fundamentals Vol. 1 by C. Kevin McCabe, dilithium Press.
Starting Forth by Leo Brodie, Prentice Hall (Advanced Techniques).
Forth, An Application Approach by David L. Toppen, McGraw-Hill.
Forth Programming by Leo J. Scanlon, Howard W. Sams Publication.

For more information about the IEEE 488, refer to the *IEEE Standard Digital Interface for Programmable Instrumentation*, published by the Institute of Electrical and Electronics Engineers, Incorporated.

For more information about what each bit represents in each I/O register of the HD64180 microprocessor, refer to the *HD64180 8-Bit High Integration CMOS Microprocessor User Manual*, available from Hitachi America, Ltd., Semiconductor and IC Division.

For more information about what each bit represents in each I/O register of the GPIB Controller chip used in the GPIB-CT, refer to the description of the µPD7210 GPIB controller chip in *NEC Microcomputer Products*, available from NEC Electronics, Inc. This description is used for interface products that contain the NAT4882 controller chip as well as interface products that contain the µPD7210 controller chip.

For information about your GPIB-CT hardware, refer to the *GPIB-232CT User Manual* (part number 320114-01) the *GPIB-422CT User Manual* (part number 320115-01), or the *GPIB-232CT-A User Manual* (part number 320504-01).

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix G, *Customer Communication*, at the end of this manual.

Chapter 1
Getting Started with IBCL

This chapter contains a brief tutorial which demonstrates the operation of the IBCL language.

IBCL (Interface Bus Control Language) is a powerful interactive programming language that can be used to program the GPIB-CT. IBCL resides in GPIB-CT memory and can serve as both the native language and the operating system. The GPIB-CT default operating system is a command-interpreted GPIB language that is executed on startup. The *GPIB-232CT User Manual* , the *GPIB-232CT-A User Manual* , and the *GPIB-422CT User Manual* describe the operation of the GPIB-CT default operating system.

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 the similarities between Forth and IBCL.

Using IBCL

Connect a terminal to the GPIB-CT unit. If you do not have a terminal, you can use a terminal-emulation program on your PC. A terminal or a terminal-emulation program gives you immediate on-screen response to your command input. This will allow you to step through the examples provided in this section.

Feel free to experiment as you work through the following tutorial. You cannot harm the GPIB-CT hardware or firmware by experimenting with IBCL. The worst that could happen is that the RAM copy of IBCL could get corrupted or that you could get in an infinite loop. In either case, turn the GPIB-CT power switch off and then back on. When the **READY** LED is lit, you are ready to start again.

Starting IBCL

IBCL is loaded automatically when the GPIB-CT is started up, so no initialization sequence is necessary. The default operating system of the GPIB-CT is the GPIB-CT default operating system. To start IBCL from the GPIB-CT default operating system, enter the command:

IBCL<CR>

You should immediately see an **ok** prompt on your screen signifying that IBCL is ready for input. The IBCL operating system responds with **ok** after a successful operation. Press <CR> a few times to verify that IBCL is responding to input properly. You should see the following lines:

<CR>
ok
<CR>
ok
<CR>
ok

Pushing and Popping Numbers from the Stack

IBCL uses a push-down stack to store the numbers you enter. Enter the following line:

1 2 3 4 . . . .<CR>

Be sure to put a space after every character including the dots (.).

After you enter a <CR>, the line should look like this:

1 2 3 4 . . . . **4 3 2 1**
ok

The period character (**.**), called a dot in this context, is an IBCL operator that returns, or *pops*, the top number from the stack and prints its value. Four dots print the top four numbers on the stack. Numbers are pushed onto the stack in the order in which they are entered and are retrieved in the reverse order.

IBCL reports the status information (either the **ok** message or an error code) from the previous command on a new line for easy program processing.

Adding Numbers on the Stack

To add two numbers, first enter the numbers you want to add followed by the operator:

For example, to add the numbers 9 and 5, enter this line:

```
9 5 + .<CR>
```

The answer **E** is displayed. E is the hexadecimal equivalent of the decimal value 14. Hexadecimal is the default base of IBCL.

To change the base to decimal, enter this line:

```
decimal<CR>
```

Enter the following line with no space between the number 5 and the plus operator (+):

```
9 5+ .<CR>
```

IBCL responds with this message:

```
5+? MSG # 0
```

MSG # 0 is the unrecognized word error. IBCL operates in terms of words, where a *word* is an unbroken string of any sequence of characters separated by a space ( ), a carriage return (<CR>), or a linefeed (<LF>). Because there is no space entered between the number 5 and the plus sign (+), IBCL interprets 5+ as one word. Notice that because there was an error, IBCL did not return the message **ok**.

If you type dot (.), IBCL responds with the following message:

```
. 46
.? MSG # 1
```

MSG # 1 is the empty stack error. Empty stack means that there are no numbers on the stack. There is nothing on the stack because IBCL clears the stack after an error occurs. Notice that **46** was printed before the error message. This is because IBCL does not detect errors until after execution of a command. In this case, IBCL popped a number off the stack and printed it before it determined that the stack was empty.

Defining New Words

You can easily add new words to the IBCL dictionary. The *dictionary* is the list of IBCL functions.

To define a new word called 3add, which will add the top three numbers on the stack, enter this line:

```
: 3add + + ;<CR>
```

Now, to execute your new word, enter this line:

```
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 (;) comprise the *definition* of the new word.

Now define a new word, 3addshow, which adds the top three numbers on the stack and prints out the result. Enter this line:

```
: 3addshow ." The answer is " 3add . ;<CR>
```

To execute 3addshow, enter this line:

```
3 4 5 3addshow<CR> The answer is 12
ok
```

Notice that 3addshow uses 3add, which is now part of the dictionary. The word ." prints out the characters to the next " exactly as they are entered.

Using Loops and Conditionals

Define a new word, doline, which loops five times and prints out the message line i, where i is the loop count. Enter this line:

```
: doline 5 0 do cr ." line " i . loop ;<CR>
```

To execute doline, enter this line:

```
doline<CR>
line 0
line 1
line 2
line 3
line 4
ok
```

do requires 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 carriage return (<CR>).

Looping and conditional constructs only work within a word definition.

Using Conditionals

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 this line:

```
: tf if ." TRUE " else ." FALSE " endif ;<CR>
```

To execute tf a few times, enter the following lines:

```
1 tf <CR> TRUE
ok
0 tf <CR> FALSE
ok
-1 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 the last command line, 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.

Manipulating the Stack

There are times when the numbers on the stack are not in the order that you want, or when you need to verify a value on the stack without changing its position. There are several stack words that you can use in these cases, such as swap, dup and drop.

Enter this line:

```
1 2 3 4 swap . . . .<CR>
```

The result is **3 4 2 1** because swap reverses the order of the top two numbers on the stack.

Enter this line:

```
2 dup . .<CR>
```

The result is **2 2**, because dup duplicates the top number on the stack.

Another stack word, drop, drops, or pops, the top number from the stack.

Looping

Using the IBCL words you have already learned, you can explore 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>
```

To execute your program, enter this line:

```
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 returns a TRUE/FALSE value indicating whether the top number on the stack is equal to four. eq4 duplicates the top number on the stack and compares it to four. The number must be duplicated, because the top two numbers are popped off the stack when the comparison to four is made. After the begin ... until, the number 4 was still on the stack; the drop removes it.

Forgetting

If you have tried to redefine an existing word definition, you have seen the IBCL warning message:

xxxx **MSG # 4**
ok

IBCL does not replace a previously defined word with a new one. It remembers both, but uses the most recently defined word. To revert to the previous definition of a word, use forget**.** forget removes the requested word and all words that were defined since that word. Enter the following sequence:

: ver ." version 1 " ;<CR>
ok
: ver ." version 2 " ;<CR> **ver MSG # 4**
ok
ver <CR> **version 2**
ok
forget ver<CR>
ok
ver <CR> **version 1**
ok
forget ver<CR>
ok
ver <CR>
ver? MSG # 0

In the previous example, the first line of input defines the new word, ver, to print **version 1**. The second line of input redefines ver to print **version 2**. **MSG # 4** is a warning message stating that a dictionary word has been redefined. The original definition is unchanged, but IBCL uses the most recent definition of a word.

In the third line of input, when the word ver is executed, the most recent definition is displayed. In the fourth line of input, the word forget removes the most recent definition of ver. When ver is executed again in the fifth line of input, the original definition is displayed. In the sixth line of input, forget removed the original definition. After forgetting the original definition, executing ver produces the unrecognized dictionary word message (**MSG # 0**).

Using GPIB Functions

If you do not have access to a GPIB device, you can skip this section. To use a GPIB device, you must first read the manual on the device to see how it responds to GPIB commands.

For this example, assume you have a digitizer at GPIB address 5. To send a device clear command to the digitizer, enter this line:

```
5 clr<CR>
```

All device functions require the address of the device as their first argument. To write data to the digitizer, enter this line:

```
5 " cap13;25" wrt<CR>
```

The double quote character (") creates a buffer with the text cap 13;25 in it. It leaves on the stack the address of the buffer and its count. These arguments are in the correct order for the wrt function.

Exiting IBCL

To exit IBCL, type this word:

```
bye<CR>
```

This returns you to the GPIB-CT default operating system. Notice that any changes that you have made in the IBCL operating system will now be in effect in the GPIB-CT default operating system and vice versa. This sharing of memory resources allows you to switch from one operating system to the other at any time.

Chapter 2
IBCL Reference

This chapter contains a formal description of the IBCL language.

Language Structure

An IBCL program is a list of numbers or one-word commands received over the GPIB-CT serial port. A word is an unbroken string composed of up to 31 characters. The IBCL standard word set includes the following characters:

```
!"#$%&'()*+,-./0123456789:;<=
>?@abcdefghijklmnopqrstuvwxyz[\]
```

IBCL defines and recognizes words composed of any sequence of 8-bit bytes. Space ( ), carriage return (<CR>), and linefeed (<LF>) characters serve as word delimiters. The backspace character, ASCII 8, causes IBCL to back over the last byte entered.

IBCL may use both upper and lower case characters; however, IBCL is case-sensitive. Thus, the input sequence SAMPLE, sample and Sample will be interpreted as three distinct words. Notice that all of the standard IBCL words use lower case characters and must be typed in lower case in order to be recognized.

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 the IBCL words are detailed and specific; they are neither ambiguous nor dependent on context.

IBCL uses postfix notation syntax. In postfix notation, you write the stack numbers and then the operators. Numbers are pushed onto a stack and taken from it. For example, examine this line:

```
7 2 12 3 / * -
```

First 7, and then 2, 12, and 3 are pushed onto the stack. The next character, the slash (/), is an operator which divides 12 by 3. The result, 4, is placed on the top of the stack. Now 7, 2 and 4 comprise the stack. The next operator is *, which multiplies 2 by 4. The result, 8, is placed on the top of the stack, leaving 7 and 8 on the stack. The next operator is -, which subtracts 8 from 7, which leaves -1 on the stack.

The definition of a new IBCL word is composed of a list of previously defined IBCL words or machine code primitives. A machine code primitive is the lowest-level routine, which is written in assembly language. A machine code primitive does not call any other IBCL words.

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.

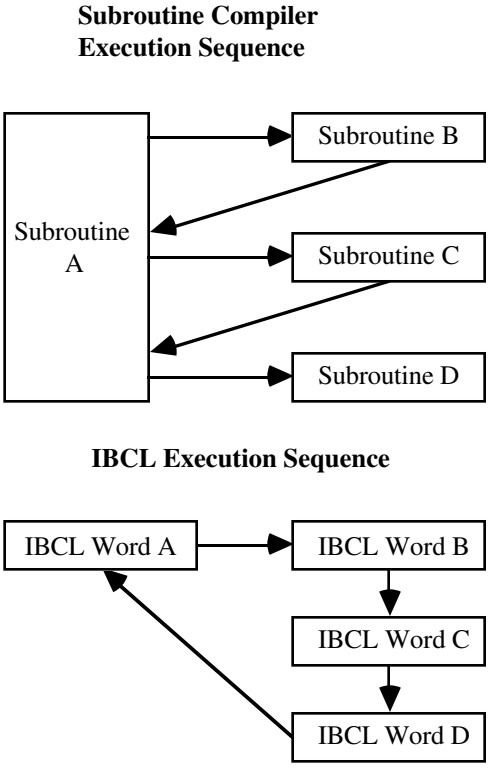


Figure 2-1. IBCL Versus the Subroutine Compiler

Stacks

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 IBCL interpreter uses the return stack to find its way back up through nested sequences of words being executed. It is always called "the return stack."

A stack can be compared to a deck of cards lying face up with each card only partially covering the one below, as in some solitaire games. With IBCL, 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.

Using the card scenario, consider the following examples. A mathematical or logical operator like max (maximum value) would take the top two cards from the stack, place the higher valued one back, and discard the other. 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 keeps the data it is using on the parameter 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 mathematical or logical 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

The parameter stack grows towards lower memory and is under direct user control. The parameter stack pointer occupies the sp register. Any words which refer to the state of the stack refer to the state that existed before the word was executed.

Table 2-1. Parameter Stack Words

Word	**Functionality**
-dup	Duplicate top number on stack if it is non-zero (useful for if constructs)
drop	Drop top number from stack
dup	Duplicate top number on stack
over	Duplicate second from top word on stack
rot	Remove third number from stack, leaving it on top
swap	Remove second number from stack, leaving it on top
2dup	Duplicate top double number on stack

(continues)

Table 2-1. Parameter Stack Words (continued)

Word	Functionality
sp@	Return current address of stack pointer
sp!	Initialize stack pointer to value in s0 (clears stack)
s0	Return the address of the user variable which holds the initial value for the stack pointer. When the stack pointer has this value, the stack is empty.

The return stack is used mainly for system needs and takes care of itself. When a higher-level word is executed, each lower-level word in its definition is executed. 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 at 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.

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.

The return stack grows towards the parameter stack. The return stack pointer is stored in the memory location rp@.

The following words are used to manipulate the return stack state. They should only be used within a word definition and should be used with extreme caution since an unbalanced return stack will crash the system.

Table 2-2. Return Stack Words

Word	Functionality
>r	Transfer top number from data stack to return stack
r	Copy top number from return stack to data stack
r>	Transfer top number from return stack to data stack
rp@	Return current address of return stack pointer

(continues)

Table 2-2. Return Stack Words (continued)

Word	Functionality
rp!	Initialize return stack pointer to value in r0 (clears return stack)
r0	Return the address of the user variable which holds the initial value for the return stack pointer. When the return stack pointer has this value, the return stack is empty.

Numeric Operations

IBCL stores numeric information in consecutive 8-bit byte memory locations and can represent character (8-bit), single precision (16-bit), or double precision (32-bit) data. It is the responsibility of the programmer to insure that the correct numeric operations are used with the proper data types, as IBCL does not differentiate between the different data formats.

Both signed and unsigned single and double precision numbers can be represented, but again, the programmer is responsible for insuring the correct representation of data types. All signed numbers are stored in two's complement form so that arithmetic operations can be handled without special consideration.

A number is interpreted as a double number with the inclusion of a decimal point anywhere within the number. A number is interpreted in its two's complement form if a negative sign directly precedes the number.

All IBCL arithmetic operations deal with integer quantities. Integer arithmetic is fast, requires very little memory and is not subject to round-off error. Although floating point routines can be written in IBCL, the easiest way to represent quantities that contain fractional parts is to scale the number. For example, to represent a value given in dollars and cents as an integer value, multiply the number by 100. This gives a value representing a number of cents. This integer value can then be used by any arithmetic function and the result can be reported back in the normalized format or can be scaled back to represent a fractional value. In a sense, IBCL automatically scales numbers which include decimal points since it converts them to double length integers and reports the position of the decimal place.

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

The ranges for the supported number types are given in Table 2-3.

Table 2-3. Supported Number Types and Ranges

Integer Type	Decimal Range		Hexadecimal Range	
signed single	-32,768	32,767	-8000	7FFF
unsigned single	0	65,535	0	FFFF
signed double	-2147483648	2147483647	-80000000	7FFFFFFF
unsigned double	0	4294967295	0	FFFFFFFF
logical	0 for FALSE; 1 (non-zero) for TRUE			

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 does not even have the correct sign. For example, adding one to 32767 gives -32768.

For division the remainder has the same sign as the dividend and the quotient is rounded toward zero.

Unary Operators

These words alter the number on the top of the stack. Most operate on either signed or unsigned integers. The few exceptions (0< and abs) must obviously deal with signed quantities.

Table 2-4. Unary Operators

Word	Functionality
1+	Add 1 to the number on the top of the stack.
2+	Add 2 to the number on the top of the stack.
0<	Leaves a TRUE flag if the number on the top of the stack is less than zero; otherwise leaves a FALSE flag.
0=	Leaves a TRUE flag if the number on the top of the stack is zero; otherwise leaves a FALSE flag.
abs	Replace the top number on the stack with its absolute value.
s->d	Convert signed single length number on the top of the stack to a signed double word number on the top of the stack.
dabs	Replace the double word number on the top of the stack with its absolute value.

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, and double operators with d.

All input words are removed from the stack and the result becomes the new top element on the stack.

Table 2-5. Signed or Unsigned Operands

Word	Functionality
+	Add the top two numbers on the stack.
-	Subtract the top number on the stack from the second number on the stack.
=	Leaves a TRUE flag if the top number on the stack is equal to the second number on the stack; otherwise leave a FALSE flag.
d+	Add the top two double numbers on the stack.
d+-	Return the double number which was second on the stack with the sign of the product of the double number and the top single number.

Table 2-6. Signed Operands

Word	Functionality
+-	Return the top number on the stack with the sign of the product of the first and second numbers on the stack.
*	Multiply the top two numbers on the stack.
/	Divide the second number on the stack by the top number on the stack leaving the quotient on the top of the stack.
/mod	Like / but the remainder is returned as the second element on the stack with the same sign as the dividend.

(continues)

Table 2-6. Signed Operands (continued)

Word	Functionality
mod	Return the remainder of / with the same sign as the dividend.
*/	Multiplies the first and second number, divides the result by the third number, and leaves the quotient on the stack. The quotient is rounded toward zero. The intermediary result (after n1 * n2) is a double number, resulting in greater precision than n1 n2 * n3 /.
*/mod	Like */ but the remainder is returned as the second element on the stack. The remainder has the same sign as the product of n1 * n2.
<	Leaves a TRUE flag if the first number is greater than the second; otherwise leaves a FALSE flag.
>	Leaves a TRUE flag if the second number is greater than the first; otherwise leaves a FALSE flag.
max	Return the greater of the top two numbers on the stack.
min	Return the lesser of the top two numbers on the stack.

Table 2-7. Mixed Length Signed Operands

Word	Functionality
m*	Multiply the two numbers on the top of the stack and return the signed double integer product.
m/	Divide the double integer by the number on the top of the stack leaving the signed quotient on the top of the stack and the remainder as the second element on the stack. The remainder takes its sign from the dividend.
m/mod	Like m/ but returns a double word unsigned quotient and an unsigned remainder from an unsigned double dividend and an unsigned single divisor.

Table 2-8. Unsigned Operands

Word	Functionality
u*	Multiplies two unsigned numbers and leaves the result as an unsigned double number on the stack.
u<	Leaves a TRUE flag if the first unsigned number is greater than the second unsigned number; otherwise leaves a FALSE flag.

Table 2-9. Logical, Sign Bit Not Significant

Word	Functionality
and	Leaves the bitwise AND of the top two numbers on the stack.
or	Leaves the bitwise inclusive-OR of the top two numbers on the stack.
xor	Leaves the bitwise exclusive-OR of the top two numbers on the stack.

Memory Access

These words allow a single byte, word, or double word to be stored or returned from memory. An entire block of bytes may be cleared or filled with any value or a contiguous block may be moved from one location to another.

Constants, variables and arrays are structures used to reserve memory locations in the IBCL system. They also provide user defined label identification for easy recall.

Load and Store

These words store values into memory or retrieve them from memory using an address on the top of the stack.

The root of these words is an at character (@) for load, and an exclamation point (!) for store. A word with the root @ requires an address on the top of 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. Double word numbers store the most significant portion toward the top, just below the address word or words.

A word with the root @ replaces the 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.

