Series 9800 Desktop Computers

Assembly Development ROM Manual

For the HP 9845

HEWLETT PACKARD

HEWLETT PACKARD

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard Desktop Computer Division products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from date of delivery.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

*For other countries, contact your local Sales and Service Office to determine warranty terms.

Assembly Development ROM

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

Periodically, this manual is updated. Each new edition of this manual incorporates all material updated since the previous edition. Each new or revised page is indicated by a revision date.

The date on the back cover changes only when each new edition is published.

First Printing...March 1, 1980

Table of Contents

Chapter 4: Assembly Language Fundamentals

Chapter 5: Arithemetic

Chapter 1
General Information

Welcome to the world of assembly language programming on the System 45[1].

It is the design of the Assembly Execution and Development Read Only Memory (ROM) and the Assembly Execution ROM to help extend the capabilities of your 9845 by giving you greater control and speed through the use of machine instructions, pseudo-instructions, and extensions to the BASIC language.

The assembly language system is provided to you as one of two ROMs which plug into the right ROM drawer of your System 45. The two ROMs are:

- The Assembly Execution and Development ROM – used to write and debug assembly language programs on the System 45, and has the complete capability of the Assembly Execution ROM.
- The Assembly Execution ROM – provides the capability to load, run, and store assembled routines and modules. Information about this ROM can be found in the Assembly Execution ROM manual.

When installed, the Assembly Execution and Development ROM reserves some read/write memory which cannot be accessed for storage of programs or data. (The Assembly Execution ROM also reserves memory.) The following table describes the actual read/write memory used (in 8-bit bytes) under various configurations:

	Execution ROM Only		Execution and Development ROM	
	I/O ROM Present	I/O ROM Not Present	I/O ROM Present	I/O ROM Not Present
Power on	270	334	590	654
After first pre-run	708	772	1028	1092

It is assumed throughout this manual that you are familiar with the basic operation and language of the 9845. It is also assumed that you are reasonably well-acquainted with at least one other assembly language.

1 The assembly language programming capability is not available for the System 45A computer.

Equipment Supplied

The following items are supplied with the Assembly Execution and Development ROM –

Item	Part Number
Assembly Development ROM Manual	09845-91083
Assembly Execution ROM Manual	09845-91082
Assembly Language Quick Reference	09845-91080
BASIC Language Interfacing Concepts	09835-90600
Demonstration Cartridge	11141-10155
Error Label	7120-8771

Structure of the Manual

It is the intent of this manual that you should be able to find between its covers everything you need to know to use the assembly language effectively. However, since assembly language programming is a complex topic, the manual relies a great deal on your past experience. Most of the information is in succinct presentations of a particular topic; it is not the intent to "teach" assembly language programming to someone not familiar with the topic.

The major topics covered are: assembly language program creation, the processor and relevant operating system constructs, assembly language fundamentals, BCD and integer arithmetic, communications with BASIC, I/O handling, debugging tools, errors and error processing, and graphics. Each topic (chapter), has a summary at the beginning detailing the information to be presented therein.

The manual is organized so that each topic can be covered completely within a given chapter. This approach was chosen over the strict syntactical or semantical treatment of the individual statements and instructions. As a consequence, you may find this difficult to use as a "quick reference" for syntax and meaning of the individual commands.

To meet your needs for "quick reference" material, an Assembly Language Quick Reference Manual (HP part number 09845-91080) is provided. In addition, you will find much of the information in this manual condensed and tabulated in the various appendices of this manual.

A recommended method for using the manuals is to start with this one as your basic learning tool. Then you should be able to use the Quick Reference Manual effectively for all future reference.

Purpose of the ROMs

The Assembly Execution and Development ROM is used to write and debug assembly language programs on the System 45, and also has the complete capability of the Assembly Execution ROM. The Assembly Execution ROM provides the capability to load, run, and store assembled routines and modules.

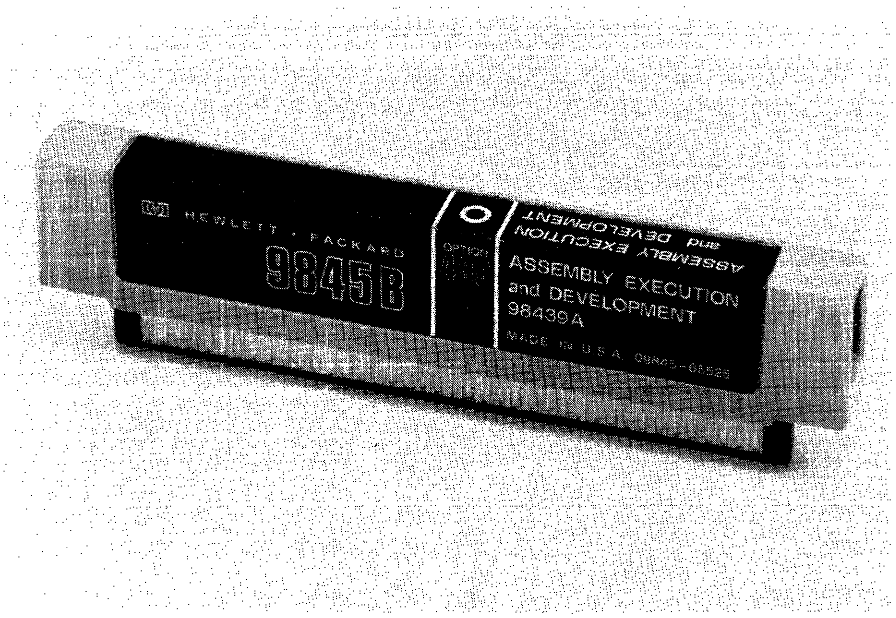
The Assembly Execution ROM is used independently of the Assembly Execution and Development ROM. Because of the overhead required by the debugging features of the Assembly Execution and Development ROM, programs run slightly more rapidly if the Assembly Execution ROM is used rather than the Assembly Execution and Development ROM.

ROM Installation

Before assembly language programming can proceed, the ROMs must be in place. The installation is a simple process.

There are two ROM drawers for the computer: one on the right side of the machine one on the left. The ROM is installed in the right ROM drawer, using these steps:

- Pull the right ROM drawer out.
- Squeeze the sides of the plastic cover and lift to gain access to the drawer connectors.
- Position the ROM over one of the connectors denoted by a ○ or □ marking.
- Press the ROM onto the ROM drawer connector so that it seats all the way down. The small circular keys on the sides of the ROM drawer should fit into the recesses in the bottom of the plastic ROM case. If they don't, make sure that you have properly oriented the ROM.



Assembly Language System ROM

After inserting the ROM, close the drawer until it is flush with the outside cover of the machine. With this done, you are now ready to begin writing assembly language programs.

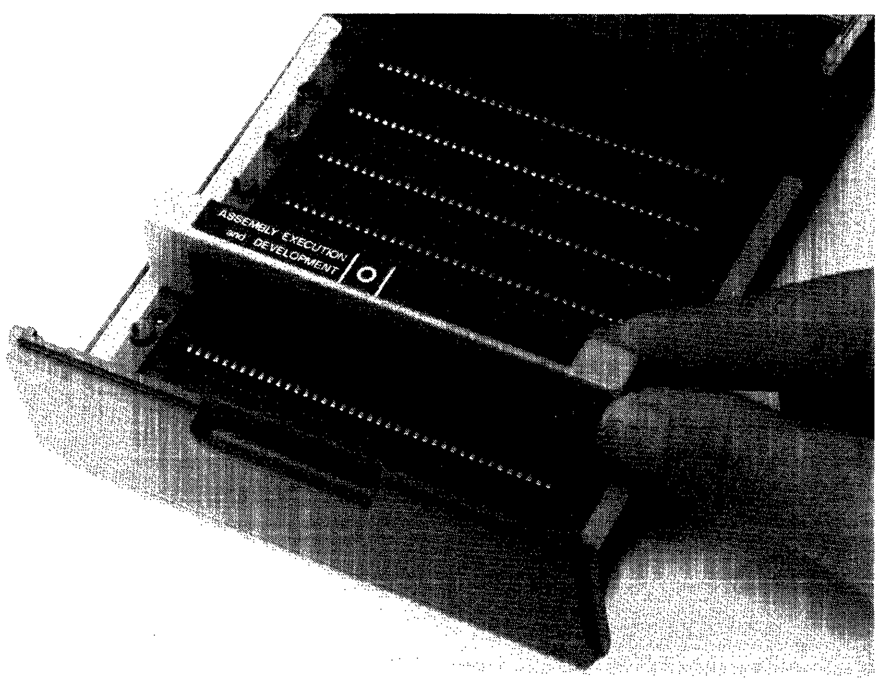


Figure 1. Installing the Assembly Execution and Development ROM

Buzzwords

During the course of the discussions in this manual, words and phrases are used which are in common circulation among those who are familiar with assembly languages. While the meanings of most are either well-known, or are deducible from the context, there are a few which may be unfamiliar, or unique to the 9845 assembly language, or are variable from one assembly language to the next and thus need to be defined for this one. They are —

assembled location — a reference to a location in memory which may be specified in one of the following forms —