Table 2-10. Load and Store Words

Syntax	Word	Functionality
nnnn	c@	Returns the character from memory location *nnnn*.
nnnn	@	Returns the single number from the memory location *nnnn*.
nnnn	2@	Returns the double number from memory location *nnnn*.
nn.nn	lc@	Returns the character from the long double length memory location *nn.nn*.
nn.nn	l@	Returns the single number from the long double length memory location *nn.nn*.
character nnnn	c!	Stores *character* into memory location byte *nnnn*.
number nnnn	!	Stores *number* at memory location starting at *nnnn*.
double nnnn	2!	Stores *double* number at memory location starting at *nnnn*.
character nn.nn	lc!	Stores *character* into long double length memory location *nn.nn*.
number nn.nn	l!	Stores *number* starting at the double length memory location *nn.nn*.
number nnnn	+!	Resembles ! in use but instead of replacing the single length number located at memory address *nnnn*, *number* is added into it.

Notice that there are four words (lc@, l@, lc! l!) which store and retrieve data from long addresses. These words are only used if your unit has 256K of RAM and you wish to use the extended memory space. For more information, see Appendix D, *Using Extended Memory.*

Fill

These words fill a block of memory with copies of a single byte length number. Nothing is returned on the stack.

Table 2-11. Memory Fill Words

Syntax	Word	Functionality
addr n byte	fill	Fill *n* consecutive memory bytes beginning at *addr* with the *byte*.
addr n	blanks	This behaves like fill, but the byte stored is hex 20 (blank).
addr n	erase	erase also behaves like fill, but the byte stored is 0.

Move

To copy a block of memory to a new, possibly overlapping block, enter the following line:

source-addr dest-addr n cmove

cmove moves a block of memory *n* bytes long beginning at the source address to the block at the destination address. The lowest addressed bytes are moved first.

The two blocks may overlap if the destination is lower in memory than the source. If the two blocks overlap and the destination is higher in memory than 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.

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 the line:

nnnn constant *name-of-new-constant*

constant is the dictionary word you are executing. *name-of-new-constant* is a new dictionary entry which is associated with the constant value.

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 executed immediately within a colon definition by using the square bracket pair. 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. An unsigned constant may range from 0 through 65535 decimal.

For example, examine the following lines:

```
5 constant five <CR>
ok
five . <CR> 5
ok
```

A few small integers are used so frequently that they have been implemented as IBCL constants. When the interpreter encounters them, they are located in the dictionary rather than being parsed by number. More importantly, 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. The predefined IBCL constants are 0, 1, 2, and 3.

A variable is defined like a constant, except that whenever the new variable is executed, its parameter field address is pushed onto the stack. Values may then be stored and retrieved from this location.

A value may be defined by entering the following line:

nnnn variable *name-of-new-variable*

variable is the dictionary word you are executing. *name-of-new-variable* is a new dictionary entry which will place the value *nnnn* on the top of the stack.

A signed variable may range from -32768 through 32767 decimal. An unsigned variable may range from 0 through 65535 decimal.

For example, examine the following lines:

```
1 variable jellybeans <CR>
ok
jellybeans @ . <CR> 1
ok
3 jellybeans +! <CR>
ok
jellybeans @ . <CR> 4
ok
```

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 address 22a hex. If multiple copies of this array are needed, the user must create another array, copy the old user array to it, and place the appropriate address in user-base for each task.

The user-base is stored at memory location 226 hex. To use your new array of user variables, you must put the address of the new array into the user-base by entering the following line:

address-of-array 226 !

After you enter this line, IBCL uses your array of user variables. If you want to restore the use of the system array, you must enter 22a hex for *address-of-array*, or turn off the GPIB-CT. No IBCL word resets the array for you.

The user variables at system initialization are listed in Table 2-12.

Table 2-12. User Variables at Initialization

Location	Variable	Location	Variable
22a	reserved	24a	context
22c	reserved	24c	current
22e	reserved	24e	state
230	S0	250	base
232	R0	252	dpl
234	tib	254	reserved
236	width	256	csp
238	warning	258	reserved
23a	fence	25a	hld
23c	dp	25c	unused
23e	voc-link	25e	unused
240	unused	260	unused
242	in	262	unused
244	out	264	unused
246	reserved	266	unused
248	reserved	268	unused

If more user variables are required, you can create a new user variable by typing this line:

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 starts at address 22a hex and is 40 hex bytes long.

Arrays can be created by first defining an array name by using variable, then reserving extra storage space by adjusting the dictionary pointer by using allot. allot takes a number off the top of the stack and reserves that number of bytes in the dictionary space. For example, to create an array many_items with 1000 bytes of storage, enter the following line:

0 variable many_items FFE allot

FFE is used during the allot since two bytes were already reserved by variable. The first element in the array will be 0. Any array element can be designated by its relative location within the array structure.

Input/Output

IBCL has provisions to send and receive both numeric and character data as well as binary arrays of data to and from the serial port. Numeric values will be converted to the corresponding base that is in effect.

IBCL Input

IBCL provides several words that receive information from the serial port. These words may be placed in one of two categories–ASCII-type input words and binary-type input words. This discussion documents IBCL's collection of input words.

ASCII-Type Input

?terminal is an IBCL input word which returns a TRUE flag if there is a character received by the serial port. This is useful to break from a routine.

All IBCL input routines use the core word key which waits for the next character to be received from the serial port and then returns its value on the stack.

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 more input from the serial port. For example, the following IBCL fragment will create a buffer and fill it with ASCII data from the serial port:

```
0 variable string-buffer 3E allot
string-buffer 40 expect
```

The IBCL phrase 0 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 execution of string-buffer on the second line leaves the buffer address on the stack, execution of 40 leaves the count of desired bytes on the stack, and execution of expect waits until either a <CR> or hex 40 bytes are received from the serial port. These bytes are placed in 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.

Binary-Type Input

The IBCL word dlm, for down load memory, allows the host serial device to transmit large arrays of binary data directly to GPIB-CT memory. This word expects a count on top of the stack and a buffer address under that. The IBCL word dlm causes the GPIB-CT to wait for the serial device to send the specified number of bytes over the serial bus and places the data at the specified address. Unlike the other IBCL input words, dlm does not echo the received characters back to the serial port. The following BASIC example illustrates operation of this word:

```
OPEN "COM1:9600,N,8,1" AS #1
OPEN "SENDFILE" FOR INPUT AS #2
PRINT #1,"0 variable buffer FFE allot"
PRINT #1, "buffer 1000 dlm"
FOR COUNT = 1 + 0 &H1000
BYTE$ = INPUT$ (1, #2);
PRINT #1, BYTE$;
NEXT
```

IBCL Output

IBCL provides several words that send information out the serial port. 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 one or more times. This word outputs a single ASCII character to the serial port. emit also increments the value stored in the user variable out which is used as on offset pointer to the last character output.

The system constant c/1, an abbreviation for characters/line, determines the maximum number of ASCII characters per display line (default 64 decimal).

The remainder of this section describes the ASCII-type output 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. cr will emit a carriage return followed by a line feed. 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, no action takes place and the address and count are removed from the stack.

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. 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.

Numeric-Based Words. The representation of a number depends on the base being used. For example, the number of states in the United States is 50 if the base is decimal, but if the base is hexadecimal, there are 32 states. The actual number of states is the same, but the representation is different. A jigsaw puzzle of the United States could be divided into five piles of ten states each with none left over (50 in decimal), or it could be divided into three piles of sixteen states each with two left over (32 in hexadecimal).

In IBCL, 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 hex stores sixteen in base. Octal could be set by entering the following line:

```
8 base !
```

The words in Table 2-13 output a number from the stack as a character string. The top stack word contains a field width for some of them. The individual digits are output by emit.

Table 2-13. Numeric Output Words

Syntax	Word	Functionality
number	.	Display *number* with a single trailing blank and, if required, a leading negative sign.
double	d.	Like . except for double word length number, *double*. The high-order word is on top of the stack with the low-order word under it.
number	u.	Like . but *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 too 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, *double*.
source-address	?	Print the number stored at *source-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
```

In the previous example, decimal changes the base to decimal and -12345 places -12345 on the stack. dup places two copies of -12345 on the stack. s->d sign extends the top copy to double length. dabs takes the absolute value of the double number. <# initializes for output conversion. The next # places the lowest order digit (5) in the buffer. The next # places the second lowest order digit in the buffer. 46 is the ASCII code for a decimal point.

hold places the ASCII character represented by the top value on the stack into the buffer. #s places the remaining digits into the buffer. rot rotates the original signed number to the top of the stack. sign places the sign of the top number on the stack into the buffer. #> terminates the output conversion and leaves the buffer address below number string length on stack. type types the number -123.45.

The <# ... #> construct converts an unsigned double length number to a string. The string is built rightmost character first and grows downward from the buffer address returned by pad. pad points to a text buffer which serves as a scratchpad area where output strings may be constructed. 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.

#s 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, and so on.

The following lines are examples of strings containing such characters:

$1,234,567.89

4-15-89

4/15/89

2:37:15

Table 2-14 lists the ASCII codes in decimal of some useful ASCII characters.

Table 2-14. ASCII Characters

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

If a sign is required, the IBCL word sign can be used as long as a number with the correct sign is available on the stack. 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 all of the digits are converted, and placed in the number string's first character position. The sign could just as easily be added to the string before any digits are converted, thus placing it at the end of the number string as required by some financial formats (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 the user variable 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.

Binary-Type Output

IBCL's binary output word is ulm, for up load memory. This word expects a count on top of the stack and a buffer address just below that. As soon as the serial device requests an ulm, the GPIB-CT sends the specified number of bytes over the serial port, starting at the specified address.

Proper handling of binary output involves cooperative action by the GPIB-CT and the serial device, as the following example shows.

BASIC Program Example:

```
OPEN "COM1: 9600, N, 8, 1" AS #1
OPEN "RECVFILE" FOR OUTPUT AS #2
PRINT #1, "0 variable data_buffer ffe allot"
PRINT #1, "data_buffer 1000 fill_up"
PRINT #1, "data_buffer 1000 ulm"
FOR COUNT = 1 to &H1000 ulm
    BYTE$ = INPUT$ ( 1, #1);
    PRINT # 2, BYTE$;
NEXT
```

ulm is most useful when you want to program a custom EPROM with user-defined words and/or an autoboot routine. For more information, see Appendix C, *Creating Permanent IBCL Words in EPROM.*

Defining New Words

This section describes 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.

IBCL can define several kinds of words, and can even define words that define new types of words. At the simplest level, it can provide direct language support for almost any data type or structure imaginable.

For example, the dot product of order n vectors can easily be reduced to this line:

```
a-vector b-vector dot
```

This is both simpler and more efficient than BASIC, which requires the following code:

```
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 calls and parentheses that have nothing to do with your algorithm or 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 memory word initialized to point to the next available dictionary location. You can then place machine language opcodes directly into this and later dictionary space by using c,. This allows you to write your own machine language primitives for speed-sensitive applications. create 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 31 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, but have some important differences.

The syntax is not cluttered with parentheses and parameter lists. This enables IBCL words to be used more nearly like the words in a human language. IBCL syntax is 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 or time consuming compile and link sequence is required.

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

```
: 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.

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.

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 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 a ;s at its end and setting execution mode. ;s 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 word's name is entered: (37 is ASCII code for %)

decimal <CR>
ok
: fifteen-percent 15 . 37 emit ; <CR>
ok
fifteen-percent <CR> **15%**
ok
hex<CR>
ok
fifteen-percent <CR> **F%**
ok

Numbers are interpreted using the current base. In the example, the previous base was discarded in favor of hexadecimal. 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.

Notice that the following definition would not have changed the base until the definition executed:

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

The 15 and 37 would be interpreted according to the previous base, and typing fifteen-percent would always change the base to decimal.

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 is immediate. Immediate words execute 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, use immediate after defining the new word:

: *name definition* ; immediate

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

Another immediate word often used in definitions is the apostrophe character ('), which is often called a "tick" in this context. 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:

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

Note: memory is an IBCL variable.

The ." immediate word is used to include a message in a definition. If ." 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 parameter field address of another word could be defined as follows:

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

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, the opening square bracket ([) which switches the user variable state from compile to execute mode, and the closing square bracket (]), which 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. Refer to the example on the previous page that begins with : ?var for an example of the correct usage of [and]. A similar word, dliteral, is available for compiling double length values from the stack into the definition.

The following lines are examples of equivalent ways to define a word returning the address of the fifth line of a block given its base address on the stack. Notice that the first is inefficient, since the operations are performed every time the word is executed:

```
: line-5    64 4 * + ;

64 4 *     : line-5  literal + ;

: line-5  [ 64 4 * ] 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.

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 do not want to execute even when the word executes. Instead we want the word to be copied to the definition being created.

compile is used within immediate words to allow the word following compile to be compiled into the dictionary entries of other words that contain the immediate word. Because compile takes the next word from the definition list, the word following compile should never be immediate.

Table 2-15 compares the behavior of an example word, called a-word, defined as non-immediate or immediate.

Table 2-15. Comparison of Non-Immediate and Immediate Characteristics

	Non-Immediate	**Immediate**
a-word executed	executed	
: q [a-word] ;	executed	executed *
: q a-word ;	compiled	executed
: q [compile] a-word ; compiled *	compiled	
: q compile a-word ; immediate	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 parentheses. The opening parenthesis (() must be preceded and followed by a space to be interpreted as an IBCL word. The terminating parenthesis ()) is a delimiter and needs no preceding space. For example:

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

Dictionary

IBCL recalls 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.

The actual definition of an IBCL word consists of four parts–the name field, the link field, the code field, and the parameter field. The name field contains the ASCII codes of characters making up the word's name, preceded by a length byte which specifies the number of characters in the name and certain attributes of the definition.

The link field immediately follows the name field. The link field holds a pointer to the name field of the next most recent word in the same vocabulary. These two fields allow for an easy comparison of input words to dictionary entries by using a linked list technique.

The code field contains an address pointer to the word's execution procedure, which is executable machine code. The parameter field immediately follows the code field. The purpose of the parameter field varies from word to word. For example, the code field of a constant holds a pointer to an execution procedure that causes a single precision constant value to be copied from the constant's parameter field to the stack whenever the name of that constant is entered. Likewise, the code field of a variable contains a pointer to an execution procedure which causes the address of a variable's parameter field (rather than the single precision value stored there) to be placed on the stack when the variable name is executed.

The parameter field of a colon-defined word contains one or more address pointers designating the code field of a component word.

Vocabularies

IBCL allows separate vocabularies that 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.

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 vocabulary, 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.

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.

A new vocabulary may be created by typing this command:

vocabulary *new-vocabulary-name* immediate

where the term *new-vocabulary-name* would be replaced by the name you want to give the new vocabulary. For example:

vocabulary assembler immediate

will create a new vocabulary titled assembler.

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

assembler

At this point, no words will be found in the assembler vocabulary, but the user variable context will contain a pointer to the assembler vocabulary 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 this line:

definitions

Now the user variable current points to the assembler vocabulary instead of the IBCL vocabulary. current governs which vocabulary receives new definitions. If you want to enter new definitions in the vocabulary my-words, but limit interpreter searches to the IBCL vocabulary, type this line:

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

This would have reset context to point to assembler.

Note: 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 this line:

forget *word-to-forget-through*

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

You can move fence by entering this line:

new-fence-limit fence !

This raises the fence beyond which forgetting is not allowed, and prevents accidental forgetting of newly-defined function words.

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. They also eliminate the need for any program position labels such as BASIC's line numbers.

Words that control the flow of program execution are used only within colon definitions. 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 0branch or an unconditional branch. The 0branch 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 0branch. 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.

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):

```
0 variable high-score
: .score dup 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 is *nnnn* high score is *nnnnn***.

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 0branch 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 an if 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 0branch. 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 conditional execution words in that they compile branches and leave addresses on the stack. As described in the previous discussion, 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-again, begin-until, and begin-while-repeat.

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

In the conditional loop, begin *phrase* until, until functions 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.

In 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 is false.

There are two types of do loops–do-loop and do-+loop.

In the loop *limit start* do *phrase* loop, the 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.

The *limit start* do *phrase number* +loop 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.

In addition to the loop control words just mentioned, there are a few words that are designed to help loop processing:

- i is used within a loop to place the current loop index onto the top of the stack. For example, the following definition:

  ```
  : display 10 0 do i c@ u. loop ;
  ```

  outputs to the serial port the bytes in memory locations 0 to 9.

- leave is used within a loop to set the index equal to the limit, thus causing an exit of the loop after the current loop finishes. This is useful if a certain condition became TRUE during execution of a loop.

Chapter 3
GPIB Extensions

This chapter describes the IBCL extensions you can use to directly operate and control the GPIB. These functions are in alphabetical order and are formatted so that you can easily reference them.

The following discussion of the GPIB-related extensions contains references to the constants `ibcnt`, `iberr`, and `ibsta`. These constants refer to memory locations within the IBCL operating system which contain information pertaining to GPIB actions.

`ibcnt` stores the number of bytes transferred from or received by the GPIB-CT during `brd`, `rd`, `bwrt`, `wrt`, or `cmd`. `iberr` stores flags to indicate certain error conditions that may have occurred. `ibsta` stores information about the current state of the GPIB system to which the GPIB-CT is attached. For more information about what each bit represents in `ibsta` and what each value represents in `iberr`, refer to the `stat` function description later in this section.

To aid in the use of these variables, you can define IBCL constants in your dictionary by typing the following lines:

```
4 constant ibcnt
2 constant iberr
0 constant ibsta
```

When you need to use the information in these locations, type the following lines:

```
ibcnt @
iberr c@
ibsta @
```

This puts the information stored at these locations onto the stack. Notice that `iberr` is an 8-bit value, and `ibcnt` and `ibsta` are 16-bit values.

Another way to obtain the information stored in `ibsta` and `ibcnt` is to use the IBCL command `stat`, which puts these two values on the stack for you.

brd

brd: Read Data from GPIB

Syntax: `buf cnt brd`

Remarks: `buf` is the address of the buffer to use.

`cnt` specifies the number of bytes to read from the GPIB.

`brd` attempts to read `cnt` bytes of data from a GPIB device that is assumed to already be properly initialized and addressed.

If the GPIB-CT GPIB port is CIC, `cmd` must be called prior to `brd` to address a device to talk and the GPIB-CT GPIB port to listen. If the GPIB-CT GPIB port is not CIC, the device on the GPIB that is the CIC must perform the addressing.

If the GPIB-CT GPIB port is Active Controller, the GPIB-CT GPIB port is first placed in Standby Controller state with ATN off and remains there after the read operation is completed.

An EADR error results if the GPIB-CT GPIB port is CIC but has not been addressed to listen with `cmd`. An EABO error results if the GPIB-CT 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` 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

When `brd` returns, `ibcnt` contains the actual number of data bytes read from the device. A short count can occur on any of the previous events but the first.

See Also: `cmd`, `eos`.

Example:

1. To read 56 bytes of data from a device at talk address 0x4C (ASCII L) and then unaddress it (the GPIB-CT GPIB port is at listen address 0x20 or ASCII blank):

```
" ?L " cmd          ( address talker and listener)
buf 56 brd          ( read data)
" _?" cmd           ( unaddress talker)
                    ( and listener)
```

bwrt

bwrt: Write Data to GPIB

Syntax: `buf cnt bwrt`

Remarks: `buf` is the address of the buffer to use.

`cnt` specifies the number of bytes to be sent over the GPIB.

`bwrt` attempts to write `cnt` bytes of data to a GPIB device that is assumed to already be properly initialized and addressed.

If the GPIB-CT GPIB port is CIC, `cmd` must be called prior to `bwrt` to address the device to listen and the GPIB-CT GPIB port to talk. Otherwise, the device on the GPIB that is the CIC must perform the addressing.

If the GPIB-CT GPIB port is Active Controller, the GPIB-CT GPIB port is first placed in Standby Controller state with ATN off and remains there after the write operation is completed.

An EADR error results if the GPIB-CT GPIB port is CIC but has not been addressed to talk with `cmd`. An EABO error results if the GPIB-CT 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
- When no listeners are detected after the operation begins (the GPIB-CT reports ENOL in this case)
- Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is CIC

When `bwrt` returns, `ibcnt` contains the actual number of data bytes written. A short count can occur on any of the previous events but the first.

See Also: `cmd`, `eos`.

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 GPIB-CT GPIB port is 0x40 or ASCII @):

```
" ?@5" cmd              ( UNL MTA MLA )
" F3R1X5P2G0" bwrt      ( send instruction bytes )
" _?" cmd               ( unaddress talker )
                        ( and listener )
```

   NOTE: The double quote (") places text in memory up to the closing quote or decimal 65 characters. " also leaves the address and string length on the stack and is thus ideal for use with `bwrt`. For instance, `" abc"` leaves the address of the string and a count of 3 on the stack.

cac

cac: Become Active Controller

Syntax: `v cac`

Remarks: If `v` is non-zero, the GPIB-CT takes control synchronously with respect to data transfer operations; otherwise, the GPIB-CT takes control immediately (and possibly asynchronously).

It is generally not necessary to use the `cac` word. Words such as `cmd` and `rpp`, which require that the GPIB-CT take control, do so automatically.

To take control synchronously, the GPIB-CT 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 a `rd` or `wrt` 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 the GPIB-CT is not CIC.

See Also: `gts`, `sic`.

Examples:

1. To take control immediately without regard to any data handshake in progress:

```
0 cac
```

2. To take control synchronously and assert ATN:

```
1 cac
```

caddr

`caddr`: Change GPIB Address of GPIB-CT

Syntax: `addr caddr`

Remarks: `addr` is a valid GPIB address.

`caddr` is used to change the GPIB address of the GPIB-CT. The new address will remain in effect until `caddr` is called again or the GPIB-CT is turned off.

The power-on default is zero with secondary addressing disabled.

Examples:

1. To change the GPIB address of the GPIB-CT to 5 with secondary addressing disabled:

```
5 caddr
```

2. To change the GPIB address of the GPIB-CT to 7 with a secondary address of 8:

```
7 8 hex 100 * + 8000 + caddr
```

clr

clr: Send Selected Device Clear (SDC)

Syntax: `addr clr`

Remarks: `addr` is a valid GPIB address.

`clr` sends the selected device clear (SDC) message. 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)

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

Example:

1. To clear the device at GPIB address 5:

```
5 clr
```

cmd

cmd: Send Command Message to GPIB

Syntax: `buf cnt cmd`

Remarks: `buf` is the 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.

`cmd` 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.

`cmd` is *not* used to transmit programming instructions to devices; programming instructions and other device dependent information are transmitted with `brd`, `bwrt`, `rd`, and `wrt`.

`cmd` 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 GPIB-CT)

When `cmd` returns, `ibcnt` contains the actual number of command bytes sent.

An ECIC error results if the GPIB-CT GPIB port is not CIC. If the GPIB-CT GPIB port is not Active Controller, it asserts ATN prior to sending the command bytes. The GPIB-CT 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" cmd
   ```

   NOTE: The double quote (") places text in memory up to the closing quote or decimal 65 characters. " also leaves the address and string length on the stack and is thus ideal for use with `cmd`. For instance, `" abc"` leaves the address of the string and a count of 3 on the stack.