{symbol} [, {numeric expression}]
{expression} [, {numeric expression}]

where:
{symbol} is an assembly location. It may be a label for a particular machine instruction (in which case the address of the associated instruction is used), or an assembler-defined symbol (in which case the associated absolute address is used), or a symbol defined by an EQU instruction (described in the "Symbolic Operations" section of Chapter 4).

{expression} may be a numeric expression or a string expression. If numeric, a decimal calculation is performed and the result is interpreted as an octal value; if the result is not an octal representation of an integer, an error results. If a string expression is used, the string must be interpretable as either an octal integer constant or a known assembly symbol (see {symbol} above).

{numeric expression} serves as a decimal offset from the given label or constant.

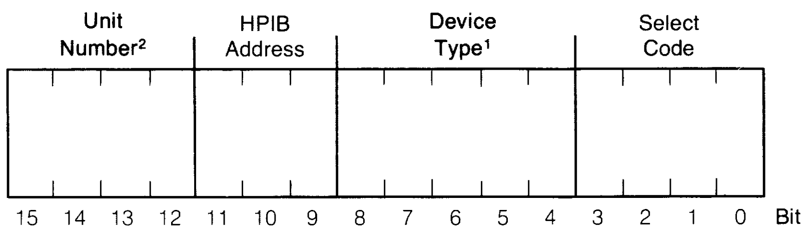
busy bits — each variable located in the BASIC value or common areas has associated with it two bits: a "read" busy bit, and a "write" busy bit. When a "read" busy bit is set, attempts should not be made to perform a function on that variable. A read operation may be performed on a "write-busy" variable. When the busy bit is cleared, the function may be performed on the variable.

byte — a group of 8 binary digits (bits).

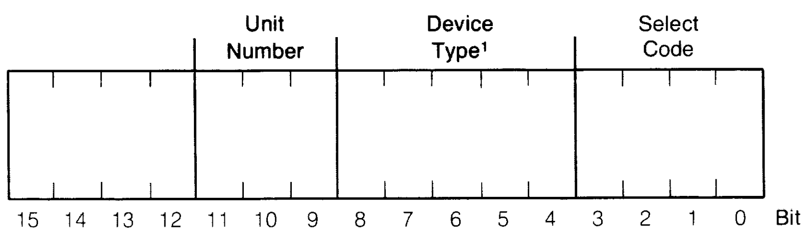
conditional assembly — an assignation that certain portions of a module are not to be assembled unless a condition has been set. The portions begin with any of the IFA through IFH, and IFP, pseudo-instructions, and end with the next XIF pseudo-instruction. IFA uses the A-condition as a test, and so on. The conditions are set by the statement assembling the module (IASSEMBLE).

interrupt service routine (ISR) — an assembly language routine intended to perform a certain action, or set of actions, when the computer receives a request from an external device. An "active" ISR is one which is currently enabled for a given device.

mass storage unit specifier (msus) — a single word corresponding to the BASIC language mass storage unit specifier as described in either the 9845 Operating and Programming Manual or the Mass Storage Techniques Manual. An msus has one of the following structures —



or



for the 9885MS Flexible Disk Drive

An msus can designate the current default as its mass storage device (meaning it will use the device indicated by the last MASS STORAGE IS statement executed). This is designated by having the msus be all ones (i.e., equal to − 1).

object module — a section of assembled code stored in the particular region of memory set aside for it. Though the source module for the object code may no longer be resident in memory, when created, the module was delimited by certain pseudo-instructions (NAM and END) and is referenced by the name given to it by the NAM pseudo-instruction.

octal expression — a numeric expression which, when displayed or printed, appears as an octal (base-8) number. Within arithmetic operations, it has a decimal value (base-10). Thus, the value 17_8 will appear as 17 (representing the value 15_{10}), but if arithmetic was performed on it, it would act as if it were 17_{10}. All octal expressions are necessarily integers in the range of 0 to 177777_8.

1 The device type is the ASCII code for the type minus 100B.
2 For tape operations, bits 9-15 are zeros.

pixel — picture element — the smallest unit of resolution on the CRT.

source module — a section of assembly language source code beginning with a NAM pseudo-instruction and ending with the END pseudo-instruction.

word — two bytes; a group of 16 binary digits (bits).

B — octal radix specifier. For example 177777B is 177777 octal. If the trailing "B" is not present, the assembler assumes decimal.

*— shorthand for current location. For example,

```
        SFC *       ! Skip if flag clear to current location.
```

is equivalent to —

```
Here:   SFC Here    ! Skip if flag clear to Here.
```

Fundamental Syntax

The syntax conventions used in this manual are those used in the Operating and Programming Manual for the 9845.

dot matrix	All syntax items displayed in dot matrix form should be programmed as shown.
[ ]	Items contained in brackets are optional items.
...	Ellipses mean that the previous item may be repeated indefinitely.

In addition, the following convention is employed throughout the Assembly Language series of manuals —

{ }	Items contained in braces are syntax items considered as a unit. The names inside are usually descriptive of the function intended for that item. Whenever an item enclosed in braces appears in the text, the notation refers to the same notation within an earlier syntax.

Chapter 2
Getting Started

Summary: This chapter contains a general discussion of the assembly language system. A format for the creation of an assembly language program is presented. Topics such as modules, routines, and memory allocation are discussed, along with methods of using them effectively. Also discussed is the storage and retrieval of modules on mass storage.

The thing to remember about the assembly language system is that it has been thoroughly integrated into the operating system of the System 45. Once the ROMs have been installed, you are able immediately to begin programming in assembly language. In addition, you have the capability to load and store your programs on mass storage, to assemble them separately or leave them in source form, to execute them from BASIC and pass BASIC variables to them, and to debug them, including a full pausing and stepping capability.

Developing Routines for Later Use

Most assembly language programs are written with the intent that they will be used many times, not just at the time they are written. It is for just such program development that the full capabilities of the assembly language system come into play. The development comes in several stages. Each stage has its unique requirements and the tools to meet those requirements.

The first stage is creation of the source program. This is achieved by the use of the editing capabilities of the System 45. Additionally, the mass storage capabilities of the computer can be used.

The second stage is the creation of the object (or machine) code. This requires not only an assembly of the source, but the ability to allocate special locations in memory to hold the newly created object code.

The third stage is the validation of the routines as written, commonly known as "debugging". This is enabled by calls from a BASIC driver, followed by application of various debugging tools provided by the assembly system. The capabilities to pause and step a program have been extended to assembly language instructions to assist this process.

The fourth stage is to store away the debugged object code so that it may be used at a later time. A special mass storage statement is provided by the assembly language system. This statement stores object code into a special assembly file.

Finally, the end-user of the routines must be able to retrieve the object code from mass storage as it is needed. He also must be able to access the routines from BASIC programs. Both these needs are met with the Execution ROM, so the capabilities are not only provided, but they are provided independently of the program development capabilities located in the Assembly Execution and Development ROM.

Each of the topics involved in these stages is discussed at length in this manual.

Figure 2 presents a graphical presentation of this overview.

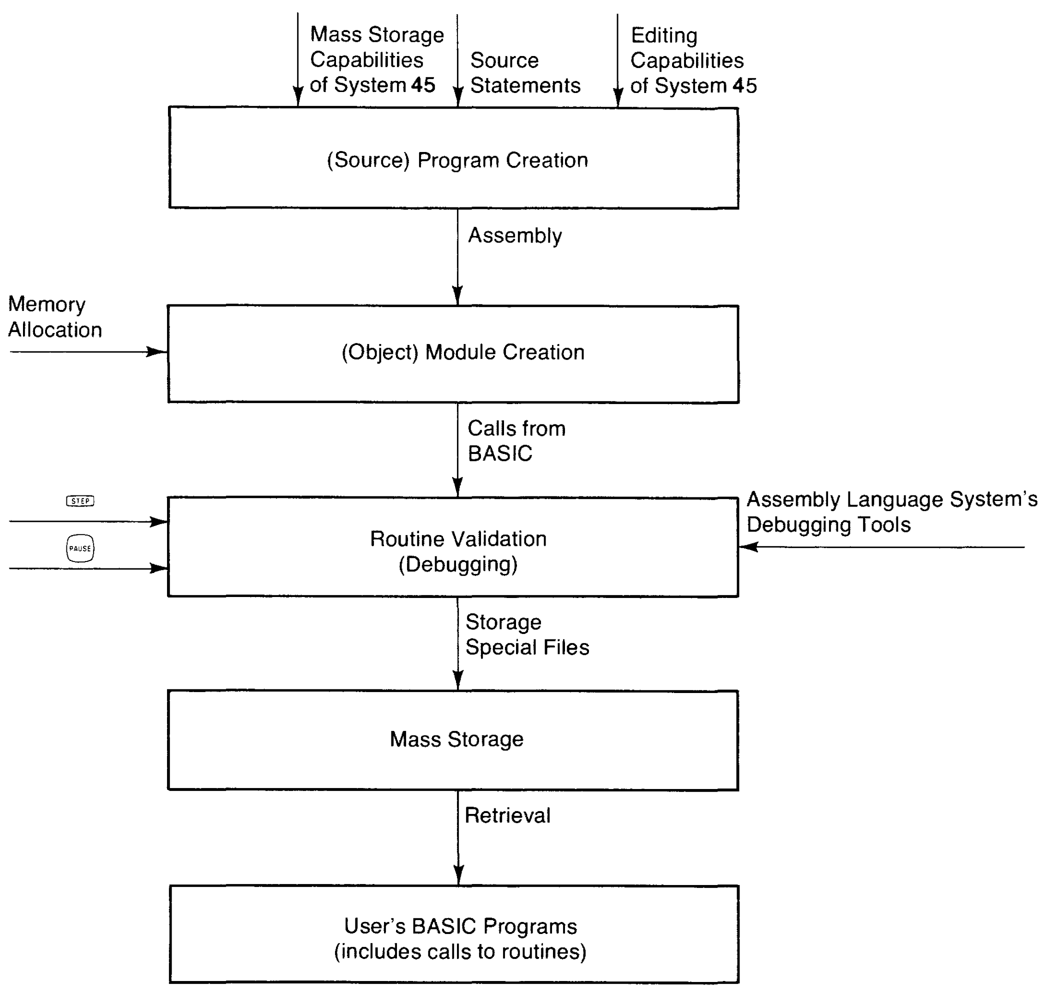


Figure 2. Overview of Assembly Language Routine Development Process

Overview

At this point, there are three fundamental structures to be explained: programs, modules, and routines.

A **program** is the set of source statements from which the object (or machine) code is generated. The assembly source statements are extensions to the BASIC language which is used in the System 45. The statements themselves are stored in the machine as part of the BASIC program in which they reside. At some point, you must take the assembly source statements and assemble them into object code, in order for them to be run. The object code is stored in a specified location in the machine.

A **module** is a subset of the object code. It is a means of separating and identifying parts of the code so that those parts may be used individually (as in mass storage operations). There may be any number of modules present at any one time, limited only by the amount of memory allocated for object code.

A **routine** is a "callable" section of a module. It is analogous to the subprogram in BASIC. It has a named entry point, possibly a parameter list, and a return. A module may contain any number of routines, again limited only by the amount of memory allocated to hold the object code.

In short, the usefulness of each structure is as follows —

- **Programs** contain assembly language source code.
- **Modules** contain object code to be loaded from or stored on mass storage.
- **Routines** are executable sections of object code.

Program Creation

The first matter which is likely to concern you about the assembly language system is how to create an assembly language program.

In general, the process of creating an assembly language subprogram consists of the following steps —

1. Enter and store the source code (program).
2. Create an area in memory which will ultimately contain the object code.
3. Assemble the source code into object code, storing the latter into the area of memory set aside for it.
4. Execute the object code (routines) from BASIC "drivers".

Each of these steps will be discussed at length in the pages of this manual, along with a number of not-so-incidental side-topics (such as "debugging" techniques). The purpose of **this** short section is to give you an impression of the general procedure through which an assembly language subprogram is created.

As an example to use to demonstrate the process, suppose the following task has been assigned to you —

> Requirement: Write an assembly language subprogram which takes two integer values and multiplies them together as integers. If the result overflows the range of an integer (− 32 768 to + 32 767), then the subprogram should return the same error as the system would (i.e., error number 20).

With this task in hand, suppose that you have completed a programming analysis that suggests that the following assembly language source code would fulfill the subprogram's functions —[1]

```
10   ISOURCE             NAM Multiplication ! Beginning of module
20   ISOURCE             EXT Error_exit,Get_value,Put_value ! Utilities
30   ISOURCE Integers:   BSS 2              ! Storage area for integers created
40   ISOURCE             SUB                ! Indicates entry point follows
50   ISOURCE Input1:     INT                ! Indicates "integer parameters are
60   ISOURCE Input2:     INT                !   passed in the order given by these
70   ISOURCE Output:     INT                !   statements and are given names
80   ISOURCE Multiply:   LDA =Integers      ! Actual entry point (name: Multiply);
90   ISOURCE             LDB =Input1        !   routine begins by fetching actual
100  ISOURCE             JSM Get_value      !   value of the input parameters
110  ISOURCE             LDA =Integers+1    !   from BASIC and storing them where
120  ISOURCE             LDB =Input2        !   the routine can use them
130  ISOURCE             JSM Get_value
140  ISOURCE             LDA Integers       ! Then it loads the values into the
150  ISOURCE             LDB Integers+1     !   arithmetic accumulator and
160  ISOURCE             MPY                !   finally multiplies them
170  ISOURCE             SBP *+2            ! A check for overflow is performed
180  ISOURCE             CMB                !   by checking the result for anything
190  ISOURCE             SZB *+3            !   in the B register when it should be 0
200  ISOURCE             LDA =20            !   and if it isn't, Error 20 is selected
210  ISOURCE             JSM Error_exit     !   and the routine is aborted
220  ISOURCE             STA Integers       ! If everything is OK, then result stored
230  ISOURCE             LDA =Integers      ! The product is then returned to the
240  ISOURCE             LDB =Output        !   output variable in BASIC listed
250  ISOURCE             JSM Put_value      !   among the arguments
260  ISOURCE             RET 1              ! We're finished, so return to BASIC
270  ISOURCE             END Multiplication ! End of module
```

1 The fact that it is rarely possible to create a running program at this stage should not get in the way of accepting the example. Usually there is debugging involved in later stages.

Now that the routine has been developed, it is necessary to get it into the memory of the machine as a program. This is done by preceding each and every assembly language statement with the keyword ISOURCE and entering it as a program line. The process of entering (with the keyword included) is the same as with any other BASIC statement — so you can use EDIT or AUTO and the (STORE) key in the same way you normally enter any BASIC statement. (This process is fully described in the "Program Entry" section of this chapter.)