2. Same as Example 1 except the Listener has a secondary address of 0x6E (ASCII n):

   ```
   " ?F1n" cmd
   ```

eos

eos: Change/Disable GPIB EOS Termination Mode

Syntax: `val eos`

Remarks: `val` specifies the eos character and the data transfer termination method according to Table 3-1.

The assignment made by this function remains in effect until `eos` is called again or the GPIB-CT is turned off. By default, no `eos` modes are enabled.

Table 3-1. Data Transfer Termination Method

Method	Value of val *	
	Byte 1	**Byte 0**
A. Terminate read when eos is detected (`brd` and `rd`)	REOS 04 hex	eos
B. Send END when eos is written (`bwrt` and `wrt`)	XEOS 08 hex	eos
C. Compare all 8 bits of eos byte rather than low 7 bits (all reads and writes)	BIN 10 hex	eos

* Byte 0 is the least significant byte.

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. If Methods B and C are chosen, a full 8-bit comparison is used. The eos character should always be the last byte sent.

See Also: `eot`.

Examples:

1. To send END when the linefeed character is written for operations involving device `dvm`:

```
80A eos
31 buf c!           ( data bytes to be written )
32 buf 1+ c!        ( are placed in buffer with )
33 buf 2+ c!        ( EOS character as last byte )
0A buf 3 + c!
dvm buf 4 wrt
```

2. To program device `dev1` to terminate a read on detection of the linefeed character that is expected to be received within 512 bytes:

```
40A eos
dev1 buf 512 rd
( The END bit in status word is set if the )
( read terminated on the eos character with )
( the actual number of bytes received )
( contained in ibcnt.)
```

3. To disable EOS termination:

```
0 eos
```


eot

eot: End of Transfer Mode

Syntax: `v eot`

Remarks: If `v` is non-zero, the GPIB-CT automatically sends the END message with the last byte of each `wrt`. If `v` is zero, END is not sent. The power-on default is 1.

`eot` is used to change how the GPIB-CT terminates GPIB writes. Using `eot`, you tell the GPIB-CT to automatically send or not send the GPIB END message with the last byte written to the GPIB.

The assignment made by `eot` remains in effect until `eot` is called again or the GPIB-CT is turned off.

The GPIB-CT sends the END message by asserting the GPIB EOI signal during the last byte of a data transfer.

Examples:

1. To disable END termination:

```
0 eot
```

2. To enable END termination:

```
1 eot
```

gts

gts: Go from Active Controller to Standby

Syntax: `v gts`

Remarks: `v` is the type of go-to-standby.

`gts` causes the GPIB-CT to go to the Controller Standby state and to unassert the ATN signal if it is the Active Controller. `gts` permits GPIB devices to transfer data without the GPIB-CT being a party to the transfer.

It is generally not necessary to use `gts`. Functions such as `rd` and `wrt`, which require that the GPIB-CT go to standby, do so automatically.

If `v` is non-zero, GPIB-CT shadows data transfer handshakes as an Acceptor and when the END message is detected, GPIB-CT 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, the GPIB-CT 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 the GPIB-CT to take control synchronously on a subsequent operation such as `cmd` or `rpp`.

The ECIC error results if the GPIB-CT is not CIC.

See Also: `cac`, `cmd`, `wait`.

Example:

1. To turn the ATN line off:

```
0 gts
```


ist

`ist:` Set or Clear Individual STatus Bit (IST)

Syntax: `v ist`

Remarks: `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. The power-on default is that the individual status bit is cleared.

`ist` is used when the GPIB-CT 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. The GPIB-CT, for example, can be assigned to drive the DIO3 data line true if IST=1 and false if IST=0; conversely, it can be assigned to drive DIO3 true if IST=0 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. The GPIB-CT receives this message via a command from the Active Controller. Once the PPE message is executed, `ist` changes the sense at which the line is driven during the parallel poll, and in this fashion the GPIB-CT can convey a 1-bit, device dependent message to the Controller.

See Also: `ppc`, `rpp`.

Examples:

1. To set the individual status bit:

```
1 ist
```

2. To clear the individual status bit:

```
0 ist
```

loc

loc: Go to Local Mode

Syntax: `addr loc`

Remarks: `addr` is a valid GPIB address.

`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)
- Unli sten (UNL)

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

Examples:

1. To return device plotter to local state:

```
plotter loc
```

onl

onl: Place Device Online or Offline

Syntax: `v onl`

Remarks: `v` is a true/false value indicating online/offline.

`onl` is used to disable communications between the GPIB-CT and the GPIB.

NOTE: Unlike the `onl` in the GPIB-CT default operating system, IBCL `onl` does not restore the GPIB-CT operating parameters. To reset your GPIB-CT to its default characteristics, you must exit to the GPIB-CT default operating system and call the `onl` function. Refer to your GPIB-CT User Manual for a description of the GPIB-CT `onl` function and the default characteristics.

If `v` is non-zero, the GPIB-CT places itself online; if zero, the GPIB-CT places itself offline. By default, the GPIB-CT starts up online, is in the Idle Controller state, and configures itself to be the System Controller.

Placing the GPIB-CT offline may be thought of as disconnecting its GPIB cable from the other GPIB devices.

Placing the GPIB-CT online allows the GPIB-CT to communicate over the GPIB.

Examples:

1. To put the GPIB-CT online:

```
1 onl
```

2. To put the GPIB-CT offline to prevent it from communicating on the GPIB:

```
0 onl
```

pct

pct: Pass Control

Syntax: `addr pct`

Remarks: `addr` is a valid GPIB address.

`pct` passes CIC authority to the specified device. The GPIB-CT 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)

If `pct` is called and the GPIB-CT is not CIC, the ECIC error is posted.

Example:

1. To pass control to the device at GPIB address 3:

```
3 pct
```

ppc

ppc: Parallel Poll Configure

Syntax: `addr v ppc`

Remarks: `addr` is a valid GPIB address.

`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 DIO8) 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=1. And if PPE=0x68, DIO1 is driven true if IST=1 and false if IST=0. Any PPD message cancels the PPE message in effect.

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

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 listed in Appendix A.

See Also: `rpp`, `ist`.

Example:

1. To configure device `dvm` to respond to a parallel poll by sending data line DIO3 true if IST=0:

```
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 70 ppc
```

rd

rd: Read Data from GPIB

Syntax: `addr buf cnt rd`

Remarks: `addr` is a valid GPIB address.

`buf` is the address of the buffer to use (`buf` might have been created using `allot`).

`cnt` specifies the number of bytes to read from the GPIB.

`rd` attempts to read `cnt` bytes of data from a GPIB device. The following steps are performed:

1. UNL is sent.
2. The device is addressed to talk and the GPIB-CT GPIB port to listen, if not already addressed to do so.
3. The GPIB-CT reads the data from the device.

An EABO error results if the operation does not complete for whatever reason within the time limits.

`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)
- Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is CIC

When `rd` returns, `ibcnt` contains the actual number of data bytes read from the device. A short count can occur on any of the previous events but the first.

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

Example:

1. To read hex 56 bytes of data from device tape:

```
tape buf 56 rd
```

rpp

rpp: Conduct a Parallel Poll

Syntax: `rpp`

Remarks: `rpp` conducts a parallel poll of previously configured devices by sending the IDY message (ATN and EOI both asserted).

When done, the parallel poll response byte is passed back as the top element on the stack. The program should also check `iberr` to determine if the response byte is valid.

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

See Also: `ist`, `ppc`, `ppu`.

Example:

1. To remotely configure a device at listen address 0x23 to respond positively on DIO3 if its individual status bit is 1, and then parallel poll all configured devices:

```
3 6a ppc            ( configure device at )
                    ( listen address 3)
rpp                 ( parallel poll all )
                    ( configured devices )
                    ( response passed back on )
                    ( stack )
```

rsc

rsc: Request or Release System Control (SC)

Syntax: `v rsc`

Remarks: `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.

`rsc` is used to enable or disable the capability of the GPIB-CT to send the Interface Clear (IFC) and Remote Enable (REN) messages to GPIB devices using the `sic` and `sre` functions. The GPIB-CT GPIB port must not be System Controller to respond to Interface Clear sent by another Controller.

In most applications, the GPIB-CT will always be the System Controller. In other applications, the GPIB-CT will never be the System Controller. In either case, `rsc` is used only if the GPIB-CT 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, `rsc` would be used in such a scheme.

Example:

1. To request to be System Controller if the GPIB-CT GPIB port is not currently so designated:

```
1 rsc
```

rsp

`rsp:` Conduct a Serial Poll

Syntax: `addr rsp`

Remarks: `addr` is a valid GPIB address.

`rsp` is used to serially poll one device and obtain its status byte. If the 0x40 bit of the response is set, the status response is positive, i.e., the device is requesting service. Before `rsp` completes, all devices are unaddressed.

Upon completion, the serial poll response is returned in the low-order 8 bits of the word on the top of the stack. If the device did not respond within the allotted time a -1 will be returned on the top of the stack.

The interpretation of the response, 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.

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

`rsp` sends the following commands:

- UNL (Unlisten)
- SPE (Serial Poll Enable)
- GPIB-CT listen address
- Talk address of the device
- Read in response byte
- SPD (Serial Poll Disable)
- UNL (Unlisten)
- UNT (Untalk)

ATN and REN remain asserted after the function call.

Example:

1. To obtain the Serial Poll response byte from device tape:

```
tape rsp
```

rsv

rsv: Request Service and/or Set Serial Poll Status Byte

Syntax: `val rsv`

Remarks: `val` specifies the serial poll response byte.

If the 0x40 bit is set in `val`, the GPIB-CT additionally requests service from the Controller by asserting the GPIB SRQ line.

`rsv` 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 the GPIB-CT.

It is not an error to call `rsv` when the GPIB-CT 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.

See Also: `rsp`.

Examples:

1. To set the Serial Poll status byte to 0x41, which simultaneously requests service from an external CIC:

```
41 rsv
```

2. To stop requesting service (unassert SRQ):

```
0 rsv
```

3. To change the status byte to 1 without requesting service:

```
1 rsv
```

sic

sic: Send Interface Clear (IFC)

Syntax: `sic`

Remarks: `sic` causes the GPIB-CT to assert the IFC signal for at least 100 µsec, provided the GPIB-CT has System Controller authority. This action initializes the GPIB and makes the GPIB-CT GPIB port CIC. `sic` is generally used when you want to become CIC or clear a bus fault condition.

The IFC signal resets 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.

The GPIB-CT records the ESAC error if you have disabled its System Controller capability with the `rsc` function.

See Also: `rsc`.

Example:

1. To initialize the GPIB and become CIC at the beginning of a program:

```
sic
```

sre

sre: Set or Clear Remote Enable (REN)

Syntax: `v sre`

Remarks: `v` specifies set or clear.

`sre` 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 the GPIB-CT is not System Controller.

See Also: `cmd`, `loc`, `rsc`, `sic`.

Examples:

1. To place a device at listen address 0x23 (ASCII #) into remote mode:

```
1 sre        ( set REN to true )
" #" cmd     ( MLA )
```

2. To exclude the local ability of the device, send the Local Lockout command (0x11), or include it in the command string in Example 1:

```
11 buf c!    ( send LLO )
buf 1 cmd
```

or

```
1 sre              ( REN true )
23 buf c!          ( MLA LLO )
11 buf 1+ c!
buf 2 cmd
```

3. To return all devices to local mode:

```
0 sre    ( set REN to false )
```

stat

`stat:` Return GPIB-CT status

Syntax: `stat`

Remarks: `stat` is used to obtain the status of the GPIB-CT to see if certain conditions are currently present. Use `stat` most often to verify if the previous operation resulted in an error.

Use stat frequently in the early stages of program development when your device's responses are likely to be unpredictable.

Status represents a combination of GPIB-CT conditions. Internally in the GPIB-CT, status is stored as a 16-bit integer. Each bit in the integer represents a single condition. A bit value of 1 indicates that the corresponding condition is in effect; a bit value of zero indicates that the condition is not in effect. Since more than one GPIB-CT condition may exist at one time, more than one bit may be set in status. The highest order bit of status, also called the sign bit, is set when the GPIB-CT detects a GPIB error. Consequently, when status is negative, an error condition exists, and when status is positive, no error condition exists. Table 3-2 lists the indication of each bit in the status.

The status is returned on the stack as two words. The top number represents the 16-bit status of the GPIB-CT. The second number is the count of bytes last transferred using `brd`, `bwrt`, `rd` and `wrt`.

Table 3-2. GPIB Status Conditions

Numeric Value (n)	Symbolic Value (s)	Description	Bit
-32768	ERR	Error detected	15
16384	TIMO	Timeout	14
8192	END	EOI or EOS detected	13
4096	SRQI	SRQ detected while CIC	12
2048	-	Reserved	11
1024	-	Reserved	10
512	-	Reserved	9
256	CMPL	Operation completed	8
128	LOK	Lockout state	7
64	REM	Remote state	6
32	CIC	Controller-In-Charge	5
16	ATN	Attention asserted	4
8	TACS	Talker active	3
4	LACS	Listener active	2
2	DTAS	Device trigger state	1
1	DCAS	Device clear state	0

See Also: Appendix B in your GPIB-CT User Manual for a detailed description of the conditions under which each bit in status is set or cleared.

tmo

tmo: Change or Disable Timeout Limit

Syntax: `val tmo`

Remarks: `val` specifies the timeout limit, as shown in Table 3-3.

Table 3-3. Timeout Limit Values

Mnemonic	val	Minimum Timeout
TNONE	0	disabled
T10us	1	10 µsec
T30us	2	30 µsec
T100us	3	100 µsec
T300us	4	300 µsec
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

Notice 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.

Examples:

1. To change the time limit for device level I/O operations to 300 µsec:

```
4 tmo
```

2. To perform I/O operations with no timeout in effect:

```
0 tmo
```

trg

trg: Send Device Trigger

Syntax: `addr trg`

Remarks: `addr` is a valid GPIB address.

`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)

If this is the first function you call that requires GPIB controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

The response to a trigger is device dependent.

Example:

1. To trigger device `analyzer`:

```
analyzer trg
```

wait

`wait:` Wait for Selected Events

Syntax: `mask wait`

Remarks: 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 in Table 3-4.

Table 3-4. Wait Mask Layout

Decimal Value	Mnemonic	Description	Hex Value	Bit
-	-	Reserved	-	15
16384	TIMO	Timeout	4000	14
8192	END	EOI or EDS detected	2000	13
4096	SRQI	SRQ detected while CIC	1000	12
-	-	Reserved	-	11
-	-	Reserved	-	10
-	-	Reserved	-	9
-	-	Reserved	-	8
128	LOK	Lockout state	80	7
64	REM	Remote state	40	6
32	CIC	Controller-In-Charge	20	5
16	ATN	Attention asserted	10	4
8	TACS	Talker active	8	3
4	LACS	Listener active	4	2
2	DTAS	Device trigger state	2	1
1	DCAS	Device clear state	1	0

If `mask = 0`, the function returns immediately. This is used to report the current device or GPIB-CT 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; otherwise the GPIB-CT waits indefinitely for the event to occur.

All activity on the GPIB-CT GPIB port is suspended until the event occurs.

See Also: `stat`, `tmo`.

Examples:

1. To wait for a service request or a timeout:

```
5000 wait
```

2. To update the status:

```
0 wait
```

3. To wait indefinitely until control is passed from another CIC:

```
20 wait
```

4. To wait indefinitely until addressed to talk or listen by another CIC:

```
C wait
```


wrt

wrt: Write Data to GPIB

Syntax: `addr buf cnt wrt`

Remarks: `addr` is a valid GPIB address.

`buf` is the address of the buffer that contains the data to be sent over the GPIB.

`cnt` specifies the number of bytes to be sent over the GPIB.

`wrt` attempts to write `cnt` bytes of data to the specified GPIB device. The following steps are performed:

1. UNL is sent.
2. The device is addressed to listen and the GPIB-CT GPIB port to talk, if not already addressed to do so.
3. The GPIB-CT writes the data to the device.

An EABO error results if the operation does not complete within the time limit.

`wrt` terminates on any of the following events:

- When `cnt` bytes have been written
- Error is detected
- Time limit is exceeded
- When no listeners are detected after the operation begins (the GPIB-CT reports ENOL in this case)
- Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is CIC

When `wrt` returns, `ibcnt` contains the actual number of data bytes written. A short count can occur on any of the previous events but the first.

If this is the first function you call that requires GPIB Controller capability, and you have not disabled System Controller capability with `rsc`, the GPIB-CT sends Interface Clear (IFC) to make itself CIC. It also asserts Remote Enable.

If you passed control to some other GPIB device, control must be passed back to you or you must send IFC to make yourself CIC before making this call. Otherwise, the ECIC error will be posted.

Example:

1. To write 10 bytes of instructions to device `dvm`:

```
dvm " F3R1X5P2G0" wrt
```

NOTE: The double quote (") places text in memory up to the closing quote or decimal 65 characters. " also leaves the address and string length on the stack and is thus ideal for use with `wrt`. For instance, `" abc"` leaves the address of the string and a count of 3 on the stack.

Chapter 4
Programming Examples

This contains sample applications written in IBCL Some examples are standalone; others have software that communicates with ongoing IBCL code from an external computer. These examples are simplified so that you can enhance them to meet your own programming needs.

Microsoft BASIC IBCL Compiler Programming Example

Example 1 demonstrates how you can use a high-level language that has access to the system's serial port to download a file of IBCL source code, macros, or data to IBCL.