The final result of entering the routine would look something like —

```
10  ISOURCE             NAM Multiplication ! Beginning of module
20  ISOURCE             EXT Error_exit,Get_value,Put_value ! Utilities
30  ISOURCE  Integers:  BSS 2            ! Storage area for integers created
40  ISOURCE             SUB              ! Indicates entry point follows
50  ISOURCE  Input1:    INT              ! Indicates "integer parameters are
60  ISOURCE  Input2:    INT              !   passed in the order given by these
70  ISOURCE  Output:    INT              !   statements and are given names
80  ISOURCE  Multiply:  LDA =Integers    ! Actual entry point (name: Multiply);
90  ISOURCE             LDB =Input1      !   routine begins by fetching actual
100 ISOURCE             JSM Get_value    !   value of the input parameters
110 ISOURCE             LDA =Integers+1  !   from BASIC and storing them where
120 ISOURCE             LDB =Input2      !   the routine can use them
130 ISOURCE             JSM Get_value
140 ISOURCE             LDA Integers     ! Then it loads the values into the
150 ISOURCE             LDB Integers+1   !   arithmetic accumulator and
160 ISOURCE             MPY              !   finally multiplies them
170 ISOURCE             SBP *+2          ! A check for overflow is performed
180 ISOURCE             CMB              !   by checking the result for anything
190 ISOURCE             SZB *+3          !   in the B register when it should be 0
200 ISOURCE             LDA =20          !   and if it isn't, Error 20 is selected
210 ISOURCE             JSM Error_exit   !   and the routine is aborted
220 ISOURCE             STA Integers     ! If everything is OK, then result stored
230 ISOURCE             LDA =Integers    ! The product is then returned to the
240 ISOURCE             LDB =Output      !   output variable in BASIC listed
250 ISOURCE             JSM Put_value    !   among the arguments
260 ISOURCE             RET 1            ! We're finished, so return to BASIC
270 ISOURCE             END Multiplication ! End of module
```

This source code demonstrates the three critical items in assembly subprograms. First, a routine has to be part of a module; modules are delimited with the NAM and END pseudo-instructions (see lines 10 and 270 in the source). Second, a routine has to have an entry point; this consists of a SUB pseudo-instruction (see line 40), any parameters (see lines 50 through 70), and a name (the label used on the first machine instruction following the SUB, see line 80). Finally, a routine must be able to return to the BASIC program which called it; this is accomplished with the RET 1 instruction (see line 260).

The NAM, END, and SUB pseudo-instructions are discussed in Chapter 4. The RET 1 instruction is discussed in Chapter 3.

The next three steps in program creation are each satisfied with BASIC-executable statements. Creation of a storage area for the object code for the program (which can be estimated at less than 40 words; there is essentially one word of object code per line of source) is accomplished by programming the statement —

```
280  ICOM 40
```

(The ICOM statement is fully discussed in the "Setting Aside Memory" section of this chapter.)

This can be followed in the same program by an instruction to assemble the source code into object code —

```
290  IASSEMBLE Multiplication
```

(The IASSEMBLE statement is fully discussed in Chapter 4.)

If the assembly is successful (and it will be in this example), then the routine can be called and used as desired. A typical call looks like —

```
600  ICALL Multiply(Index,Dimension,Subscript)
610  Array(Subscript)=Value
```

(The ICALL statement is fully discussed in Chapter 6.)

Thus, the final result could easily be —

```
10  ISOURCE                NAM Multiplication ! Beginning of module
20  ISOURCE                EXT Error_exit,Get_value,Put_value ! Utilities
30  ISOURCE  Integers:     BSS 2              ! Storage area for integers created
40  ISOURCE                SUB                ! Indicates entry point follows
50  ISOURCE  Input1:       INT                ! Indicates "integer parameters are
60  ISOURCE  Input2:       INT                !   passed in the order given by these
70  ISOURCE  Output:       INT                !   statements and are given names
80  ISOURCE  Multiply:     LDA =Integers      ! Actual entry point (name: Multiply);
90  ISOURCE                LDB =Input1        !   routine begins by fetching actual
100 ISOURCE                JSM Get_value      !   value of the input parameters
110 ISOURCE                LDA =Integers+1    !   from BASIC and storing them where
120 ISOURCE                LDB =Input2        !   the routine can use them
130 ISOURCE                JSM Get_value
140 ISOURCE                LDA Integers       ! Then it loads the values into the
150 ISOURCE                LDB Integers+1     !   arithmetic accumulator and
160 ISOURCE                MPY                !   finally multiplies them
170 ISOURCE                SBP *+2            ! A check for overflow is performed
180 ISOURCE                CMB                !   by checking the result for anything
190 ISOURCE                SZB *+3            !   in the B register when it should be 0
200 ISOURCE                LDA =20            !   and if it isn't, Error 20 is selected
210 ISOURCE                JSM Error_exit     !   and the routine is aborted
220 ISOURCE                STA Integers       ! If everything is OK, then result stored
```

```
230 ISOURCE                LDA =Integers    ! The product is then returned to the
240 ISOURCE                LDB =Output      !   output variable in BASIC listed
250 ISOURCE                JSM Put_value    !   among the arguments
260 ISOURCE                RET 1            ! We're finished, so return to BASIC
270 ISOURCE                END Multiplication ! End of module
280  ICOM 40
290  IASSEMBLE Multiplication
        .
        .
        .
600  ICALL Multiply(Index,Dimension,Subscript)
610  Array(Subscript)=Value
        .
        .
        .
```

It isn't necessary that a program be assembled in every BASIC program which uses it. Object code can be stored on mass storage with a statement like —

```
300  ISTORE Multiplication;"MULT"
```

So if the example were instead made to read —

```
10  ISOURCE              NAM Multiplication ! Beginning of module
20  ISOURCE              EXT Error_exit,Get_value,Put_value ! Utilities
30  ISOURCE  Integers:   BSS 2            ! Storage area for integers created
40  ISOURCE              SUB              ! Indicates entry point follows
50  ISOURCE  Input1:     INT              ! Indicates "integer parameters are
60  ISOURCE  Input2:     INT              !   passed in the order given by these
70  ISOURCE  Output:     INT              !   statements and are given names
80  ISOURCE  Multiply:   LDA =Integers    ! Actual entry point (name: Multiply);
90  ISOURCE              LDB =Input1      !   routine begins by fetching actual
100 ISOURCE              JSM Get_value    !   value of the input parameters
110 ISOURCE              LDA =Integers+1  !   from BASIC and storing them where
120 ISOURCE              LDB =Input2      !   the routine can use them
130 ISOURCE              JSM Get_value
140 ISOURCE              LDA Integers     ! Then it loads the values into the
150 ISOURCE              LDB Integers+1   !   arithmetic accumulator and
160 ISOURCE              MPY              !   finally multiplies them
170 ISOURCE              SBP *+2          ! A check for overflow is performed
180 ISOURCE              CMB              !   by checking the result for anything
190 ISOURCE              SZB *+3          !   in the B register when it should be 0
200 ISOURCE              LDA =20          !   and if it isn't, Error 20 is selected
210 ISOURCE              JSM Error_exit   !   and the routine is aborted
220 ISOURCE              STA Integers     ! If everything is OK, then result stored
230 ISOURCE              LDA =Integers    ! The product is then returned to the
240 ISOURCE              LDB =Output      !   output variable in BASIC listed
250 ISOURCE              JSM Put_value    !   among the arguments
260 ISOURCE              RET 1            ! We're finished, so return to BASIC
270 ISOURCE              END Multiplication ! End of module
280  ICOM 40
290  IASSEMBLE Multiplication
300  ISTORE Multiplication;"MULT"
310  END
```

the object code is consequently stored into the file "MULT".