Example 1

```
10    CLS
20    INPUT "Enter the filename of the source code using a
      correct pathname: ";file$
30    OPEN file$ FOR INPUT AS #1
                                        ' Opens the disk file for
                                        ' input
40    OPEN "COM1:9600,N,8,1" AS #2
                                        ' Opens the RS-232 com-
                                        ' munications port
50    LINENUM = 1 : Locate 2,1 : PRINT "Compiling line # ";
      LINENUM                           ' Initialize a line count
                                        ' variable
60    CMDSTR$ = "ibcl"+CHR$(13)         ' Command to enter IBCL from
                                        ' the GPIB-CT default
                                        ' operating system.
70    PRINT #2,CMDSTR$                  ' Send the string to RS-232
                                        ' port
80    LINE INPUT #2,STAT$               ' Ensure ok message given
90    STAT$ = RIGHT$ (STAT$,2)
100   IF STAT$ <> "ok" THEN 500
                                        ' Check if successfully
                                        ' entered IBCL
110   while (not EOF(1))                ' Keep reading until disk
                                        ' file is depleted of words,
                                        ' macros, and commands.
```

```
120 LINE INPUT #1,CMDSTR$              ' Get command from disk file
130   PRINT #2,CMDSTR$                 ' Send command to IBCL
140   LINE INPUT #2,COPY$               ' Read in echo of CMDSTR$
                                        ' from IBCL
150   LINE INPUT #2,STAT$               ' Get status of last command
160   STAT$ = RIGHT$ (STAT$,2)
170   IF STAT$<> "ok" THEN 500          ' Some error occurred. Stop!
180   LINECNT = LINECNT + 1             ' Increment count of lines
                                        ' sent
190   WEND                              ' Repeat the loop
200   PRINT : PRINT LINECNT;" lines were successfully
      transmitted."
210 END

500   REM ' This error routine could be any action you
510   REM ' take when an error occurs in a transfer.
520   PRINT : PRINT "Transmission interrupted due to an
      error."
530   PRINT "Value of CMDSTR$ which created an error is
      ";CMDSTR$
540   PRINT "Value of STAT$ after the error is ";STAT$
550   END
```

Modem Programming Examples

Example 2 is an IBCL program that loops twenty decimal times reading the settings from an oscilloscope located at GPIB address 1. When all the readings are obtained, IBCL will use a modem and dial the number of a modem attached to another computer executing Example 3.

Example 3 is a Microsoft BASIC program that dials a modem connected to a distant GPIB-CT, downloads a file (the program from Example 2), and then waits for IBCL to call back with the results of its operation.

Example 2

```
0 variable buff                 ( Create a buffer named "buff")
1000 allot                      ( Allocate 1000 hex bytes for)
                                ( the buffer)
: analyze  ( Define a word "analyze")
     buff  ( Original buffer address onto)
                                ( the stack)
     14 0 do                    ( Loop 20 decimal times)
          dup dup               ( Duplicate the moving buffer)
                                ( address twice)
          c tmo                 ( Set up a 3 sec timeout)
```

```
            4000 wait           ( Wait 3 sec between data)
                                ( acquisitions)
            d tmo               ( Reset default 10 sec timeout)
            1 " set?" wrt       ( Request the instrument state)
                                ( of the Tektronix 2230)
                                ( oscilloscope located at GPIB)
                                ( address 1)
            1 swap 1000 rd      ( Read data from instrument at)
                                ( GPIB address 1 into the)
                                ( buffer whose address is on)
                                ( the stack)
            d swap              ( Put a <CR> on the stack and)
                                ( swap its value with the)
                                ( remaining buffer address)
            stat drop +         ( Get actual count of chars.)
                                ( read and add this value to)
                                ( the buffer address)
            c!                  ( Store the <CR> into the)
                                ( buffer)
            stat drop 1+ +      ( Get new buffer address by)
                                ( adding count of chars. read)
                                ( plus 1 for <CR>)
        loop ( Loop and do again)
        ." ATDT9,3358570"       ( Send command to modem serial)
                                ( port to dial)
        cr                      ( Send a <CR><LF>--." does not)
                                ( automatically)
        4000 wait               ( Wait 10 sec to insure)
                                ( Carrier Detect)
        buff - buff swap        ( Get starting address of)
                                ( buffer and count of chars.)
                                ( to send to the host)
        ulm  ( Upload the data to the)
                                ( serial port)
;
```

Example 3

```
10 CLS : KEY OFF
20 DIM RESULT$(20)                  ' Create an array for results
30 ON COM(1) GOSUB 520              ' Trap routine for incoming
                                    ' serial data
40 INPUT "filename"; FILE$          ' Disk file of IBCL data,
                                    ' program, etc.
50 OPEN FILE$ FOR INPUT AS #1       ' Ope n IBCL command file
60 OPEN "com1:1200,n,8,1" AS #2
70 PRINT #2,"ATs2=42"               ' Local modem escape character
                                    ' is the asterisk (*)
80 LFCR$=INPUT$(2,2)                ' Get leading CR and LF
```

```
90 LINE INPUT #2,ESCSTAT$              ' Status from modem
100 LF$=INPUT$(1,2)                    ' Get trailing LF
110 PRINT #2,"ATDT9,335857O"           ' Dial IBCL modem
120 LFCR$=INPUT$(2,2)                  ' Get leading CR and LF
130 LINE INPUT #2,DLSTAT$              ' Status from modem
140 LF$ = INPUT$(1,2)                  ' Get trailing LF
150 IF INSTR(DLSTAT$,"CONNECT") = 0 THEN PRINT
   "No carrier detected" : END
                                       ' Check if CONNECT message
                                       ' received
160 FOR PAUSE = 1 TO 3000 : NEXT PAUSE
                                       ' Wait to ensure Carrier
                                       ' Detect
170 PRINT #2,"task"                    ' An IBCL do-nothing word -
                                       ' insures that IBCL
                                       ' is ready for commands
180 LINE INPUT #2,COPY$                ' Echo of IBCL command
190 LINE INPUT #2,STAT$                ' Status after IBCL processes
                                       ' command
200 LF$=INPUT$(1,2)                    ' Get trailing LF
210 PRINT #2,"cold"                    ' OPTIONAL--IBCL will be in a
                                       ' known state after execution
                                       ' of this command
220 LINE INPUT #2,COPY$                ' Echo of IBCL command
230 LINE INPUT #2,STAT$                ' Status after IBCL processes
                                       ' command
240 LF$=INPUT$(1,2)                    ' Get trailing LF
250 IF INSTR(STAT$,"ok") = 0 THEN PRINT "IBCL is not
   responding.  Program terminated" : END
                                       ' Check if status was an ok
                                       ' message
260 LINENUM = 1  : LOCATE 2,1 : PRINT "Compiling line
   # ";LINENUM
270 IF EOF(1) THEN PRINT "downloaded file" : GOTO 400
280 LINE INPUT #1,CMDSTR$              ' Get next IBCL command from
                                       ' file
290 PRINT #2,CMDSTR$                   ' Send IBCL command to GPIB-CT
                                       ' to be compiled
300 LINE INPUT #2,COPY$                ' Receive echoed command line -
                                       ' disregard
310 LF$ = INPUT$(1,2)                  ' Get trailing LF
320 LINE INPUT #2,STAT$                ' receive status line (ok or
                                       ' error message)
330 LF$ = INPUT$(1,2)                  ' Disregard line feed character
340 IF STAT$ <> "ok" THEN GOTO 380
                                       ' if not ok then go to error
                                       ' routine
350 LINENUM = LINENUM + 1              ' increment line number
```

```
360 LOCATE 2,18 : PRINT LINENUM
370 GOTO 270
380 PRINT : PRINT "Compile error on line number " ; LINENUM     ' error routine
390 PRINT CMDSTR$ : PRINT STAT$ : END
400 PRINT #2, "analyze"                    ' Name of the program just
                                           ' downloaded--'analyze' in our
                                           ' example
410 FOR PAUSE = 1 TO 5000 : NEXT PAUSE
                                           ' Modem guard time for esc.
                                           ' sequence
420 PRINT #2,"***";                        ' Enter modem escape mode
430 FOR PAUSE = 1 TO 5000 : NEXT PAUSE
                                           ' Modem guard time for esc.
                                           ' sequence
440 RESP$=INPUT$(LOC(2),2)                 ' Get any characters in comm.
                                           ' buffer-disregard these
450 PRINT #2,"ATH0"                        ' Modem command to hang up
                                           ' phone
460 LFCR$=INPUT$(2,2)                      ' Get CR and LF
470 LINE INPUT #2,STATUS$                  ' Get hang-up status string
480 LF$=INPUT$(1,2)                        ' Get trailing LF
490 REM The following section of code is the trap routine for
   incoming serial data.
500 COM(1) ON                              ' Enable communications
                                           ' trapping
510 GOTO 510                               ' No meaningful work to be
                                           ' done, so wait for results
520 COM(1) OFF                             ' Stop communications trapping
530 CLS:PRINT "Now getting ring and connect status"
540 CRLF$=INPUT$(2,2)                      ' Get leading CR and LF
550 LINE INPUT #2,RING$                    ' Get ring status string (RING)
560 LF$=INPUT$(1,2)                        ' Get trai ling LF
570 WHILE RING$="RING"                     ' Keep getting ring status
                                           ' string until not = to RING
580     CRLF$=INPUT$(2,2)
590     LINE INPUT #2,RING$
600     LF$=INPUT$(1,2)
610 WEND
620 FOR PAUSE = 1 TO 3000 : NEXT PAUSE
                                           ' Wait to insure Carrier Detect
630 IF INSTR(RING$,"CONNECT") = 0 THEN PRINT "No carrier
   detected" : END                         ' Stop program if no Carrier
                                           ' Detected
640 ON COM(1) GOSUB 670                    ' New trap address
650 COM(1) ON                              ' Enable communications
                                           ' trapping
660 GOTO 660                               ' No meaningful work, so wait
670 COM(1) OFF                             ' Disable communications
```

```
                                        ' trapping
680 CLS:PRINT "Now waiting for the results"
690 FOR RESULT = 1 TO 20
700 LINE INPUT #2,RESULT$(RESULT)
                                        ' Get a result string
                                        ' from the communications
                                        ' buffer
710 PRINT : PRINT RESULT$(RESULT)
720 NEXT RESULT
730 FOR PAUSE = 1 TO  5000 : NEXT PAUSE
                                        ' Modem escape guard time
740 PRINT #2,"***";                     ' Local modem escape sequence
750 FOR PAUSE = 1 TO 5000 : NEXT PAUSE
                                        ' Modem escape guard time
760 IF LOC(2) <> 0 THEN RESP$=INPUT$(LOC(2),2)
                                        ' Insure comm. buffer empty
770 PRINT #2,"ATH0"                     ' Command for modem to hang-up
780 CRLF$=INPUT$(2,2)                   ' Get leading CR and LF
790 LINE INPUT #2,STAT$                 ' Get modem status
800 CLOSE #2                            ' Close the communications port
810 CLOSE #1                            ' Close the disk file
820 END
```

Macro Programming Example

Example 4 is an IBCL macro that will set your GPIB-CT to default values different from those of the GPIB-CT default operating system. For this example, assume you want the following default values for the GPIB-CT default operating system that differ from those at startup or at ONL 1:

- Do not send EOI on the last byte of a GPIB write
- Send EOI with the carriage return (ASCII 13)
- Terminate GPIB reads upon receiving a carriage return
- Configure an IBM 7372 color plotter to participate in a parallel poll by returning a positive response when its pen is down
- Set timeout limit at 3 sec

In the GPIB-CT default operating system, you would have to type all of the instructions each time that you wanted these changes to occur. However, with the addition of the IBCL operating system, you have the ability to create a macro called mydefault that will do these steps for you.

This solution still requires you to type in the macro definition at startup, but it will be there whenever needed thereafter (for instance, after an ONL 1 is executed and all startup defaults are reset). For an even more powerful macro ability, after studying this example, refer to Appendix C, *Creating Permanent IBCL Words in EPROM*, to learn how to make the macros permanent and avoid typing in each macro definition after each startup. Notice that mydefault is defined the same way as any other IBCL word. It is called a macro because its function resembles that of a macro. There is no special defining technique required for defining macros.

Example 4

Follow these steps to create a macro named mydefault which sets the five previously described defaults:

1. In the GPIB-CT default operating system, enter:

   ibcl<CR>

2. If you are using a terminal emulator program, wait for an ok message from IBCL. If you are in BASIC or some other language, read in the status string and check for ok. The ok appears instantly if there is no problem.

3. Now, create an IBCL word which will be the macro:

```
: mydefault                 ( Macro name)
    0 eot                   ( Disable EOI sent with the)
                            ( last byte of GPIB writes)
    C0D eos                 ( Enable EOI sent with <CR>)
                            ( and GPIB reads to terminate)
                            ( when a <CR> is received)
    5 " im 223,0,1;" wrt
                            ( Sends to 7372 plotter at)
                            ( GPIB address 5 the command)
                            ( to participate in a parallel)
                            ( poll when it's pen is down)
    c tmo                   ( Change timeout limit to 3)
                            ( sec)
    bye                     ( Inclusion of this command)
                            ( will cause a return to the)
```

```
                                    ( GPIB-CT default operating)
                                    ( system)
                                    ( immediately upon completion)
                                    ( of the macro.  Leaving this)
                                    ( statement out allows you to)
                                    ( remain in IBCL after)
                                    ( execution of the macro.)
        ;                           ( End the macro definition)
```

This example demonstrates only a fraction of the power available to you with IBCL and macros. After you have completed this example, you can type mydefault if you are operating in IBCL, or ibcl<CR> mydefault<CR> if you are in the GPIB-CT default operating system, to set these defaults.

Timed Applications Examples

Example 5

Within IBCL, you can access the on-board system timer to time activities. To do this, you must convert the required time limit into a 4-byte value that the timer can use and load those values into four specified memory locations within IBCL.

To program any value, you may derive the values required for IBCL in the following manner:

- The first number is the actual number of timer interrupts that will occur before the routine completes. Valid values for this number lie in the range of 1 to 65535.

- The second number is the length of time before a timer interrupt will occur. Valid values for this number lie in the range of 3 to 65535. Each increment in the second number represents a time of 3.26 µsec (0.0000326 sec). Therefore, the minimum value of time which can be generated is roughly 10 µsec (3, the minimum value of the second number, times 3.26 µsec is roughly equal to 10 µsec). Incrementing the second number by 1 produces a timer value of roughly 13 µsec (4, the minimum value of the second number, + 1 * 3.26 µsec is roughly equal to 13 µsec).

A useful general formula is given here:

(VAL2 * 3.26 µsec) * VAL1 ≈ desired time value

Example 6

Assume you have an application which requires servicing approximately every 24 µsec Using the previous formula, VAL2 = 8 and VAL1 = 1 produce the proper values for this application ((8 * 3 µsec) * 1 ≈ 24 µsec).

Example 7

Assume you have an application requiring service at approximately 48 µsec intervals. One way to produce this value is to let VAL1 = 2 ((8 * 3 µsec) * 2 ≈ 48 µsec). However, this is not the most efficient solution because the timer interrupt has to be serviced more often than necessary. A more efficient method of achieving the same time limit is to change VAL2 instead of VAL1 ((16 * 3 µsec) * 1 ≈ 48 µsec). Of course, when VAL2 becomes larger than 65535, VAL1 will have to be adjusted to accommodate longer times.

After you have these numbers (VAL1 and VAL2 comprise the necessary 4 bytes for the timer), they have to be loaded into the memory locations starting at 1C hex using the IBCL command ! (pronounced store). In this example, the values used are from the 24 µsec given in Example 6.

First, store VAL1, the actual number of timer interrupts at memory location 1C hex by typing:

```
1 1c !
```

Next, store VAL2, the time interval between interrupts, at memory location 1E hex by typing:

```
8 1e !
```

Example 8

Suppose you have an application that takes measurements at 5 sec intervals. You want to continuously read the data (guaranteed to be 2 bytes long) into a buffer for a duration of 10 min. Follow these steps:

1. Set up a buffer in IBCL to accommodate the readings by typing the following:

   0 variable buff ee allot <CR>

   This step creates a buffer named buff and allocates 240 (12 readings/min * 10 min * 2 bytes/reading) decimal bytes of space for the readings.

2. Derive the values which should be stored at location 1c hex to ensure proper timing using the following formula:

   5 sec = 5,000,000 µsec/ 65500 (almost as large as possible) / 3 = VAL1 = number of times timer must interrupt = 19 hex and 65500 = ffdc hex = VAL2 = time before an interrupt occurs.

3. Create a word that will perform the specified application:

```
: analysis                    ( Define a word to take)
                              ( readings)
    20 1c do                  ( ** See warning below-copies)
                              ( old timer values)
     i c@ >r                  ( Get byte and put onto)
                              ( return stack)
    loop
    19 1c !                   ( Store the new timer values)
    ffdc 1e !

    buff dup                  ( Put the address of buffer)
                              ( on stack)
    5 " put cmd here" wrt
                              ( Issue command for the)
                              ( device)
                              ( at GPIB address 5 to start)
                              ( taking measurements.)
    120 0 do                  ( Take the 120, 2 byte)
                              ( readings)
```

```
        dup 2+ swap             ( Get next buffer location)
                                ( and leave old buffer)
                                ( location on top of stack)
        5 swap 2                ( Put the GPIB address of)
                                ( device on the stack, swap)
                                ( with old buffer location,)
                                ( and put the number of)
                                ( characters to read {2} on)
                                ( top. This leaves the)
                                ( parameters required for a)
                                ( GPIB read on the stack--)
                                ( addr{5} buffer{old buffer})
                                ( and count {2})
        4000 wait               ( Wait the required 5 sec)
        rd                      ( Execute a GPIB read of 2)
                                ( chars into the buffer)
    loop                        ( Start the loop again)
    4 0 do                      ( Get old timer values from)
                                ( return stack and put them)
        r>                      ( on the computation )
    loop                        ( stack)
    20 1c do                    ( Store the old values)
        i c!
    loop
;
```

6. Type the following to execute the program:

```
analysis<CR>
```

Warning: Using this method of setting timeout limits will work and will remain in effect until changed again by this method or by the tmo command. However, if the >r and r> blocks of code are omitted from the previous example, the reporting of timeout limits in the GPIB-CT default operating system will be incorrect because the string holding the TMO value will not be changed. So, although the string contains one value, the actual value in the timer routine will be your last value stored. By using the >r and r> blocks in the previous example, the string reports the correct value as the values before you make any changes will have been reset at the end of execution of the code.

Chapter 5
Technical Information

This chapter contains information for improving and customizing performance from the GPIB-CT.

Loading Programs

There are several ways in which you can load IBCL source code. Since IBCL treats incoming source code as normal text, any method you have of sending data over the serial port is an effective way to download source code into IBCL.

The easiest way to communicate with the GPIB-CT is through a terminal emulation program. Using a terminal emulator is a preferred way of creating and debugging an application as you can see everything sent and received over the serial port at one time. If you are using a terminal emulation program, downloading source code is as easy as sending a text file of the code over the serial port.

Another way you could download source code is through a programming language which has access to the system's serial port. An example using this method is provided in Section Four, *Programming Examples*, Microsoft BASIC IBCL Compiler Programming Example.

If you wish to make your code permanent in IBCL after downloading, see Appendix C, *Creating Permanent IBCL Words in EPROM*, for instructions.

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 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 a new word.

Inner Interpreter Sequence

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 proceeds 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 compiled by (docol) which is an IBCL word to which users have no access. This primitive nests control to a lower-level definition.

;s 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. (.") controls display of the subsequent in-line string and causes interpretation to skip to the word subsequent to that string, while (abort) could cause execution of a user-supplied routine in the event of an error. lit causes the next word value to be pushed onto the stack; interpretation continues after that value.

execute causes a branch to the word pointed to by the top value on the stack, just as if the pointer to that word's 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 subsequent 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, these words 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 one line at a time from the host. A line can be up to 80 bytes long. The interpreter further breaks each line or block into individual words and processes them sequentially. A word is a string of characters preceded and followed by blank spaces or by a <CR>. 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 an opening parenthesis (). The interpreter ignores input after the (word until the next closing parenthesis ()) or until the end of the current line. 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 immediately–even within a colon definition.

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 numerical base. 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. 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 previous numbers. dpl can be 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 behavior is similar except that each byte-pair will be compiled separately, along with pointers to lit.

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

The interpreter uses -find 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 compiles (number) which does the conversion. (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 ? MSG #0 error. The user variable dpl will contain -1 if no decimal point was present in the numeric string. In this case, the number was of single word length and the top word on the stack may be dropped. 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 the discussions *Defining New Words*, *Colon Definitions*, and *IBCL Input*, in Chapter 2, *IBCL Function Reference*.

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

Defining a new word increases the memory allocated in the dictionary. If an error causes the definition to abort before completion, memory allocated in the dictionary is reclaimed.

Advanced Defining Techniques

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

```
: constant <builds , does> @ ;
```

does> is an immediate word. It executes when the definition is entered. The @ is compiled as usual.

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

```
5 constant five <CR>
```

The 5 is placed on the stack and momentarily ignored. Referring to the definition of constant, the <builds 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. This word is initialized with a pointer to the code for constants, and the working end of the definition list is incremented by two to point to the parameter field of the word being defined, five. Next the comma (,) takes the top value on the stack, 5, and stores it at the working end of the definition list, the parameter field of five, and increments the end pointer by two.

Next, the does> replaces the contents of the code field with a pointer to a few bytes of code created by does> each time it is used. This 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 a routine to be compiled that unnests the interpreter one level, and then terminates the definition.