Later programs can retrieve the object code for use, such as in the following program —

```
10   INTEGER Dimension,Index,Subscript
20   ICOM 40
30   ILOAD "MULT"
      .
      .
      .
600  ICALL Multiply(Index,Dimension,Subscript)
610  Array(Subscript)=Value
      .
      .
      .
```

(Both ISTORE and ILOAD are discussed in the "Retrieving and Storing Modules" section of this chapter.)

Program Entry

The assembly language source statement is an **extension** to the BASIC language used in the System 45. This means that each assembly language statement is entered using a "keyword" — in this case ISOURCE — as a message to the operating system that the line is an assembly language statement.

By looking at an example, you can see what is meant —

```
10   LET A=10
20   LET B=20
30   PRINT A,B
40   ISOURCE    NAM Example
50   ISOURCE    NOP
60   ISOURCE    END Example
70   END
```

Lines 10, 20, 30, and 70, are all recognizable as BASIC statements. The keywords they use — LET, PRINT, and END — direct that certain actions take place. Lines 40, 50, and 60, are all assembly language statements; this was indicated by the ISOURCE keyword used in these lines.

Entering assembly language statements, by using the ISOURCE keyword, is thereby the same process as entering other types of BASIC statements. You may use all of the system editing features that you are used to using in the creation of BASIC programs — EDIT, AUTO, etc. You store each line with the STORE key, as you would any other BASIC line. See Appendix I for Demo Tape Special Function Keys which are useful for program entry.

Also, assembly lines do not have to be in any special place in the BASIC program. The previous example could be re-arranged as follows —

```
10   LET A=10
20   ISOURCE   NAM Example
30   LET B=20
40   ISOURCE   NOP
50   PRINT A,B
60   ISOURCE   END Example
70   END
```

Thus, you are free to enter your assembly statements anywhere in your BASIC program. But, you may ask, what is the effect of spreading them out like this? The answer is, simply, none. When the time comes to use them, assembly statements and BASIC statements are separated by the operating system and treated differently.

When the BASIC program is run, **only** the BASIC statements are executed. The ISOURCE statements are **ignored**, and, as you will be shown in Chapter 4, when the assembly language lines are assembled, the BASIC statements are ignored. A way to consider it is that there are two programs in one — BASIC's and the assembler's. So you can envision the example above as being this way —

BASIC

```
10   LET A=10
30   LET B=20
50   PRINT A,B
70   END
```

source

ASSEMBLER

```
20   ISOURCE   NAM Example
40   ISOURCE   NOP
60   ISOURCE   END Example
```

You should note, then, that ISOURCE statements are not "executable" in the usual BASIC sense. Their location in the program does not indicate the place where they will be executed. Assembly instructions are not executed until a routine is "called"; this is discussed in detail in Chapter 4.

Now that it has been said that the two types of statements can be thoroughly intermixed, it should also be said that the practice is **not recommended**. As a good programming practice — i.e., for readability and to preserve the self-documenting features of BASIC — it is recommended that assembly statements be collected together and placed in one spot in the program.

The first example is a recommended practice over the second, even though the second is permissible.

Other Extensions

In addition to the ISOURCE statement, there are a number of other BASIC language extensions provided by the assembly languge system. Unlike the ISOURCE statement, they are "executable", and their appearances are part of the BASIC lines (as distinguished from the assembler's). Where they appear is where the action associated with them is taken. This is identical to the way the other BASIC statements perform. The statements involved are —

IASSEMBLE
IBREAK
ICALL
ICHANGE
ICOM
IDELETE
IDUMP
ILOAD
INORMAL
IPAUSE OFF
IPAUSE ON
ISTORE
OFF INT
ON INT

Also provided are four numeric functions —

DECIMAL
IADR
IMEM
OCTAL

The functions can be used wherever numeric functions in general may be used.

All of these statements (except ICOM and ISOURCE) and the functions are available to you as live keyboard operations as well as programmable statements. A full discussion of each of the statements and functions can be found within this manual.

Modules and Routines

There are three basic activities associated with using assembled modules and routines. First, there is the need to retrieve them from wherever they may be stored (including providing a place for them to be kept while they are resident in the memory of the machine). Second, there is the actual execution of the routines. And third, there is the occasional requirement to store, or re-store a module on mass storage (including, perhaps, the need to free the space in memory it previously occupied).

Names

Routines, modules, and files all have names. The names given them may or may not bear some significance to one another; that depends upon you and the way that you name things.

Conventions for the naming of files and methods of general file manipulation can be found in the Operating and Programming Manual and in the Mass Storage Techniques Manual. The conventions are not any different than for files in general.

Names for modules are assigned with the creation of the source. In the assembly language source code, you have a NAM pseudo-instruction. This serves two purposes — to designate the beginning of the module and to assign the module a name. All of the assembly source statements which follow the NAM are in that module until an END pseudo-instruction is encountered. Thus, recalling the previous example —

```
20   ISOURCE   NAM Example
40   ISOURCE   NOP
60   ISOURCE   END Example
```

All of the ISOURCE statements between lines 20 and 60 (in this case, just the one) form the module called "Example". The formal syntaxes of these pseudo-instructions are —

NAM {module name}
END {module name}

{module name} is a symbol which becomes the name of the module. It follows the same rules as names in BASIC: up to fifteen characters; starts with a capital letter; followed by only non-capital letters, numbers, or the underscore character.

The {module name} in the END statement must correspond to the {module name} of the NAM statement or an assembly error ("EN") results.

You may have any number of modules in your source code. Each module begins with a NAM and ends with an END pseudo-instruction as above.

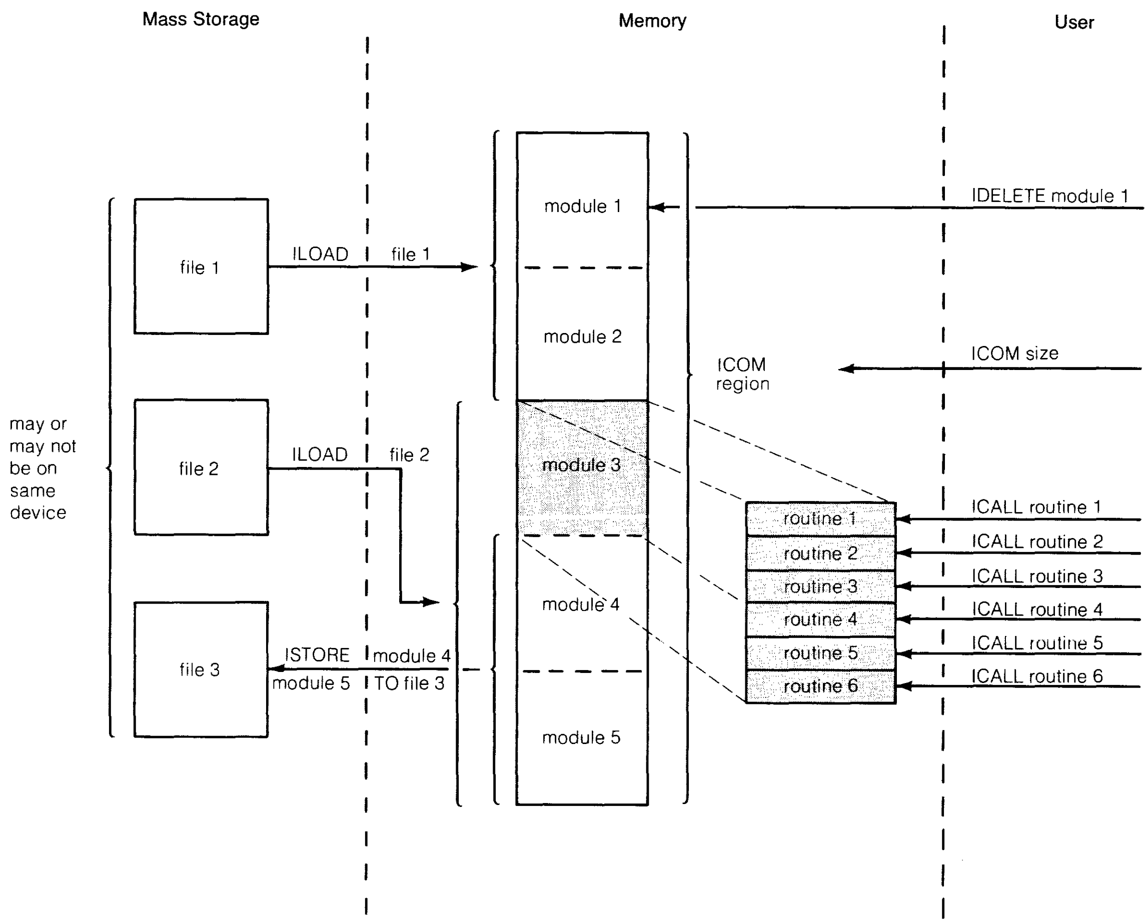


Figure 3. Overview of Routines and Modules

Survey of Modules and Routines

To sketch the functional relationships of modules and routines, please refer to Figure 3 above.