When five is executed, the code created by does> is executed. The address of the parameter field of five 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.

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

```
: dconstant <builds swap , ,
does> dup @ swap 2+ @ ;

hex 1234.5678 dconstant longfellow
```

dconstant creates 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.

The complexity and utility of words that define words is unlimited.

Machine Code Primitives

This discussion provides 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.

The GPIB-CT contains a Hitachi HD64180 microprocessor which has an instruction set that is a superset of the Z-80 instruction set. You should be familiar with the Z-80 instruction set before attempting machine code primitives.

Three addresses very important to creating your own machine code primitives are dpush, hpush, and next. There are no given system constants for these addresses, although the fixed offsets from the origin are 6A hex for dpush, 6B hex for hpush, and 6C hex for next. To get the true address of these words, you can enter this line:

```
offset +origin u.
```

The 64180 has six general-purpose registers–B, C, D, E, F, G, H, and L. You can combine these six into three general-purpose 16-bit registers (BC, DE, and HL), an 8-bit accumulator, an 8-bit flag register (F), and two index registers (IX and IY). Along with these registers, the 64180 has an alternate register set (HL', BC', DE', and AF').

dpush is used to push onto the stack the value stored in the DE register pair, and then push onto the stack the value stored in the HL register pair. hpush is used when you wish to push onto the stack only the value stored in the HL register pair. next is a routine which returns control to the IBCL operating system after execution of the primitive. dpush and hpush automatically execute next, but in all primitives which do not put anything onto the stack with dpush or hpush, you will specifically need to jump to next at the end of your primitive.

Warning: IBCL uses the value in register pair BC as its address pointer; therefore, register pair BC should be used with extreme caution to prevent a system crash.

The following example, fastadd, demonstrates these topics. Notice that this is how the + word is defined, so this example will not execute faster than the existing + operation.

Enter this sequence:

```
create    fastadd                    ( Makes a dictionary )
                                     (entry for fastadd)
e1            c,                     ( pop hl)
d1            c,                     ( pop de)
19            c,                     ( add hl, de)
c3            c, 6B c, 2 c,          ( jp hpush)
( see note about hpush following example)
smudge        ( Toggle the definition's)
                                     ( smudge kit to allow)
                                     ( fastadd to be found in)
                                     ( dictionary searches)
```

Note: In the current version of the IBCL software, the code for hpush is at 026Bhex. This code is not guaranteed to remain in that location in future software revisions. Before doing this step, you should find out the address of hpush as detailed earlier in this section in the discussion of +origin.

In the previous example, create makes a dictionary entry for fastadd and leaves the dictionary pointer at the code field of fastadd. c, then puts the byte on the stack into the memory location pointed to by the dictionary pointer and increments the dictionary pointer. This process places the machine language sequence into the dictionary.

Vectored Execution

You cannot 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, or vector, of the word to be defined.

Consider the following example:

```
    nnnn            variable        vector-name
( creates a variable called vector-name, initialized to)
( nnnn, although this value does not matter)

    : some-word     words           vector-name @ execute words;
( defines some-word, which needs to use a currently)
( undefined word)

    : future-word   words ;
( define the future word)

    future-word cfa vector-name !
( put the code field of future-word into the vector)

    some-word       ( execute the complete 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 as if it had been compiled into the definition.

Memory Organization

Figure 5-1 is a logical memory map of the IBCL operating system. IBCL memory space is actually located from physical 40000H to 4FFFFH, but IBCL recognizes only 16-bit addresses 0 through FFFFH.

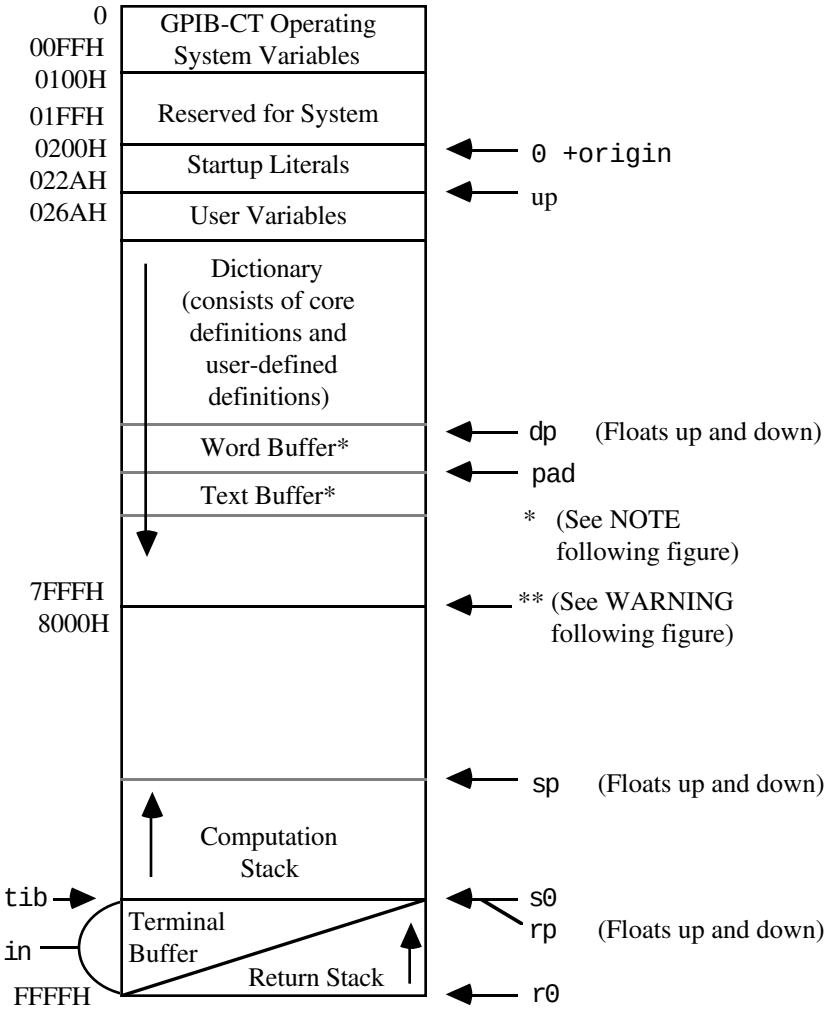


Figure 5-1. Logical Memory Map

Note: These buffers float immediately above the dictionary at a fixed offset from the dictionary pointer.

Warning: After expanding the IBCL dictionary past logical address 7FFFH, any definitions made are not guaranteed to remain if you leave IBCL to the GPIB-CT default operating system, because the GPIB-CT default operating system uses addresses starting at 8000H as a serial port input buffer.

The operating system variables at the top of IBCL space are shared between the GPIB-CT default and IBCL operating systems. This sharing of resources ensures that any changes made to the characteristics of the GPIB-CT in either operating system are present in the other operating system as well.

Logical space from 100H to lFFH is reserved for the system and should never be changed.

Startup literals begin at the origin of IBCL, 200H. These values are necessary for system initialization. Some literals specify values which are copied to the user area of memory during initialization. Other literals specify the starting address of the user area, the ASCII code of the GPIB-CT's backspace character, and pointers to the top definition and the end of the core dictionary.

The user variables section contains the user variables dp, fence, r0, s0, tib, voc-link, warning, width, base, context, current, in, out, dpl, csp, hld, and state. For more information on the user variables, refer to the discussion *Constants, Variables, and Arrays*, in Chapter 2, *IBCL Function Reference*.

The dictionary is the largest and most essential element of IBCL. This area contains the definitions of all core and user-created words. The dictionary area expands in memory as new words are defined, and contracts when word definitions are deleted. The user variable dp always contains the address of the next available dictionary location and floats as the dictionary grows and shrinks.

The word buffer floats immediately above the top of the dictionary, beginning at the location stored in dp. The fixed length of the buffer is 68 characters. As the dictionary expands and contracts, the limits of the word buffer move the same distance up or down in memory without retaining its contents.

The text interpreter parses the input stream by obtaining individual words from the terminal input buffer and placing them into the word buffer.

The text buffer lies immediately above the word buffer a fixed distance above the dictionary. Like the word buffer, this area goes up or down in memory in response to dictionary movements, without saving its contents during relocation. This buffer serves as a scratchpad area where output text strings may be constructed character by character, or the IBCL word quote (") constructs the buffer for bwrt, wrt, or cmd. This buffer is the same length as an output line that is stored in the system constant c/l and defaults to 64 decimal bytes.

The terminal buffer and return stack share memory from address FFFFH to FF60H. As IBCL calls other words in a definition, the return stack grows towards low memory. As IBCL returns from each level of execution, the return stack shrinks towards high memory.

The terminal buffer begins at the value stored in tib and grows toward high memory. This buffer holds each line of input data from the serial port. As soon as a line is entered and processed, the buffer is reset for the next line.

The computation stack is the stack where parameters are stored. It is based in s0, and grows towards low memory and the dictionary. sp contains the stack pointer at all times. The value in sp can be viewed and altered using sp@ and sp!.

General Port I/O

IBCL provides two port input/output words, p! and p@, which change or recall the internal parameters of the GPIB-CT. These words transfer data between the top of the stack and any of the on-board GPIB-CT I/O ports. In normal applications, these words should be used only in the following circumstances:

- To read the states of the user defined switch (U20)
- To set up the DMA controller for GPIB reads and writes to extended memory

All other port accesses should be done with extreme caution. Improper use can cause the system to crash.

p! outputs a byte to the I/O port address represented by the word on top of the stack. The byte to be output is in the low-order position of the second word on the stack. The high-order byte of the second word is ignored.

p@ inputs a byte from the I/O port address represented by the word on the top of the stack. The byte input replaces the I/O address on the top of the stack and the high-order bytes of the word is zero-filled.

Table 5-1 is an I/O system map of the ports supported on the GPIB-CT. Only the port addresses noted should be used. Any access to any other addresses could produce unexpected results.

Table 5-1. I/O System Map of Ports Supported on the GPIB-CT

Name	Address
Asynchronous Serial Communication Interface (ASCI):	
ASCI Control Register A, Channel 0	00H
ASCI Control Register A, Channel 1	01H
ASCI Control Register B, Channel 0	02H
ASCI Control Register B, Channel 1	03H
ASCI Status Register, Channel 0	04H
ASCI Status Register, Channel 1	05H
ASCI Transmit Data Register, Channel 0	06H
ASCI Transmit Data Register, Channel 1	07H
ASCI Receive Data Register, Channel 0	08H
ASCI Receive Data Register, Channel 1	09H
Programmable Reload Timer (PRT):	
PRT Data Register, Channel 0L	0CH
PRT Data Register, Channel 0H	0DH
Reload Register, Channel 0L	0EH
Reload Register, Channel 0H	0FH
Timer Control Register	10H
PRT Data Register, Channel 1L	14H
PRT Data Register, Channel 1H	15H
Reload Register, Channel 1L	16H
Reload Register, Channel 1H	17H

(continues)

Table 5-1. I/O System Map of Ports Supported on the GPIB-CT (continued)

Name	Address
Direct Memory Access (DMA):	
DMA Source Address Register, Channel 0L	20H
DMA Source Address Register, Channel 0H	21H
DMA Source Address Register, Channel 0B	22H
DMA Destination Address Register, Chan. 0L	23H
DMA Destination Address Register, Chan. 0H	24H
DMA Destination Address Register, Chan. 0B	25H
DMA Byte Count Register, Channel 0L	26H
DMA Byte Count Register, Channel 0H	27H
DMA Memory Address Register, Channel 1L	28H
DMA Memory Address Register, Channel 1H	29H
DMA Memory Address Register, Channel 1B	2AH
DMA I/O Address Register, Channel 1L	2BH
DMA I/O Address Register, Channel 1H	2CH
DMA Byte Count Register, Channel 1L	2EH
DMA Byte Count Register, Channel 1H	2FH
DMA Status Register	30H
DMA Mode Register	31H
DMA/WAIT Control Register	32H
Interrupts:	
IL Register (Interrupt Vector Low Register)	33H
INT/TRAP Control Register	34H
Dynamic RAM Refresh:	
Refresh Control Register	36H
Memory Management Unit (MMU):	
MMU Common Base Register	38H
MMU Bank Base Register	39H
MMU Common/Bank Area Register	3AH
I/O Control Register	3FH
* See NOTE following table.	
GPIB Controller Read Only I/O Address Registers:	
Data In Register	40H
Interrupt Status Register 1	41H
Interrupt Status Register 2	42H
Serial Poll Status Register	43H
Address Status Register	44H

(continues)

Table 5-1. I/O System Map of Ports Supported on the GPIB-CT (continued)

Name	Address
GPIB Controller Read Only I/O Address Registers (continued):	
Command Pass Through Register	45H
Address Register 0	46H
Address Register 1	47H
GPIB Controller Write Only I/O Address Registers:	
Command/Data Out Register	40H
Interrupt Mask Register 1	41H
Interrupt Mask Register 2	42H
Serial Poll Mode Register	43H
Address Mode Register	44H
Auxiliary Mode Register	45H
Address Register 0/1	46H
End of String Register 47H	
* See NOTE following table.	
GPIB Controller DMA Acknowledge Register:	
DMA Acknowledge Register	48H
Board Registers:	
Board Control Register (write only, controls front panel LEDs)	50H
Switch 1 Register (read only, settings of DIP Switch U20; use this register to set your own switch configurations)	68H
Switch 2 Register (read only, settings of DIP Switch U22; this is used by the GPIB-CT operating system)	70H

Note: I/O addresses in the range of 00H to 3FH are internal to the microprocessor (HD64180). For specific information about what each bit represents in each I/O register, refer to the *HD64180 8-Bit High Integration CMOS Microprocessor User Manual*, available from Hitachi America, Ltd.

I/O addresses in the range of 40H-47H are internal to the GPIB Controller chip used in the GPIB-CT. For specific information about what each bit represents in each I/O register, refer to the section describing the µPD7210 intelligent GPIB controller chip in *NEC Microcomputer Products*, available from NEC Electronics, Inc. This description is used for interface products that contain the NAT4882 controller chip as well as interface products that contain the µPD7210 controller chip.

Appendix A
Multiline Interface Messages

The following tables are multiline interface messages (sent and received with ATN TRUE).

The subsequent pages contain an interface message reference list, which describes the mnemonics and messages which correspond to the interface functions.

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL		20	040	32	SP	MLA0
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	PPC	25	045	37	%	MLA5
06	006	6	ACK		26	046	38	&	MLA6
07	007	7	BEL		27	047	39	'	MLA7
08	010	8	BS	GET	28	050	40	(	MLA8
09	011	9	HT	TCT	29	051	41	)	MLA9
0A	012	10	LF		2A	052	42	*	MLA10
0B	013	11	VT		2B	053	43	+	MLA11
0C	014	12	FF		2C	054	44	,	MLA12
0D	015	13	CR		2D	055	45	-	MLA13
0E	016	14	SO		2E	056	46	.	MLA14
0F	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	US		3F	077	63	?	UNL

Multiline Interface Messages

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	O	MTA15	6F	157	111	o	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	

Interface Message Reference List

Mnemonic	Message	Interface Function(s)
LOCAL MESSAGES RECEIVED (by interface functions)		
gts	go to standby	C
ist	individual status qualifier	PP
lon	listen only	L, LE
[lpe]	local poll enable	PP
ltn	listen	L, LE
lun	local unlisten	L, LE
nba	new byte available	SH
pon	power on	SH, AH, T, TE, L, LE, SR, RL, PP, C
rdy	ready	AH
rpp	request parallel poll	C
rsc	request system control	C
rsv	request service	SR
rtl	return to local	RL
sic	send interface clear	C
sre	send remote enable	C
tca	take control asynchronously	C
tcs	take control synchronously	AH, C
ton	talk only	T, TE
REMOTE MESSAGES RECEIVED		
ATN	attention	SH, AH, T, TE, L, LE, PP, C
DAB	data byte	(via L, LE)
DAC	data accepted	SH
DAV	data valid	AH
DCL	device clear	DC
END	end	(via L, LE)
GET	group execute trigger	DT
GTL	go to local	RL
IDY	identify	L, LE, PP
IFC	interface clear	T, TE, L, LE, C
LLO	local lockout	RL
MLA	my listen address	L, LE, RL
[MLA]	my listen address	T
MSA or [MSA]	my secondary address	TE, LE
MTA	my talk address	T, TE
[MTA]	my talk address	L
OSA	other secondary address	TE
OTA	other talk address	T, TE
PCG	primary command group	TE, LE, PP

Interface Message Reference List (continued)

Mnemonic	Message	Interface Function(s)
REMOTE MESSAGES RECEIVED (continued)		
PPC	parallel poll configure	PP
[PPD]	parallel poll disable	PP
[PPE]	parallel poll enable	PP
PPRn	parallel poll response n	(via C)
PPU	parallel poll unconfigure	PP
REN	remote enable	RL
RFD	ready for data	SH
RQS	request service	(via L, LE)
[SDC]	selected device clear	DC
SPD	serial poll disable	T, TE
SPE	serial poll enable	T, TE
SRQ	service request	(via C)
STB	status byte	(via L, LE)
TCT or [TCT]	take control	C
UNL	unlisten	L, LE
REMOTE MESSAGES SENT		
ATN	attention	C
DAB	data byte	
DAC	data accepted	AH
DAV	data valid	SH
DCL	device clear	(via C)
END	end (via T)	
GET	group execute trigger	(via C)
GTL	go to local	(via C)
IDY	identify	C
IFC	interface clear	C
LLO	local lockout	(via C)
MLA or [MLA]	my listen address	(via C)
MSA or [MSA]	my secondary address	(via C)
MTA or [MTA]	my talk address	(via C)
OSA	other secondary address	(via C)
OTA	other talk address	(via C)
PCG	primary command group	(via C)
PPC	parallel poll configure	(via C)
[PPD]	parallel poll disable	(via C)
[PPE]	parallel poll enable	(via C)
PPRn	parallel poll response n	PP
PPU	parallel poll unconfigure	(via C)
REN	remote enable	C
RFD	ready for data	AH

Interface Message Reference List (continued)

Mnemonic	Message	Interface Function(s)
REMOTE MESSAGES SENT (continued)		
RQS	request service	T, TE
[SDC]	selected device clear	(via C)
SPD	serial poll disable	(via C)
SPE	serial poll enable	(via C)
SRQ	service request	SR
STB	status byte	(via T, TE)
TCT	take control	(via C)
UNL	unlisten	(via C)
UNT	untalk	(via C)

Appendix B
IBCL Status and Error Messages

This appendix contains a table of the IBCL status and error messages.

Table B-1 contains a list of the status and error messages returned by IBCL and a description of each. Message numbers shown are in decimal.

Table B-1. IBCL Status and Error Messages

Message Number (MSG #)	**Description**
0	Unrecognized dictionary word
1	Empty stack
2	Dictionary full
3	Has incorrect address mode
4	Is not unique (dictionary word redefined)
7	Full stack
17	Legal only within a colon definition
18	Not legal within a colon definition
19	Conditionals not paired (for example, IF but no THEN)
20	Definition not finished
21	In protected dictionary
24	Declare vocabulary

Appendix C
Creating Permanent IBCL Words in EPROM

This appendix describes the procedure for permanently adding new words and data to the IBCL operating system. It will also explain how to automatically run a permanently-saved application when the GPIB-CT is powered-on.

All newly defined IBCL words are compiled and stored into the dictionary which is stored in the system's dynamic RAM. Since dynamic RAM is volatile, its contents will be lost if power to the unit is removed.

Compiled words and data can be permanently added to the IBCL system by including them in an unused section of the system's EPROM. Then, each time the unit is powered on, the new IBCL system stored in the EPROM will be copied to and run out of RAM.

To add additional code to the EPROM of the GPIB-CT you will need an EPROM programmer and software as well as a blank 27256 or 27C256 EPROM with a maximum access time of 150 nsec. **Do not** reprogram the EPROM provided with the system. The system EPROM is like a master diskette–once it is copied it should be put aside for safekeeping. You may only copy the GPIB-CT system EPROM to add code to its dictionary, because the operating system within the EPROM is copyrighted.

Follow these steps to permanently save a new custom IBCL dictionary in EPROM:

1. Enter IBCL and create your extended dictionary. This can be done several ways. The most common method uses colon definitions to compile new words into the dictionary, and uses `constant`, `variable` and `allot` to add data to the dictionary.

   Make sure that your words are fully tested and debugged before attempting to put them in EPROM. It is much easier to make changes while you are running in RAM than to program another EPROM.

2. After you have compiled and debugged your code, you need to verify that the extended dictionary will fit into the available EPROM space. A 27256 EPROM has 32K (8000 hex) bytes of storage. Approximately 11,520 (2D00 hex) bytes are taken up by the GPIB-CT operating system. This leaves 21,248 (5300 hex) bytes available for the IBCL system and your extended words.

   The IBCL operating system starts at 200 hex. Thus, if you enter the `hex here 200 - u.` command string and the value that is returned is less than 5300, then the added code will fit into the 27256 EPROM. If the value returned is greater than 5300, then the extended dictionary exceeds the capacity of the EPROM. This probably means that you have allocated extremely large amounts of buffer space, because compiled IBCL code is very compact.

   Here are some tips for reducing the amount of storage space required for the dictionary due to buffer space:

   - Be realistic when allocating space for buffers. Do not allocate 1000 bytes of space if you only expect to use 100 bytes of the buffer.

   - If the buffer is uninitialized (there is no valid data in it prior to your application being run), buffer space can be allocated within RAM when your application is run rather than creating and storing the buffer space in the EPROM. For instance, you could include a word in your application that would allocate space for a buffer just before the space was needed. In this way you would not be increasing the dictionary size until your application has been copied out of the EPROM and executed.

   - If the buffer is uninitialized prior to your application, consider defining the buffer space to be in extended RAM (see Appendix D). This can be done by defining a constant which is a pointer to a buffer area in extended RAM. Be careful not to allow buffers to overwrite one another in extended space, as this space is free to be used, and no protection mechanism is implemented.

3. It is now necessary to change the boot-up literals which IBCL uses on start-up to determine the size and placement of the dictionary. These new values of the literals can be determined and stored in the start-up area by the following code:

```
here 1e +origin !
here 1c +origin !
latest c +origin !
```

The first line is used to store the location of the end of the dictionary. Notice that only code and data added to the system through memory location `here` will be saved. Any code or data stored outside the dictionary will not be recognized and stored.

The second line is used to determine where the fence will be placed. This is not absolutely necessary, but is highly recommended so that words defined in the newly expanded dictionary will not be inadvertently forgotten.

The third line stores the name field address of the last word defined in your extended dictionary. This tells IBCL where to begin its dictionary searches.

If you want to auto-boot, that is, to start an application on power-up, you will also need to store the code field address of the word you want to boot with in the boot-up area. This can be done by the following code:

```
hex ' name-of-power-on-word cfa 258 !
```

When the GPIB-CT is power-on, it will look at location 258 hex. If it finds the code field address of a word, it will execute that word. This word will most likely consist of an infinite loop that will continuously run an application, but it could consist of a word that will terminate.

4. You now have an exact image of the extended operating system in RAM. This memory needs to be stored in the system EPROM so that it will get loaded back into RAM at boot-up time.

   The following example shows you how to upload the IBCL system over the serial port using the `ulm` word. It assumes you are using an IBM PC or compatible and are running BASICA, but other computers and languages can be handled in a similar fashion. The program will create a DOS file that is an exact binary image of the IBCL system.

   NOTE: It will be necessary to start BASICA with the `/c:num` option in order to allocate enough space for the BASIC communication buffer to receive the entire IBCL system. `num` is the total number of bytes in the IBCL system determined in Step 2 of this procedure.

```
10 open "ibcl.bin" for output as #1
20 open "com1:9600,n,8,1" as #2
30 cmd$ = "decimal 512 here 512 - dup . ulm"
40 print #2, cmd$
50 copy$ = input$(len(cmd$),2)
60 input #2, bytes
70 for iter = 1 to bytes + 1
80 print #1,input$(1,2);
90 next iter
100 end
```

   Since you are uploading binary data, make sure that your program opens up your communications port for 8-bit data and that the configuration switches in your GPIB-CT are set accordingly. The IBCL system starts at memory location 200 hex and extends through memory location `here`. The number of bytes to upload is `here - 512`. This value is returned to the program in line 60. Lines 70 through 90 input each byte of the IBCL system and store them in the binary file.

5. To remove and copy the system EPROM and program a new custom IBCL EPROM, follow these steps:

   a. Disconnect power to the GPIB-CT and disconnect any cables that may be connected to the GPIB-CT.

b. Unscrew the two screws on the opposite sides of the rear panel.

   NOTE: Before attempting to change the system EPROM, remember that the system's EPROM, as well as most of the system's circuitry, uses CMOS technology and can be damaged by static electricity. Avoid touching the legs of components, and take any necessary CMOS handling precautions before opening the unit.

c. Remove the rear panel bezel by pulling it straight away from the rest of the unit. The board should slide out the back of the enclosure.

d. Locate and remove the system EPROM (U19). This can be done with an IC extractor tool, or by carefully prying up on each end of the EPROM with a flat head screwdriver until it pops out of the socket. Be careful not to bend any of the EPROM's legs while removing it.

e. Place the EPROM into your EPROM programmer and read its contents into a buffer file. You must read at least from 0 to 2CFF hex, but you may read all of the contents.

f. Load the binary file `ibcl.bin` (created in Step 4 of this procedure) into the EPROM programmer's buffer starting at address 2D00 hex.

g. Place the blank EPROM in the programmer and program the EPROM from address 0 to 7FFF hex.

6. Carefully insert the newly programmed EPROM back into the GPIB-CT EPROM socket, insuring that pin 1 of the EPROM is aligned with pin 1 of the socket. Also, make sure that all the EPROM legs are firmly inserted into the socket and that none are bent underneath the EPROM.

7. Close the unit and reinsert the rear panel screws removed in Part b. of Step 5.

8. Reconnect the power cord and power on the unit. Your new extended dictionary has now become a permanent part of IBCL.

Appendix D
Using Extended Memory

This appendix describes the extended memory of the GPIB-CT, and gives guidelines for its use with IBCL.

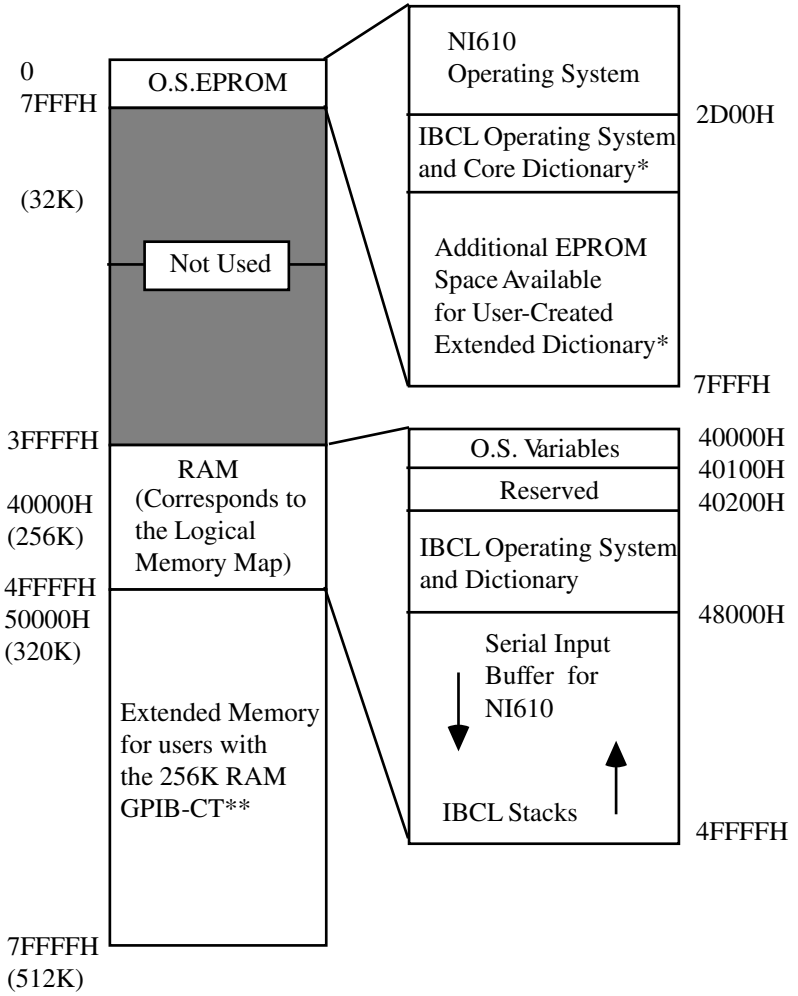
About Extended Memory

The baseline GPIB-CT comes with 64K bytes of dynamic RAM. 32K bytes are used to store the IBCL system. The remaining 32K bytes are used by the GPIB-CT default operating system as a serial input buffer, and by the IBCL operating system as stack space and free dictionary area.

If the GPIB-CT was ordered with 256K bytes of RAM, an additional 192K bytes of memory is available for use by the IBCL system. This memory is referred to as extended memory, and can be used to store data and compiled IBCL code. Extended memory is for storage purposes only–IBCL cannot run outside of its 64K bytes logical address range.

Extended memory lies hidden from IBCL and is only accessible by reprogramming the onboard Memory Management Unit (MMU). The actual programming of the MMU is rather complex, and is taken care of automatically by the extended access words. The MMU is used to form the upper address lines to map the logical 64K bytes address range of the GPIB-CT into the 512K bytes physical address space. The physical memory map of the GPIB-CT is shown in Figure D-1.

Notice that although the IBCL operating system appears to be operating in memory ranging from location 0 to FFFFH, it is actually operating from physical memory location 40000H to 4FFFFH. The MMU is loaded at power-on with the values needed to form this offset. Extended RAM space lies from physical locations 50000H to 7FFFFH.



* Copied into RAM at start-up.

** This space is useful for data storage only.

Figure D-1. Physical Memory Map

There are four IBCL words that are used to move data between extended memory and the data stack. These words are l@ (long at), lc@ (long character at), l! (long store), and lc! (long character store). These words program the MMU to allow access to any physical address space within the system. They also restore the MMU to its default condition after the memory access is complete, so that normal operation may continue.

All extended access words use a double number on the top of the stack to represent the physical address of the memory address to be accessed. For example, to print the byte at physical memory location 65848H, you would enter this line:

```
hex 6.5848 lc@ .<CR>
```

The period between the 6 and the 5 forms a double number of the address on the stack. Notice that only addresses through location 7FFFFH are supported. If a larger double number is supplied the unused upper bits of the specified address will be truncated.

Although you could write to and retrieve any physical memory location using these long words, it is suggested that you use @, c@, !, and c! to access memory in the physical range of the IBCL system (40000H to 4FFFFH). These words are slightly faster, as they do not require the reprogramming of the MMU.

You can also use extended memory as a buffer area for GPIB read and write operations. Both of these operations use the on-board DMA controller, which bypasses the MMU. This allows you to directly specify a starting address for a DMA transfer anywhere in physical memory.

The upper address register of the DMA controller supplies the upper four address lines during a DMA transfer. This is why it is only necessary to specify a 16-bit buffer address during GPIB read and write operations. Normally this register is programmed at power-on by the operating system to always DMA data within the IBCL address range (40000H to 4FFFFH).

The DMA upper address register can be changed at any time to allow DMA transfers to take place within the extended memory space. This register can be changed by writing the new value of the upper 4 address bits to I/O port address 2AH.

For example, to read 8 bytes from GPIB device 3 to an extended memory buffer starting at physical address 58000H, enter these lines:

```
5 2a p!          ( change DMA address range to 50000H thru 5FFFFH )
3 8000 8 rd      ( read 8 bytes from device 3 into offset 8000 )
4 2a p!          ( restore DMA address range to IBCL space )
```

Notice that the last line restores the DMA controller upper address register to its original default condition. This insures that any future DMA operations will be performed in IBCL space. This step can be omitted if your next DMA operation is scheduled to use the same page in extended memory. The DMA upper address register as well as all data stored in extended memory remain the same until the unit is powered-down or these values are overwritten, even if you return to the GPIB-CT default operating system and later come back to IBCL.

To print the 8 bytes received from GPIB device 3 in the previous example, you can use the lc@ word as shown here:

```
: read_extended 8 0 do 5.8000 i s->d d+ lc@ . loop ;
```

Appendix E
Other Useful IBCL Words

This appendix contains descriptions of IBCL words that are application-specific. The word description includes the purpose of the word, the kind of parameters required for execution, and where the code expects any incoming parameters. The fully commented programming examples that follow the descriptions create dictionary words. Use these programming examples to add these words to the IBCL dictionary.

dump

dump takes an address followed by a byte count as its arguments from the stack and displays the bytes in memory locations beginning with the memory address on the stack. Values are generated in hex and printable ASCII characters are also generated. Characters that are not printable (0 to 19 hex and 7F to FF hex) are displayed as a period. The byte count that is the second argument is rounded up to an even 10 hex.

Programming Example

```
: dump                          ( Memory dump)
    base @ rot rot              ( Store the current base)
                                ( at bottom of stack)
    0 do                        ( Loop from 0 to the count)
                                ( specified)
        cr dup hex              ( Duplicate the address)
                                ( and change output base)
                                ( to hex)
        0                       ( Zero fill upper 16 bits)
                                ( of double word)
        <# # # # # # > type
                                ( Convert address and)
                                ( display it)
        ." .."                  ( Separate address from)
                                ( contents with this)
                                ( string.)
        10 0 do                 ( Go from address through)
                                ( 10 addresses)
```

```
                dup c@ dup          ( Get the byte at the)
                                    ( address)
                0                   ( Zero fill upper 16 bits)
                                    ( of double word)
                <# 20 hold # # #> type
                                    ( Display ASCII code)
                                    ( of byte)
                r> r> rot >r >r >r
                                    ( Put copy of byte onto)
                                    ( return stack)
                1+                  ( Increment address)
            loop
            10 0 do
                r> r> r> rot rot >r >r
                                    ( Bring the bytes from the)
                                    ( return stack.)
            loop
            5 spaces                ( Insert 5 spaces into)
                                    ( output string)
            10 0 do
                dup                 ( Duplicate byte)
                20 <                ( Check if less than 20)
                                    ( hex-unprintable)
                if
                    drop 2e         ( If unprintable, replace)
                                    ( with a period)
                then
                dup                 ( Duplicate byte)
                7e >                ( Check if greater than 7e)
                                    ( hex-unprintable)
                if
                    drop 2e         ( If unprintable, replace)
                                    ( with a period)
                then
                emit                ( Display the byte)
            loop
        10                          ( Increment address by 10)
                                    ( hex)
        +loop
        drop cr                     ( Drop what would have)
                                    ( been the address of the)
                                    ( next row of the dump)
        base !                      ( Restore base)
    ;
```

ud.

ud. removes the top double length number from the stack and displays it as unsigned in the current base.

Programming Example

```
: ud.                       ( Unsigned double print)
     <# #s #> type          ( Convert to a string and)
                            ( type it out)
     space                  ( Insert a space in output)
                            ( string)
;
```

depth

depth counts the number of words on the stack (prior to execution of the word) and leaves the count as the top value on the stack.

Programming Example

```
: depth                     ( Count the depth of the)
                            ( data stack)
     s0 @                   ( Put stack's origin)
                            ( address on stack)
     sp@                    ( Put current stack)
                            ( address on stack)
     -                      ( Get {depth + 2}*2)
     2 -                    ( Compensate for length)
                            ( being on stack)
     2 /                    ( Depth is distance)
                            ( between address / 2)
;
```

not

not performs the logical NOT of the value on the stack. This is an example of redefining an existing IBCL word with a more meaningful name (the IBCL word 0= performs the logical NOT).

Programming Example

```
: not                    ( Logical NOT function)
    0=                   ( 0= provides the)
                         ( logical NOT operation in)
                         ( IBCL)
;
```

0>

0> checks if the word on the top of the stack has a value greater than zero. If it does, a TRUE flag is left on the stack. If it is not, a FALSE is left on the stack.

Programming Example

```
: 0>                     ( Zero greater)
    -dup 0=              ( Duplicate if not zero)
                         ( the number to be checked)
                         ( then compare it to zero)
    if                   ( If number tested was)
                         ( zero, put false flag on)
                         ( the stack)
        0
    else
        0<               ( Determine whether number)
                         ( is < 0)
        not              ( Get the logical NOT of)
                         ( the flag)
    then
;
```

binary

binary sets the I/O base to binary, in the same way that decimal and hex set the I/O base to decimal and hexadecimal respectively.

Programming Example

```
: binary                    ( Set I/O base to 2)
     [ decimal ] 2 base !
                            ( Store a 2 in the)
                            ( base user variable)
;
```

octal

octal sets the I/O base to binary, in the same way that decimal and hex set the I/O base to decimal and hexadecimal respectively.

Programming Example

```
: octal                     ( Set I/O base to 8)
     [ decimal ] 8 base !
                            ( Store an 8 in the base)
                            ( user variable)
;
```

msa

msa takes its arguments on the stack a primary address with a secondary address on top. It formulates the single word necessary to use with any GPIB word requiring a device address as a parameter. Use the word like this:

```
23 ( primary) 67 ( secondary) msa caddr
```

or like this:

```
23 ( primary) 67 ( secondary) msa clr
```

Programming Example

```
: msa                          ( Make secondary address)
      1f and                   ( AND out unnecessary)
                               ( bits in sec. addr)
                               ( Only lower 5 bits make)
                               ( up an address)
      100 *                    ( Shift the secondary)
                               ( address into the low 5)
                               ( bits of the high order)
                               ( byte)
      8000 or                  ( Set the upper bit to)
                               ( indicate the presence of)
                               ( a secondary address)
      or                       ( Put in the primary)
                               ( address)
;
```

.s

.s displays the contents of the stack non-destructively. In this example, the program prints the contents as unsigned words. If you want the contents to be displayed differently, change cr i @ u. accordingly (for example, cr i @ . for a signed stack display).

Programming Example

```
: .s                           ( Show the stack)
      s0 @
      sp@ - 2 -                ( Get number of words on)
                               ( the stack)
      -dup 0=                  ( Duplicate the length if)
                               ( not zero and check if)
                               ( equal to zero)
      if
            cr ." EMPTY STACK"
                               ( Tell user no data on the)
                               ( stack)
      else
            2+                 ( Increment word count)
                               ( by 2)
```

```
            sp@ +               ( Get do loop limit-this)
                                ( is the highest stack)
                                ( address plus 2)
            sp@ 2+ do           ( Get beginning address)
                cr
                i               ( Get current address)
                @               ( Get what's at that)
                                ( address)
                u.              ( Display the value)
                2
            +loop               ( Loop again-loop index)
                                ( now at next stack word)
        then
;
```

pick

pick takes as its parameter the top word on the stack. This word is treated as an index into the stack. pick copies the value at that index onto the top of the stack. If the stack is not deep enough to have a corresponding value on it, no error message is printed, nothing is put on the stack, and the word aborts, causing the stack to be reset.

Programming Example

```
: pick                          ( Pick the number from the)
                                ( stack and duplicate it)
                                ( on the top of the stack)
    dup                         ( Duplicate top number -)
                                ( index into stack)
    depth 2 -                   ( Get depth of stack - 2)
                                ( because of the extra)
                                ( count on top)
    >                           ( Check if the requested)
                                ( element exists on the)
                                ( stack)
    if
        drop                    ( Remove duplicate index)
                                ( into stack)
        abort                   ( Stop executing)
    else
        2 *                     ( Convert byte index to)
                                ( word index)
```

```
            sp@                 ( Get current stack)
                                ( location)
            +                   ( Get address of desired)
                                ( stack location)
            @                   ( Get value stored there)
        then
;
```

roll

roll takes as its parameter the top word on the stack. This word is treated as an index into the stack. roll puts the value at that index onto the top of the stack, removing it from its present location in the stack. If the stack is not deep enough to have a corresponding value on it, no error message is printed, nothing is put on the stack, and the word aborts, causing the stack to be reset.

Programming Example

```
: roll                          ( Put a value from the)
                                ( stack on top of stack)
    dup                         ( Duplicate stack index)
                                ( of element wanted)
    depth 1 -                   ( Get true depth-disregard)
                                ( the extra index)
    >                           ( Check if stack is deep)
                                ( enough)
    if
        drop                    ( Remove the extra index)
        abort                   ( Stop executing the word)
    else
        dup dup >r >r           ( Duplicate index and)
                                ( store on return stack)
                                ( twice)
        pick                    ( Get a copy of the)
                                ( desired value)
        r>                      ( Get one of the indexes)
                                ( from the return stack)
                                ( do loop limit)
        0 do
            r> r>               ( Bring the do loop cnt)
                                ( and limit from the)
                                ( return stack)
```