Modules are stored in files and may be retrieved and placed in memory using the "ILOAD" command. When the ILOAD command is executed, all of the modules in the file are loaded into the memory. Note that many files can be loaded, with many modules each, with all of the modules able to remain resident in the memory.

Alternatively, modules which are already in memory may be stored into a single file using the "ISTORE" command. When the ISTORE command is executed, the designated modules are stored into an "option ROM" (OPRM) type of file (on tape cartridges) or an "Assembly" (ASMB) type of file (on non-tape mass storage media). After storage, the modules are still in memory. They may be removed (i.e., the space they occupy in memory is "freed") by using the "IDELETE" command.

The area of memory where the modules are stored is called the "ICOM region". It is a particular contiguous area which must be large enough to hold all of the object code you wish to have resident in the memory at any one time.

Each module contains one or more routines. Your access to the routines is through the ICALL statement, which is very similar to the CALL statement used for BASIC subprograms. The ICALL statement may have arguments which you need to "pass" (send down) to the routine itself. What these arguments, if any, may be, and what meaning they hold depends upon what you have in mind for that routine. There are corresponding items in the assembly source code; these are discussed in Chapter 6.

Setting Aside Memory

As indicated by Figure 3, you cannot load a module until there is an ICOM region into which to load it. Neither can you assemble your source code into object code unless there is an ICOM region into which the object code can go.

The statement to use to create an ICOM region is —

```
ICOM {size}
```

where {size} is a non-negative integer constant indicating the number of words to be used to form the ICOM region. The maximum size is 32 718 words.

The ICOM statement is a "declaration"; that is, it is not executable, but rather is used when assignment of memory takes place just before a program is run. This is similar to a DIM or COM statement. As with a DIM or COM statement, the statement cannot be executed from the keyboard.

Once created, the ICOM region remains in existence until it is explicitly destroyed. But it is possible to change the size by using another ICOM statement.

The order in which modules appear in the ICOM region is determined by the order in which they are loaded using the ILOAD statement discussed in the next section or are created by the IASSEMBLE statement discussed in Chapter 4.

In most cases, the space which is freed by reducing the size of the ICOM region is returned to your available memory space. Sometimes, however, it is not returned, this being caused by the status of the common area allocated in memory, or by other option ROMs. The space is returned whenever —

- There is no common area assigned (with the COM statement); and,
- The requirements of another option ROM do not interfere.

There may be any number of ICOM statements in a program. The current size of the ICOM region is determined by the last one which appears in the program when the (RUN) key is pressed (or the command RUN is executed).

For example, suppose you have a program with the following statements in it —

```
20   ICOM 984
30   DIM A$[100]
       .
       .
       .
300  ICOM 492
       .
       .
       .
610  ICOM 2000
       .
       .
       .
900  END
```

Upon pressing (RUN), the ICOM region would be 2 000 words long. This is because line 610 is the final ICOM appearance.

The region continues to exist even if you load in another program which contains no ICOM statements. All ICOM statements must appear in the **main** program, not in any subprogram.

ICOM statements in a program must appear before any COM statement. This is to insure that the ICOM region will be allocated before the common is allocated.

There are three ways to eliminate the ICOM region —

- Execute SCRATCH A
- Execute ICOM 0 in a program.
- Turn off the machine.

After any of these actions, the region is no longer in existence. If there are any modules in the region, they disappear as well. If any of those modules contain an active interrupt service routine, you get an error (number 193) if you try to eliminate the region using ICOM 0. If any of your routines provided to other users contain active ISRs, your documentation for the routine should warn the users of that fact so they can avoid this error.

Two methods are recommended for deleting all previous modules. The methods differ only in the times at which the deletion operation is performed.

The first method involves the following sequence of statements:

```
100 ICOM 0
110 ICOM 2000
```

which assures that an ICOM region of 2000 words is in existence at program execution, and that the ICOM region is completely clear of any previously loaded modules. The deletion operation takes place every time the (RUN) key is pressed, before program execution begins.

The second method involves the use of the IDELETE statement in the following sequence:

```
100 ICOM 2000
110 IDELETE ALL
```

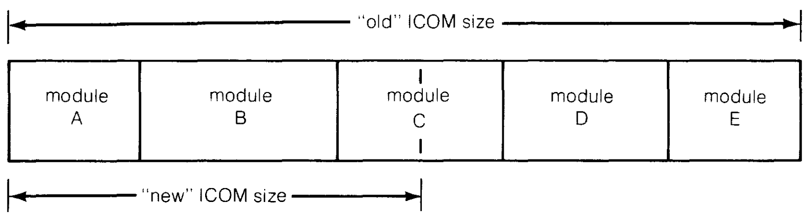
The IDELETE statement clears the ICOM region when executed, and is executed only when it is encountered in a program. Therefore, the deletion of the ICOM region can be avoided by starting or continuing execution at a point beyond the IDELETE statement.

When you are altering the size of the ICOM region, the new size specified becomes the size of the region from the moment of running the program. If the size being requested is larger than that which already exists, the additional space needed is requested from the operating system. If the space is available, everything proceeds uneventfully. If the space is not available, an error (number 2) results. To make the space available, one of the following procedures must be followed —

- Execute SCRATCH A.
- Execute SCRATCH C.

Each procedure has its separate effects, and the course selected should be determined by your circumstances at the time. Consult the Operating and Programming Manual for details on the other effects of each of these commands.

If the size being requested is smaller, modules are deleted if they no longer fit into the smaller region. For example, suppose the following situation existed —



Upon compilation of the new ICOM statement, the modules E, D, and C are deleted. None of those modules may contain an active interrupt service routine or an error results (number 193).

Retrieving and Storing Modules

Modules are stored in files on mass storage media as Option ROM (OPRM) or Assembly (ASMB) types of files. On tape media, they are stored in the OPRM type and on non-tape media they are stored in the ASMB type. In this case, the two file types are equivalent.[1]

To retrieve a module, or modules, from mass storage, identify the file name of the file containing the module. Combine the name with the mass storage unit specifier[2] of the device to form a file specifier. Then execute the statement —

ILOAD {file specifier}

This retrieves **all** the modules in the file and stores them in the ICOM region.

If there are modules already loaded in the ICOM region, these additional modules are added to them, (**not** written over them). If an existing module in the ICOM area has the same name as one of the modules being loaded, the existing module is deleted and the loaded version takes its place.

1 OPRM-type files may be created by other option ROMs for their particular purposes. In those cases, the contents are entirely different.

2 Not to be confused with the single-word msus described in Chapter 1. This form is used by BASIC's Mass Storage statements (see the Operating and Programming Manual or Mass Storage Techniques Manual).

If you do not want all the modules in a given file, you can purge the unwanted ones from the ICOM region using the IDELETE statement —

IDELETE {module name} [,{module name} [,...]]

For example, if you had loaded a file which had the routines Larry, Pat, Ed, and Piper, and you want to keep only Larry, then you execute the statements —

```
IDELETE Pat
IDELETE Ed
IDELETE Piper
```

or, more simply —

```
IDELETE Pat, Ed, Piper
```

Deletions do not have to be done immediately after loading. They can be done at any time. After the IDELETE has been executed, the portion of the ICOM region which the module previously occupied is made available for use in loading other modules. The space is NOT returned to the generally available memory; that action is done with an ICOM statement with a smaller size.

Whenever a module is deleted, other modules are moved, as necessary, to take up any slack space in the ICOM region. This is done so that all of the free space in the region is at the end. If a module is being deleted, or being moved as above, and it contains an active interrupt service routine, an error results (number 193).

No error results when an IDELETE statement is used to delete a non-existent module.

If you desire at any time to delete all of the modules in your ICOM region, you can do so by executing either of the following statements —

```
IDELETE ALL
IDELETE
```

IDELETE ALL is the most efficient method of deleting all modules.

Sometimes you may desire to move modules in the opposite direction — from memory to mass storage. This is done with the ISTORE statement. The statement has the form —

ISTORE {module name} [, {module name} [, ...]] ; {file specifier}

A {module name} must be the name of a module currently stored in the ICOM region. Upon execution of the statement, a file with the name and mass storage unit specifier given in the {file specifier} is created and the modules are stored in the file, in the order listed.

The file created by an ISTORE statement is an OPRM or ASMB type, as appropriate to the medium involved. It can then be used in ILOAD statements at a later time.

In the case that you might want to store all of the routines currently in the ICOM region into a particular file, you can use either of the following statements —

ISTORE ALL; {file specifier}
ISTORE; {file specifier}

NOTE

Executing a CONTROL STOP command during a module load, store or delete operation may clear the entire ICOM region.

Chapter 3
The Processor and the Operating System

Summary: This chapter contains the necessary information on the structure of the processor and the operating system. Topics covered are: machine architecture, memory organization, data structures, and the machine instructions.

Before proceeding to the actual assembly language, it is useful to discuss the processor and operating system with which you are dealing. This chapter discusses various concepts related to the processor, the machine instruction set, the operating system organization, and data structures.

Machine Architecture

The System 45 has two "hybrid" processors. For the purposes of assembly language, the two processors function together as a single unit. The hybrid consists of a Binary Processor Chip (BPC), an Input-Output Controller (IOC), and an Extended Math Chip (EMC). Each has its own set of instructions, but all three work in conjunction. It is not necessary in using the assembly system that you know on which chip a particular instruction resides. In the presentation of the instruction set — and for all practical purposes while working with the computer — no distinction need be made between the processors, and the entire instruction set may be considered as being resident on a single processor.

In addition to the processors, the hybrid also contains an I/O bus which is controlled by certain instructions. The I/O bus has an "address" part and a "data" part. Some of the instructions (it is indicated which ones) cause an "input cycle" to occur on the bus, which means that an address is given to the address part of the bus, and the data which appears on the data part is considered to be input. Other instructions cause an "output cycle", which means that the data is to be output to the given "address".

Figure 4 is a graphical representation of this architecture.

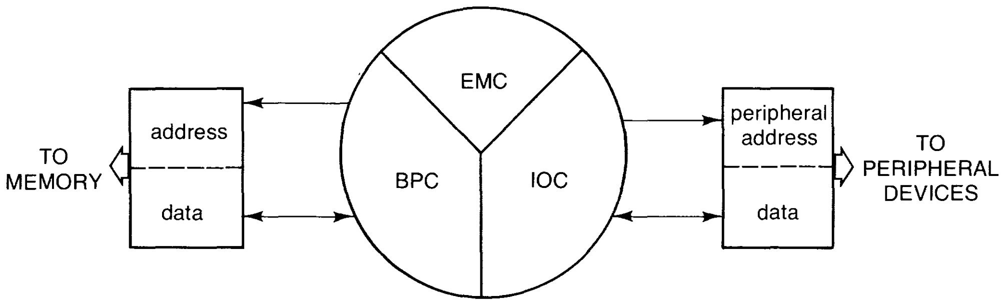


Figure 4. Generalized Machine Architecture

Registers

The memory locations in the machine are addressed from 0 to 177777B. There are 32 memory locations which are addressed as if they were part of the computer read/write memory, but actually are part of the processor. These locations are called "internal registers". Each internal register has a specific location and has been given a name. As you will learn in "Symbolic Operations" (Chapter 4), these names have been reserved and cannot be redefined while using the assembly system.

The internal registers are —

Name	Address (Octal)	Description
A	0	Arithmetic accumulator
Ar2	20-23	BCD arithmetic accumulator
B	1	Arithmetic accumulator
C	16	Stack pointer
Cb	13	Block bit for byte pointer in C (use most significant bit only, read only)
D	17	Stack pointer
Db	13	Block bit for byte pointer in D (use second most significant bit only, read only)
Dmac	15	DMA count register
Dmama	14	DMA memory address register
Dmapa	13	DMA peripheral address register (use lower 4 bits only)
P	2	Program counter
Pa	11	Peripheral address register (use lower 4 bits only)
R	3	Return stack pointer
R4	4	I/O (Input/Output) registers
R5	5	
R6	6	
R7	7	
Se	24	Shift-extend register (use lower 4 bits only)

Figure 5 is a map of where these registers lie. In addition to these registers, the addresses 25B through 37B are also registers, but are not (except for a few isolated cases) used in assembly programming.

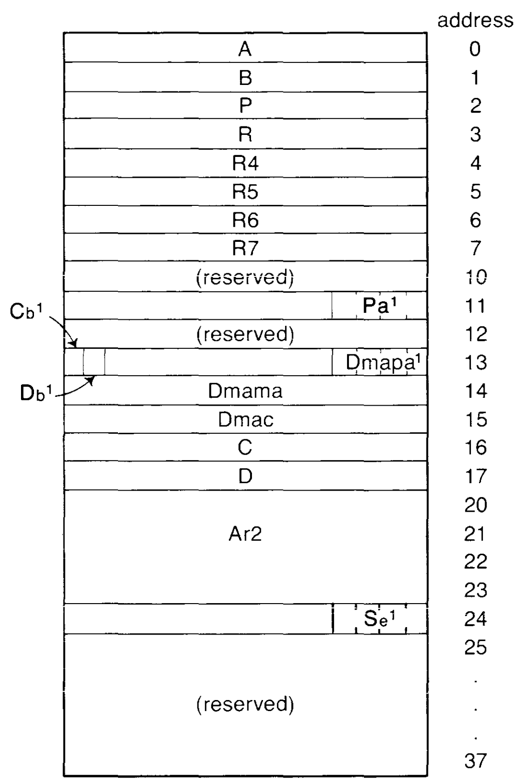


Figure 5. Map of Lowest Memory

All of these registers can be referenced either by their names or by their actual addresses. The two methods are equivalent, though reference by name is recommended as a programming practice.

1 See Chapter 8 for debugging considerations.

In addition to the above internal registers, there are some "external" registers which reside in the computer read/write memory. They are —

Name	Address (octal)	Description
Ar1	177770-177773	BCD arithmetic accumulator
Base_page	177645-177655	Base_page temporary area (9 words)
Oper_1	177656	Arithmetic utility operand address registers
Oper_2	177657	
Result	177660	Arithmetic utility result address register
Utltemps	177661-177665	Utility temporary storage area
Utlcount	177666	Used to create user utilities

General Memory Organization

In order to find your way around the machine effectively, you should be aware of where things are stored in memory. Occasionally these areas can become considerations in your programming.

First in the memory come the internal registers. They were discussed above.

Next in the memory comes the ICOM area. The starting location is dependent upon system needs, but never lower than 41B. The size of the ICOM region depends upon the size designated by the ICOM statement. Its maximum ending address is 77756B. This is the reason for the limitation on the size in the ICOM statement.

Next in the memory comes the area reserved for the system to store programs and the like. This area extends from the end of the ICOM region to 177644B.

This area is followed by the registers in the read/write memory (see the list in the previous section) with a number of interspersed system-reserved areas.

Figure 6 is a graphical presentation of this organization.