```
                rot             ( Put the value that was)
                                ( top value on stack)
                                ( before r> r> onto top)
                r>              ( Bring over last index)
                                ( from return stack)
                swap            ( Swap it with the value)
                                ( that was rotated up from)
                                ( the data stack)
                >r >r >r >r     ( Put all values on return)
                                ( stack-value from stack,)
                                ( index, do limit, do)
                                ( count)
            loop                ( Do this until we get to)
                                ( where the value is on)
                                ( the stack that we rolled)
                                ( up)
            drop                ( Remove that value)
            r> 0 do             ( Get the index-do loop)
                                ( limit)
                r> r> r>        ( Bring over do limit, do)
                                ( count and stack value)
                rot             ( Put the do limit on top)
                rot             ( Put the do count on top)
                >r >r           ( Put these two back on)
                                ( the return stack-leaving)
                                ( the data value on the)
                                ( data stack)
            loop
        then
;
```

decom

decom decompiles a word which is composed of other IBCL words, such as a word defined in a colon definition. It goes through the definition of the word specified and prints each component word's name. Use this word in this form:

decom <dname>

where dname is a defined word.

There are a few limitations to this word:

- It cannot decompile a machine code primitive
- If a word has a ." or " followed by ASCII data, this program continues trying to decompile the ASCII data. This might cause IBCL to crash. If IBCL does not crash, the output of this type of operation will look strange.

Even with these restrictions, it is a useful word if you have forgotten what you have previously entered as a word's definition.

Programming Example

```
: decom                     ( Decompile an IBCL word)
    [compile] ' cr          ( Get the pfa of the)
                            ( requested word)
    dup cfa @               ( Get the address of the)
                            ( first word composing its)
                            ( definition)
    [ ' task cfa @ ] literal
                            ( Compile into the)
                            ( definition the value of)
                            ( a known non-machine)
                            ( coded IBCL word)
    =                       ( Check to see if the word)
                            ( asked for is a machine)
                            ( coded word, variable,)
                            ( constant, or a colon)
                            ( definition)
    if
```



```
            begin               ( The word is composed of)
                                ( other IBCL words-NOT)
                                ( machine coded)
            dup 2+ swap         ( Get address of next)
                                ( component word and store)
                                ( at bottom of the stack)
            @                   ( Get the cfa of component)
                                ( word)
            dup 2+              ( Convert to pfa of)
                                ( component word)
            nfa                 ( Convert to nfa of)
                                ( component word)
            id. cr              ( Display the name of the)
                                ( word whose nfa is on the)
                                ( stack)
            ' ;s cfa            ( Get the cfa of the word)
                                ( that has to complete a)
                                ( colon definition)
            = until             ( Keep decompiling until)
                                ( this word is reached)
            drop                ( Remove the address which)
                                ( was the next component)
                                ( word)
                                ( in the requested word's)
                                ( pfa list)
        else
            cr. "Machine Code Primitive"
                                ( Display message)
        then
    ;
```

cls

cls clears the screen on many terminals by emitting an ASCII 1A hex (decimal 26), or <CTRL-z>, which clears many terminal screens.

Programming Example

```
: cls                          ( Clear the terminal)
                               ( screen)
      1a emit                  ( Ctrl-z character)
;
```

Redefining the Basic IBCL Mathematical Operators to Use Infix Notation

These five examples show how you can redefine the basic IBCL mathematical operators to use infix notation. These examples are very simple. They work on only 2 operands, must have an = entered after each expression (for example, 3 + 4 = + 5 = . instead of 3 + 4 + 5 = .), and execute from left to right. Precedence rules are obeyed only if you enter the expression in the correct order.

Programming Examples

Redefining =

```
: =                            ( Redefinition of IBCL =)
      rot execute              ( Leaves on the stack the)
                               ( result from the)
                               ( operation whose cfa is)
                               ( under the 2 operands)
;
```

Redefining +

```
: +                            ( Redefinition of IBCL +)
      [ ' + cfa ] literal swap
                               ( Puts the cfa of the IBCL)
                               ( + on the stack and swaps)
                               ( it with the first)
                               ( operand)
;
```

Redefining -

```
: -                          ( Redefinition of IBCL -)
     [ ' - cfa ] literal swap
                             ( Puts the cfa of the IBCL)
                             ( - on the stack and swaps)
                             ( it with the first)
                             ( operand)
;
```

Redefining *

```
: *                          ( Redefinition of IBCL *)
     [ ' * cfa ] literal swap
                             ( Puts the cfa of the IBCL)
                             ( * on the stack and swaps)
                             ( it with the first)
                             ( operand)
;
```

Redefining /

```
: /                          ( Redefinition of IBCL /)
     [ ' / cfa ] literal swap
                             ( Puts the cfa of the IBCL)
                             ( / on the stack and swaps)
                             ( it with the first)
                             ( operand)
;
```

Appendix F
Glossary of IBCL Functions

This appendix contains a list of commonly used IBCL words and a description of each. The definitions are divided into two parts–*GPIB Glossary*, which contains GPIB-related IBCL functions, and *Standard Glossary*, which contains all other IBCL words.

Glossary Conventions

Table F-1 contains the conventions that are used throughout this glossary.

Table F-1. Glossary Conventions

Abbreviation	Meaning
addr	a value representing a memory address
b	a value representing an 8-bit byte
c	a value representing an 7-bit ASCII code
d	a 32-bit signed double number
f	a Boolean value (0=FALSE, not 0 = TRUE)
gaddr	a value representing a GPIB device address
n	a 16-bit signed integer
ud	a 32-bit unsigned double number
un	a 16-bit unsigned integer

GPIB Glossary

Table F-2 contains a listing of the IBCL GPIB extensions. For a detailed description of each word in the GPIB glossary, refer to Chapter 3, *GPIB Extensions*.

Table F-2. GPIB Glossary

Word	Stack
brd	addr un –>
bwrt	addr un –>
cac	f –>

(continues)

Table F-2. GPIB Glossary (continued)

Word	Stack
caddr	gaddr –>
clr	gaddr –>
cmd	addr un –>
eos	un –>
eot	f –>
gts	f –>
ist	f –>
loc	gaddr
onl	f –>
pct	gaddr –>
ppc	gaddr b –>
rd	gaddr addr un –>
rpp	–> b
rsc	f –>
rsp	gaddr –> n
rsv	b –>
sic	
sre	f –>
stat	–> un n
tmo	b –>
trg	gaddr –>
wait	un –>
wrt	gaddr addr un –>

Standard Glossary

All IBCL words other than the IBCL GPIB extensions are listed here. Because IBCL words can contain non-alphanumeric characters, the words in this glossary are arranged in the order that the characters appear in the ASCII chart of Appendix A, *Multiline Interface Messages*:

```
!"#$%&'()*+,-./0123456789:;<=
>?@abcdefghijklmnopqrstuvwxyz[\]
```

Word: !
Stack: n addr –>
Description: Store value n at address addr. Called "store."

Word: `!csp`
Description: Save the stack position in `csp`. Used as part of the compiler security.

Word: `"`
Description: Used in the form `"cccc"` where `cccc` is data to be written over the GPIB or to be sent as commands over the GPIB. `"` compiles an in-line string `cccc` (delimited by the trailing `"`) with an execution procedure that places the `addr` and `n` on the stack that is required for `wrt` or `cmd`. If executed outside a definition, `"` immediately places the `addr` and `n` on the stack.

Word: `#`
Stack: `d1 –> d2`
Description: Generate from a double number `d1` the next ASCII character which is placed in an output string. Result, `d2`, is the quotient after division by `base`, and is maintained for further processing. Used between `<#` and `#>`. See `#s`.

Word: `#>`
Stack: `d –> addr count`
Description: Terminates numeric output conversion by dropping `d`, leaving the text address and character count suitable for `type`.

Word: `#s`
Stack: `d1 –> d2`
Description: Generates ASCII text in the text output buffer, by the use of the `#`, until a zero double number `d2` results. Used between `<#` and `#>`.

Word: '
Stack: –>
Description: Used in the form:

```
' nnnn
```

Leaves the parameter field address of dictionary word `nnnn`. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of `context` and `current`, an appropriate error message is given. Called "tick."

Word: (
Description: Used in the form:

```
(  cccc )
```

Ignore a comment that will be delimited by a close parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

Word: (.")
Description: The run-time procedure, compiled by `."`, which transmits the subsequent in-line text to the serial port. See `."`.

Word: (+loop)
Stack: n –>
Description: The run-time procedure compiled by `+loop`, which increments the loop index by `n` and tests for loop completion. See `+loop`.

Word: (abort)
Description: Executes after an error when `warning` is -1. This word normally executes `abort`, but may be changed (with care) to a user's alternative procedure.

Word: (do)
Description: The run-time procedure compiled by `do` which moves the loop control parameters to the return stack. See `do`.

Word: `(dq)`
Stack: `—>`
Description: The run-time procedure compiled by `"` which puts the `addr` and `n` on the stack as required for `bwrt`, `wrt` and `cmd`. See `"`.

Word: `(find)`
Stack:
```
addr1  addr2  —> pfa b t f    (ok)
addr1  addr2  —>   f         (bad)
```
Description: Searches the dictionary starting at the name field address `addr2`, matching to the text at `addr1`. Returns parameter field address, length byte of name field, and boolean TRUE for a good match. If no match is found, only a boolean FALSE is left.

Word: `(loop)`
Description: The run-time procedure compiled by `loop` which increments the loop index and tests for loop completion. See `loop`.

Word: `(number)`
Stack: `d1 addr1 —> d2 addr2`
Description: Convert the ASCII text beginning at `addr1+1` with regard to `base`. The new value is accumulated into double number `d1`, being left as `d2`. `addr2` is the address of the first unconvertible digit. Used by `number`.

Word: `*`
Stack: `n1  n2  —> n3`
Description: Leave the signed product, `n3`, of two signed numbers.

Word: `*/`
Stack: `n1  n2  n3  —>  n4`
Description: Leave the ratio `n4 = n1*n2/n3` where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence:

```
n1  n2  *  n3  /
```

Word: `*/mod`
Stack: `n1 n2 n3 –> n4 n5`
Description: Leave the quotient `n5` and remainder `n4` of the operation `n1*n2/n3`. A 31 bit intermediate is used as for `*/`.

Word: `+`
Stack: `n1 n2 –> n3`
Description: Leave the sum `n3` of `n1+n2`.

Word: `+!`
Stack: `n addr –>`
Description: Add `n` to the value at the address. Called "plus-store."

Word: `+-`
Stack: `n1 n2 –> n3`
Description: Leave `n3`, with magnitude of `n1` and sign of `n1*n2`.

Word: `+loop`
Stack: `n1 –> ( run )`
`addr n2 –> (compile)`
Description: Used in a colon-definition in the form:

```
do ... n1 +loop
```

At run-time, `+loop` selectively controls branching back to the corresponding `do` based on `n1`, the loop index and the loop limit. The signed increment `n1` is added to the index and the total compared to the limit. The branch back to `do` occurs until the new index is equal to or greater than the limit (`n1>0`), or until the new index is equal to or less than the limit (`n1<0`). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, `+loop` compiles the run-time word `(+loop)` and the branch offset computed from `here` to the address left on the stack by `do`. `n2` is used for compile time error-checking.

Word: `+origin`
Stack: `n –> addr`
Description: Leave the memory address with a relative offset of `n` to the origin parameter area. `n` is the byte number; that is, to access the fourth word in the origin area, you would specify `n` to be 6. This definition is used to access or modify the boot-up parameters at the origin area.

Word: `,`
Stack: `n –>`
Description: Store `n` into the next available dictionary memory cell, advancing the dictionary pointer. Called "comma."

Word: `-`
Stack: `n1 n2 –> n3`
Description: Leave the difference of `n1-n2` as `n3`.

Word: `-dup`
Stack:
```
n1 –> n1        (if zero)
n1 –> n1 n1  (non-zero)
```
Description: Reproduce `n1` only if it is non-zero. This is usually used to copy a value just before `if`, to eliminate the need for an `else` to drop it.

Word: `-find`
Stack:
```
–> pfa   b   f        (found)
–>       f          (not found)
```
Description: Accepts the next text word (delimited by blanks) in the input stream to `here`, and searches the `context` and then `current` vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean TRUE is left. Otherwise, only a boolean FALSE is left.

Word: `-trailing`
Stack: `addr n1 –> addr n2`
Description: Adjusts the character count `n1` of a text string beginning at address `addr` to suppress the output of trailing blanks. For example, the characters at `addr+n1` to `addr+n2` are blanks.

Word: `.`
Stack: `n –>`
Description: Print a number from a signed 16-bit two's complement value, converted according to the numeric `base`. A trailing blank follows. Called "dot."

Word: `."`
Description: Used in the form:

```
."  cccc"
```

Compiles an in-line string `cccc` (delimited by the trailing `"`) with an execution procedure to transmit the text to the serial port. If executed outside a definition, `."` will immediately print the text until the final `"`. See `(."")`.

Word: `.r`
Stack: `n1 n2 –>`
Description: Print the number `n1` right aligned in a field whose width is `n2`. No following blank is printed.

Word: `/`
Stack: `n1 n2 –> n3`
Description: Leave the signed quotient of `n1/n2` as `n3`.

Word: `/mod`
Stack: `n1 n2 –> n3 n4`
Description: Leave the remainder `n3` and signed quotient `n4` of `n1/n2`. The remainder has the sign of the dividend.

Word: `0 1 2 3`
Stack: `–> n`
Description: These small numbers are used so often that it is helpful to define them by name in the dictionary as constants.

Word: `0<`
Stack: `n –> f`
Description: Leave a TRUE flag if the number is less than zero (negative), otherwise leave a FALSE flag.

Word: `0=`
Stack: `n –> f`
Description: Leave a TRUE flag if the number is equal to zero, otherwise leave a FALSE flag.

Word: `0branch`
Stack: `f –>`
Description: The run-time procedure to conditionally branch. If `f` is FALSE (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by `if`, `until`, and `while`.

Word: `1+`
Stack: `n1 –> n2`
Description: Increment `n1` by 1.

Word: `2!`
Stack: `d addr –>`
Description: Store the double number `d` beginning at `addr`.

Word: `2+`
Stack: `n1 –> n2`
Description: Increment `n1` by 2.

Word: `2@`
Stack: `addr –> d`
Description: Leave the 32-bit contents of address `addr` on the stack.

Word: `2dup`
Stack: `d –> d d`
Description: Duplicate the top double number on the stack.

Word: :
Description: Used in the form called a colon-definition:

```
: cccc ... ;
```

Creates a dictionary entry defining `cccc` as equivalent to the following sequence of IBCL word definitions `'...'` until the next `';'`.

The compiling process is done by the text interpreter as long as `state` is non-zero. Other details are that the `context` vocabulary is set to the `current` vocabulary and that words with the precedence bit set (see `immediate`) are executed rather than being compiled.

Word: `;`
Description: Terminate a colon-definition and stop further compilation. Compiles the run-time word `;s`.

Word: `;s`
Description: `;s` is the run-time word compiled at the end of a colon-definition which returns execution to the calling procedure.

Word: `<`
Stack: `n1 n2 –> f`
Description: Leave a TRUE flag if `n1` is less than `n2`; otherwise leave a FALSE flag.

Word: `<#`
Description: Setup for pictured numeric output formatting using the words:

```
<#  #  #s  sign  #>
```

The conversion is done on a double number producing text at `pad`.

Word: <builds
Description: Used within a colon-definition:

```
: cccc <builds ...
       does>  ...  ;
```

Each time cccc is executed, <builds defines a new word with a high-level execution procedure.

Executing cccc in the form:

```
cccc nnnn
```

uses <builds to create a dictionary entry for nnnn with a call to the does> part for nnnn.

When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after does> in cccc.

Word: =
Stack: n1 n2 –> f
Description: Leave a TRUE flag if n1=n2; otherwise leave a FALSE flag.

Word: >
Stack: n1 n2 –> f
Description: Leave a TRUE flag if n1 is greater than n2; otherwise leave a FALSE flag.

Word: >r
Stack: n –>
Description: Removes a number from the computation stack and places it as the most accessible on the return stack. Use should be balanced with r> in the same definition. Called "to-R." Also see r> and r.

Word: ?
Stack: addr –>
Description: Print the value contained at the address in free format according to the current base.

Word: `?comp`
Description: Issue error message if not compiling.

Word: `?csp`
Description: Issue error message if stack position differs from value saved in `csp`.

Word: `?error`
Stack: `f n –>`
Description: Issue an error message number `n`, if the boolean flag is TRUE.

Word: `?exec`
Description: Issue an error message if not executing.

Word: `?pairs`
Stack: `n1 n2 –>`
Description: Issue an error message if `n1` does not equal `n2`. The message indicates that compiled conditionals do not match.

Word: `?stack`
Description: Issue an error message if the stack is out of bounds.

Word: `?terminal`
Stack: `–> f`
Description: Perform a test of the serial port to see if a character has been sent. A TRUE flag indicates that a character has been sent.

Word: `@`
Stack: `addr –> n`
Description: Leave the 16-bit contents of address `addr` on the stack.

Word: `abort`
Description: Clear both the computation stack and the return stack. Return control to the operators terminal.

Word: `abs`
Stack: `n –> u`
Description: Leave the absolute value of `n` as `u`.

Word: `again`
Stack: `addr n –> (compiling)`
Description: Used in a colon-definition in the form:

```
begin   ...    again
```

At run-time, `again` forces execution to return to the corresponding `begin`. There is no effect on the stack. Notice that this is an infinite loop structure and execution cannot leave this loop (unless `r> drop` is executed one level below).

At compile time, `again` compiles `branch` with an offset from `here` to `addr`. `n` is used for compile-time error-checking.

Word: `allot`
Stack: `n –>`
Description: Add the signed number to the dictionary pointer `dp`. May be used to reserve dictionary space or re-origin memory.

Word: `and`
Stack: `n1 n2 –> n3`
Description: Leave the bitwise logical AND of `n1` and `n2` as `n3`.

Word: `back`
Stack: `addr –>`
Description: Calculate the backward branch offset from `here` to `addr` and compile into the next available dictionary memory address.

Word: `base`
Stack: `–> addr`
Description: A user variable containing the current number base used for input and output numerical conversion.

Word: `begin`
Stack: `–> addr n (compiling)`
Description: Occurs in a colon-definition in the form:

```
begin   ...   until
begin   ...   again
begin   ...   while   ...   repeat
```

At run-time, `begin` marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding `until`, `again`, or `repeat`. When executing `until`, a return to `begin` will occur if the top of the stack is FALSE. For `again` and `repeat`, a return to `begin` always occurs.

At compile time `begin` leaves its return address and `n` for compiler error-checking.

Word: `bl`
Stack: `–> c`
Description: A constant that leaves the ASCII value for "blank."

Word: `blanks`
Stack: `addr n –>`
Description: Fill an area of memory beginning at `addr` with blanks for `n` bytes.

Word: `branch`
Description: The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer to branch ahead or back. `branch` is compiled by `else`, `again`, `repeat`.

Word: `bye`
Description: Exit IBCL to the GPIB-CT default operating system.

Word: `c!`
Stack: `b addr –>`
Description: Store 8 bits at `addr`.

Word: `c,`
Stack: `b –>`
Description: Store 8 bits of `b` into the next available dictionary byte, advancing the dictionary pointer.

Word: `c/l`
Stack: `–> n`
Description: A constant leaving the number of characters per source code screen line.

Word: `c@`
Stack: `addr –> b`
Description: Leave the 8-bit contents of `addr`.

Word: `cfa`
Stack: `addr1 –> addr2`
Description: Convert the parameter field address `addr1` of a word to its code field address `addr2`.

Word: `cmove`
Stack: `addr1 addr2 n –>`
Description: Move `n` bytes beginning at address `addr1` to address `addr2`. The contents of `addr1` is moved first and proceeds toward high memory.

Word: `cold`
Description: The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via `abort`. May be called to remove application programs and restart.

Word: `compile`
Description: When the word containing `compile` executes, the execution address of the word following `compile` is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

Word: constant
Stack: n –>
Description: A defining word used in the form:

```
n  constant  cccc
```

to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n onto the stack.

Word: context
Stack: –> addr
Description: A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

Word: count
Stack: addr1 –> addr2 n
Description: Leave the address addr2 and byte count n of text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically count is followed by type.

Word: cr
Description: Transmit a carriage return and linefeed to the serial port.

Word: create
Description: A defining word used in the form:

```
create  cccc
```

to create a dictionary header for an IBCL definition. The code field contains the address of the word's parameter field. The new word is created in the current vocabulary.

Word: csp
Stack: –> addr
Description: A user variable temporarily storing the stack pointer position, for compilation error-checking.

Word: current
Stack: –> addr
Description: A user variable containing a pointer to the vocabulary within which new dictionary words will be entered.

Word: d+
Stack: d1 d2 –> d3
Description: Leave the double number sum d3 of two double numbers d1 + d2.

Word: d+-
Stack: d1 n –> d2
Description: Leave d2, with magnitude of d1 and sign of n*d1.

Word: d.
Stack: d –>
Description: Print a signed double number from a 32-bit two's complement value. The high-order 16 bits are most accessible on the stack. Conversion is performed according to the current base. A blank follows. Called "D-dot."

Word: d.r
Stack: d n –>
Description: Print a signed double number d right aligned in a field n characters wide.

Word: dabs
Stack: d –> ud
Description: Leave the absolute value ud of a double number d.

Word: decimal
Description: Set the numeric conversion base for decimal input-output.

Word: definitions
Description: Used in the form:

```
cccc  definitions
```

Set the current vocabulary to the context vocabulary. In the example, executing vocabulary name cccc made it the context vocabulary and executing definitions made both specify vocabulary cccc.

Word: digit
Stack:
```
c  n1  –> n2  f       (ok)
c  n1  –> f          (bad)
```
Description: Converts the ASCII character c (using base n1) to its binary equivalent n2, accompanied by a TRUE flag. If the conversion is invalid, leaves only a FALSE flag.

Word: dliteral
Stack:
```
d  –> d      (executing)
d –>         (compiling)
```
Description: If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

Word: dlm
Stack: addr un –>
Description: dlm is a binary input word. dlm downloads from the host directly to the GPIB-CT memory un bytes starting at addr.

Word: dminus
Stack: d1 –> d2
Description: Convert d1 to its double number two's complement.

Word: do
Stack:
```
n1   n2  ---    (execute)
addr  n  ---    (compile)
```
Description: Occurs in a colon-definition in the form:

```
do  ...  loop
do  ...  +loop
```

At run-time, do begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. do removes these from the stack. Upon reaching loop, the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after do; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop i will copy the current value of the index to the stack. Also see i, loop, +loop, leave.

When compiling within the colon-definition, do compiles (do), which leaves the following address addr and n for later error-checking.

Word: does>
Description: A word which defines the run-time action within a high-level defining word. does> alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following does>. Used in combination with <builds. When the does> part executes, it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include multi-dimensional arrays, and compiler generation.

Word: dp
Stack: –> addr
Description: A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by here and altered by allot.

Word: dpl
Stack: –> addr
Description: A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used to hold output column location of a decimal point in user generated formatting. The default value on single number input is -1.

Word: drop
Stack: n –>
Description: Drop the top number from the stack.

Word: dup
Stack: n –> n n
Description: Duplicate the top value on the stack.

Word: else
Stack: addr1 n1 –> addr2 n2 (compiling)
Description: Occurs within a colon-definition in the form:

```
if ... else ... endif
```

At run-time, `else` executes after the TRUE part following `if`. `else` forces execution to skip over the following FALSE part and resumes execution after the `endif`. It has no stack effect.

At compile-time, `else` compiles `branch` reserving a branch offset and leaves the address `addr2` and `n2` for error-checking. `else` also resolves the pending forward branch from `if` by calculating the offset from `addr1` to `here` and storing at `addr1`.

Word: emit
Stack: c –>
Description: Transmit the ASCII character `c` to the serial port. `out` is incremented for each character output.

Word: `enclose`
Stack: `addr1 c –> addr1 n1 n2 n3`
Description: The text scanning primitive used by `word`. From the text address `addr1` and an ASCII delimiting character `c`, is determined the byte offset to the first non-delimiter character `n1`, the offset to the first delimiter after the text `n2`, and the offset to the first character not included. This procedure will not process past an ASCII null, treating it as an unconditional delimiter.

Word: `end`
Description: This is a duplicate definition for `until`.

Word: `endif`
Stack: `addr n –> (compile)`
Description: Occurs in a colon-definition in the form:

```
if ... endif
if ... else ... endif
```

At run-time, `endif` serves only as the destination of a forward branch from `if` or `else`. It marks the conclusion of the conditional structure. `then` is another name for `endif`. See also `if` and `else`.

At compile-time, `endif` computes the forward branch offset from `addr` to `here` and stores it at `addr`. `n` is used for error-checking.

Word: `erase`
Stack: `addr n –>`
Description: Clear a region of memory to zero from address `addr` over `n` addresses.

Word: `error`
Stack: `n –>`
Description: Execute error notifications and restart of system. `warning` is first examined. If `warning` is a 0 or 1, `n` is printed as a message number. If `warning` is a -1, the definition `(abort)` is executed, which executes the system `abort`. The user may cautiously modify this execution by altering `(abort)`. Final action is execution of `quit`.

Word: `execute`
Stack: `addr –>`
Description: Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

Word: `expect`
Stack: `addr n –>`
Description: Transfer characters from the serial port to address `addr` until a return or the count of `n` characters have been received. One or more nulls are added at the end of the text.

Word: `fence`
Stack: `–> addr`
Description: A user variable containing an address `addr` below which `forget`ting is trapped. To forget below this point, the user must alter the contents of `fence`.

Word: `fill`
Stack: `addr n b –>`
Description: Fill memory at the address `addr` with the specified quantity, `n`, of bytes `b`.

Word: `forget`
Description: Executed in the form:

```
forget  cccc
```

Deletes definition named `cccc` from the dictionary with all entries physically following it. An error message will occur if the `current` and `context` vocabularies are not currently the same.

Word: `here`
Stack: `–> addr`
Description: Leave the address of the next available dictionary location.

Word: `hex`
Description: Set the numeric conversion base to sixteen (hexadecimal).

Word: `hld`
Stack: `–> addr`
Description: A user variable that holds the address of the latest character of text during numeric output conversion.

Word: `hold`
Stack: `c –>`
Description: Used between `<#` and `#>` to insert an ASCII character into a pictured numeric output string. For example, `2E hold` will place a decimal point.

Word: `i`
Stack: `–> n`
Description: Used within a `do-loop` to copy the loop index to the stack. Also see `r`.

Word: `ibcl`
Description: The name of the primary vocabulary. Execution makes `ibcl` the `context` vocabulary. Until additional user vocabularies are defined, new user definitions become a part of `ibcl`. `ibcl` is immediate, so it will execute during the creation of a colon-definition to select this vocabulary at compile time.

Word: `id.`
Stack: `addr –>`
Description: Print a definition's name from its name field address.

Word: `if`
Stack: `f –>` `(run-time)`
`–> addr n` `(compile)`
Description: Occurs in a colon-definition in the form:

```
if (true part) ... endif
if (true part) ... else  (false part) ... endif
```

At run-time, `if` selects execution based on a boolean flag. If `f` is TRUE (non-zero), execution continues ahead through the TRUE condition. If `f` is FALSE (zero), execution skips until just after `else` to execute the FALSE condition. After either condition, execution resumes after `endif`. `else` and its FALSE condition are optional; if missing, FALSE execution skips to just after `endif`.

At compile-time `if` compiles `0branch` and reserves space for an offset at `addr`. `addr` and `n` are used later for resolution of the offset and error-checking.

Word: `immediate`
Description: Mark the most recently created definition so that when encountered at compile time, it will be executed rather than being compiled. In other words, the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceding it with `[compile]`.

Word: `in`
Stack: `–> addr`
Description: A user variable containing the byte offset within the current input text buffer from which the next text will be accepted. `word` uses and moves the value of `in`.

Word: `interpret`
Description: The outer text interpreter which sequentially executes or compiles text from the serial port depending on `state`. If the word name cannot be found after a search of the `context` and then the `current` vocabulary, it is converted to a number according to the current base. That also failing, an error message echoing the name with a `" ?"` will be given. Text input will be taken according to the convention for `word`. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. Also see `number`.

Word: `key`
Stack: `–> c`
Description: Leave the ASCII value of the next serial character received.

Word: `l!`
Stack: `n d –>`
Description: Stores value `n` at address specified by the double `d`.

Word: `l@`
Stack: `d –> n`
Description: Leave the 16-bit contents of the memory addressed by `d` on the stack.

Word: `latest`
Stack: `–> addr`
Description: Leave the name field address of the topmost word in the `current` vocabulary.

Word: `lc!`
Stack: `b d –>`
Description: Store 8 bits of `b` at address specified by the double `d`.

Word: `lc@`
Stack: `d –> b`
Description: Leave the 8-bit contents of the memory addressed by `d` on the stack.

Word: `leave`
Description: Force termination of a `do-loop` at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until `loop` or `+loop` is encountered.

Word: `lfa`
Stack: `pfa –> lfa`
Description: Convert the parameter field address of a dictionary definition to its link field address.

Word: `limit`
Stack: `–> u`
Description: A constant leaving the address of the highest system memory available.

Word: `lit`
Stack: `–> n`
Description: Within a colon-definition, `lit` is automatically compiled before each 16-bit literal number encountered in input text. Later execution of `lit` causes the contents of the next dictionary address to be pushed to the stack.

Word: `literal`
Stack: `n –>`
Description: If compiling, then compile the stack value `n` as a 16-bit literal. This definition is immediate so that it will execute during a colon-definition. The intended use is:

```
: xxx [ calculate ] literal ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and `literal` compiles this value.

Word: loop
Stack: addr n –> (compiling)
Description: Occurs in a colon-definition in the form:

```
do  ...  loop
```

At run-time, loop selectively controls branching back to the corresponding do based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to do occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile-time, loop compiles (loop) and uses addr to calculate an offset to do. n is used for error-checking.

Word: m*
Stack: n1 n2 –> d
Description: A mixed magnitude math operation which leaves the signed double number product of two numbers.

Word: m/
Stack: d n1 –> n2 n3
Description: A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3 from a double number dividend d and divisor n1. The remainder takes its sign from the dividend.

Word: m/mod
Stack: ud1 u2 –> u3 ud4
Description: An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3 from a double dividend ud1 and single divisor u2.

Word: max
Stack: n1 n2 –> n3
Description: Leave the greater of two numbers.

Word: `message`
Stack: `n –>`
Description: Prints `MSG# n.` `n` may be positive or negative. `message` could be used to alert a user of a condition, provided the user knows what each message number represents.

Word: `min`
Stack: `n1 n2 –> n3`
Description: Leave the smaller of two numbers.

Word: `minus`
Stack: `n1 –> n2`
Description: Leave the two's complement of a number.

Word: `mod`
Stack: `n1 n2 –> n3`
Description: Leave the remainder of `n1/n2`, with the same sign as `n1`.

Word: `nfa`
Stack: `pfa –> nfa`
Description: Convert the parameter field address of a word to its name field address.

Word: `number`
Stack: `addr –> d`
Description: Convert a character string left at `addr` to a signed double number using the current numeric base. The string consists of the characters for conversions preceded by a 1-byte count of characters to convert followed by a blank (hex 20). If a decimal point is encountered in the text, its position will be given in `dp1`, with no other effect. If a numeric conversion is not possible, error message `0` will be given.

Word: `or`
Stack: `n1 n2 –> n3`
Description: Leave the bit-wise logical inclusive-OR of two 16-bit values.

Word: out
Stack: –>addr
Description: A user variable that contains a value incremented by emit. The user may alter and examine out to control display formatting.

Word: over
Stack: n1 n2 –> n1 n2 n1
Description: Duplicates the second element on the stack.

Word: p!
Stack: b addr –>
Description: p! place b into I/O addr. Pronounced "P-store."

Word: p@
Stack: addr –> b
Description: p@ reads b from I/O address addr. Pronounced "P-at."

Word: pad
Stack: –> addr
Description: Leave the address of the text output buffer, which floats at a fixed offset above here.

Word: pfa
Stack: nfa –> pfa
Description: Convert the name field address of a word to its parameter field address.

Word: query
Description: Input 80 characters of text (or until a return) from the serial port. Text is positioned at the address contained in tib with in set to zero.

Word: quit
Description: Clear the return stack, stop compilation, and return control to the operator's terminal. No ok message is sent.

Word: r
Stack: –> n
Description: Copy the top of the return stack to the computation stack.

Word: r>
Stack: –> n
Description: Pops the top value from the return stack and pushes it onto the computation stack. Called "R-from." Also see >r and r.

Word: r0
Stack: –> addr
Description: A user variable containing the initial location of the return stack. Called "R-zero." Also see rp!

Word: repeat
Stack: addr n –> (compiling)
Description: Used within a colon-definition in the form:

```
begin  ...  while  ...  repeat
```

At run-time, repeat forces an unconditional branch back to just after the corresponding begin.

At compile-time, repeat compiles branch and the offset from here to addr. n is used for error-checking.

Word: rot
Stack: n1 n2 n3 –> n2 n3 n1
Description: Rotate the top three values on the stack, bringing the third to the top.

Word: rp!
Description: A procedure to initialize the return stack pointer from user variable r0. Extreme caution should be used with this word.

Word: rp@
Stack: –> addr
Description: A procedure that places the return stack pointer address onto the stack.

Word: s->d
Stack: n –> d
Description: Sign extend a single number to form a double number.

Word: s0
Stack: –> addr
Description: A user variable that contains the initial value for the stack pointer. Called "S-zero." Also see sp!

Word: sign
Stack: n d –> d
Description: Stores an ASCII hyphen (-) just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <# and #>.

Word: smudge
Description: Used during word definition to toggle the "smudge bit" in a word's name field. This prevents an uncompleted definition from being found during dictionary searches until compiling is completed without error.

Word: sp!
Description: A procedure to initialize the stack pointer from s0.

Word: sp@
Stack: –> addr
Description: A procedure to return the address of the stack position to the top of the stack, as it was before sp@ was executed. For example, the line 1 2 sp@ @ . . . would type 2 2 1).

Word: space
Description: Transmit an ASCII blank to the serial port.

Word: spaces
Stack: n –>
Description: Transmit n ASCII blanks to the serial port.

Word: state
Stack: –> addr
Description: A user variable containing the compilation state. A non-zero value indicates compilation state. A zero value indicates execution state.

Word: swap
Stack: n1 n2 –> n2 n1
Description: Exchange the top two values on the stack.

Word: `task`
Description: A no-operation word which can mark the boundary between user definitions and the ibcl default dictionary.

Word: `then`
Description: An alias for `endif`.

Word: `tib`
Stack: `–> addr`
Description: A user variable containing the address of the terminal input buffer.

Word: `toggle`
Stack: `addr b –>`
Description: Complement the contents of `addr` by the bit pattern `b`.

Word: `traverse`
Stack: `addr1 n –> addr2`
Description: Move across the name field of a variable length IBCL name field. `addr1` is the address of either the length byte or the last letter. If `n=1`, the motion is toward high memory; if `n=-1`, the motion is toward low memory. The `addr2` resulting is the address of the other end of the name.

Word: `type`
Stack: `addr n –>`
Description: Transmit `n` characters from address `addr` to the serial port.

Word: `u`
Stack: `un –>`
Description: Print a number from an unsigned 16-bit value, converted according to the numeric `base`. A trailing blank follows. Pronounced "U-dot."

Word: `u*`
Stack: `u1 u2 –> ud`
Description: Leave the unsigned double number product of two unsigned numbers.

Word: u<
Stack: u1 u2 –> f
Description: Leave a TRUE flag if u1 is less than u2; otherwise leave a FALSE flag.

Word: u/
Stack: ud u1 –> u2 u3
Description: Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

Word: ulm
Stack: addr un –>
Description: ulm is a binary output word. ulm uploads from the GPIB-CT to the host un bytes starting at addr.

Word: until
Stack: f –> (run-time)
addr n –> (compile)
Description: Occurs within a colon-definition in the form:

```
begin ... until
```

At run-time, until controls the conditional branch back to the corresponding begin. If f is FALSE, execution returns to just after begin; if TRUE, execution continues ahead.

At compile-time, until compiles (0branch) and an offset from here to addr. n is used for error-checking.

Word: `user`
Stack: `n –>`
Description: A defining word used in the form:

```
n  user  cccc
```

which creates a user variable `cccc`. The parameter field of `cccc` contains `n` as a fixed offset relative to the user pointer register user-base for this user variable. When `cccc` is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

Word: `variable`
Description: A defining word used in the form:

```
n  variable  cccc
```

When `variable` is executed, it creates the definition `cccc` with its parameter field initialized to `n`. When `cccc` is later executed, the address of its parameter field (containing `n`) is left on the stack, so that a fetch or store may access this location.

Word: `voc-link`
Stack: `–> addr`
Description: A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for `forget`ting through multiple vocabularies.

Word: `vocabulary`
Stack: `—>`
Description: A defining word used in the form:

```
vocabulary  cccc  immediate
```

to create a vocabulary definition `cccc`. Subsequent use of `cccc` will make it the `context` vocabulary which is searched first by `interpret`. The sequence `cccc definitions` will also make `cccc` the `current` vocabulary into which new definitions are placed.

`cccc` will be so chained as to include all definitions of the vocabulary in which `cccc` is itself defined. All vocabularies ultimately chain to `ibcl`. By convention vocabulary names are to be declared `immediate`. Also see `voc-link`.

Word: `vlist`
Stack: `->`
Description: List the names of the definitions in the `context` vocabulary. Any serial character received will terminate the listing.

Word: `warm`
Description: Similar to `cold`, except that the dictionary is not cleared.

Word: `warning`
Stack: `—> addr`
Description: A user variable containing a value controlling messages. If `warning` is 0 or 1, messages are displayed by number. If `warning` is -1, execute `(abort)` for a user-specified procedure. See `message`, `error`.

Word: `while`
Stack: `f --- (run-time)`
`addr1 n1 --- addr1 n1 addr2 n2`
Description: Occurs in a colon-definition in the form:

```
begin ... while (true part) ... repeat
```

At run-time, `while` selects conditional execution based on boolean flag `f`. If `f` is TRUE (non-zero), `while` continues execution of the TRUE part through to `repeat`, which then branches back to `begin`. If `f` is FALSE (zero), execution skips to just after `repeat`, exiting the structure.

At compile-time, `while` emplaces `(0branch)` and leaves `addr2` of the reserved offset. The stack values will be resolved by `repeat`.

Word: `width`
Stack: `–> addr`
Description: A user variable containing the maximum number of characters saved in the compilation of a definition's name. It must be 1 through 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in `width`. The value may be changed at any time within the above limits.

Word: `word`
Stack: `c –>`
Description: Read the next text characters from the input stream being interpreted until a delimiter `c` is found, storing the packed character string beginning at the dictionary buffer `here`. `word` leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurrences of `c` are ignored.

Word: `xor`
Stack: `n1 n2 –> n3`
Description: Leave the bitwise logical exclusive-OR of two values.

Word: `[`
Description: Used in a colon-definition in the form:

```
: xxx [ words ] more words ;
```

Suspend compilation. The words after `[` are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with `]`. Also see `literal` and `]`.

Word: `[compile]`
Description: Used in a colon-definition in the form:

```
: xxx [compile] cccc ;
```

`[compile]` will force the compilation of the immediate word `cccc` that would otherwise execute during compilation.

Word: `]`
Description: Resume compilation, to the completion of a colon-definition. Also see `[`.

Appendix G
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Corporate Headquarters
(512) 795-8248
Technical Support fax: (512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	03 879 9422	03 879 9179
Austria	0662 435986	0662 437010 19
Belgium	02 757 00 20	02 757 03 11
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 65 33 00	1 48 65 19 07
Germany	089 7 14 50 93	089 7 14 60 35
Italy	02 48301892	02 48301915
Japan	03 3788 1921	03 3788 1923
Netherlands	01720 45761	01720 42140
Norway	03 846866	03 846860
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 27 00 20	056 27 00 25
U.K.	0635 523545	0635 523154
	or 0800 289877 (in U.K. only)	

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___)_____ Phone (___)_____

Computer brand _____

Model _____ Processor _____

Operating system _____

Speed _____MHz RAM _____M

Display adapter _____

Mouse _____yes _____no

Other adapters installed _____

Hard disk capacity _____M Brand _____

Instruments used _____

National Instruments hardware product model _____

Revision _____

Configuration _____

(continues)

National Instruments software product _____

Version _____

Configuration _____

The problem is _____

_____

_____

_____

_____

_____

_____

_____

List any error messages _____

_____

_____

_____

_____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

_____

_____

_____

_____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **GPIB-CT IBCL Reference Manual**

Edition Date: **December 1993**

Part Number: **320132-01**

Please comment on the completeness, clarity, and organization of the manual.

(continues)

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (_____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678

Glossary

Prefix	Meaning	Value
m- μ- n-	milli- micro- nano-	10^{-3} 10^{-6} 10^{-9}

EPROM	erasable programmable read-only memory
hex	hexadecimal
in.	inches
K	1,024 bytes (of memory)
M	megabytes of memory
sec	seconds