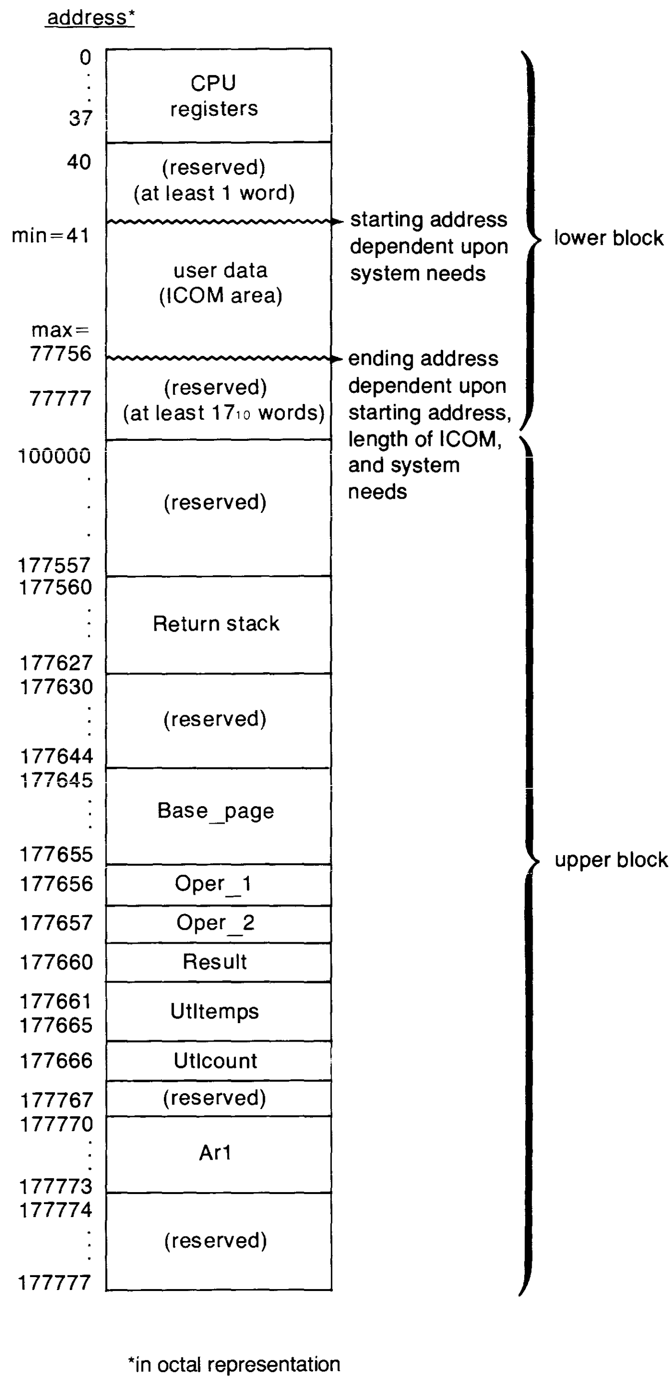


Figure 6. Memory Map

The immediately addressable memory consists of 65 536 words, which is all that can be addressed by a 16-bit word (the basic unit of memory in the system). Note that the memory is divided into two blocks — an "upper" block and a "lower" one. This distinction between blocks becomes significant when addressing individual bytes in memory.

Protected Memory

All of the reserved areas mentioned above are known as "protected memory". To give some measure of security to the operating system, it is advised that no attempt should be made to write or branch into these areas.

Access to certain portions of protected memory (e.g., BASIC variables) is provided by utilities within the assembly system. The user should access those areas only through the utilities.

Some measure of protection against access into these areas is provided during debugging. See the chapter entitled "Debugging" for a discussion of how this is done and the extent of the protection provided.

Base and Current Page

A concept that occasionally arises during discussion of the instructions and the assembler is that of the "page", the "base" and "current" pages in particular.

A page is 1 024 words of memory.

The "base" page is a wrap-around page. It consists of the upper half of the last page in the machine (addresses 177000B to 177777B) and the lower half of the zero page (addresses 0 to 777B). This is the same as a page which runs from − 512 to + 511, effectively "wrapping around" address 0.

During execution, the program counter (P) points to the address of the current instruction. The "current" page is those 1 024 words of memory centered upon the current instruction. Therefore, the current page is a continually changing page, extending from (P)− 512 to (P)+ 511.

Nesting of Subroutine Calls

Assembly language subroutines are called using the JSM instruction and exited using the RET instruction, both of which are described later in this chapter. Subroutine calls may be nested, just as they are in BASIC.

The JSM and RET instructions automatically adjust the R register (return stack pointer) so that the machine doesn't "lose its place" in the midst of subroutine calls and returns. The R register contains an address within an area of memory called the R stack, which is 40 words in size.

You are not free to use all 40 words in the R stack, however. The operating system and ICALL require 5 words. Interrupt service routines (refer to Chapter 7 for more information on ISRs) require 10 words. Break points (refer to Chapter 8 for more information on the IBREAK statement) require 5 words.

Thus, 20 words are left for the nesting of user JSMs. Calling system utilities also requires some of these 20 words. Appendix F, Utilities, contains the information necessary to determine the number of words needed by the various utilities.

For example, the following program segment illustrates the use of the R stack space –

```
10        .                    User R stack entries (after execution)
20        .
30     ICALL Nest
40        .
50        .
60     ISOURCE          SUB
70     ISOURCE Nest:    NOP                 0
80     ISOURCE          JSM Nest1           1
90     ISOURCE          RET 1               0
100       .
110       .
120    ISOURCE Nest1:   NOP                 1
130    ISOURCE          JSM To_system       2 to a max. of 7[1+(5+1)]
140    ISOURCE          NOP                 1
150    ISOURCE          RET 1               0
```

The system does not check for R stack overflow. Violation of the R stack limits could result in a machine lock up.

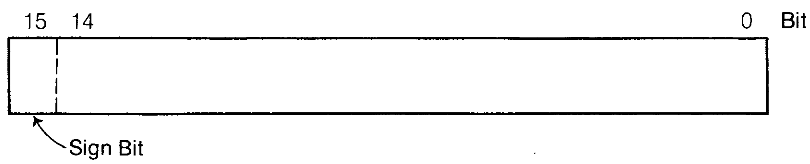
Data Structures

It is common to access BASIC variables from an assembly language routine then retrieve the contents, manipulate them, or alter them. To be effective at it, you should be aware of how BASIC stores a value in each of its data types.

There are four data types in BASIC: full-precision numeric values, short-precision numeric values, integers, and strings. Each is stored in its own unique structure.

Integers

The simplest of the types is the integer. (Variables are declared as integers using BASIC's INTEGER statement.) An integer consists of a single word. Values between − 32 768 and + 32 767 can be stored in the word. Negative values are stored in two's complement form. An integer looks like —



Strings

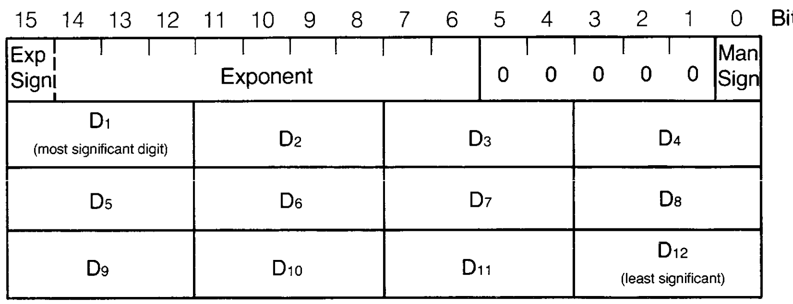
Strings are the next simplest structure. A string is a succession of bytes, one character to a byte. A string may be of variable length. To be able to designate the length, the string is preceded by a word which contains the number of bytes in the string.

If a string has an odd number of bytes in it, then the left-over byte in the word containing the last character of the string is wasted. A typical string of length n looks like —

n(length)	
byte 1	byte 2
byte 3	byte 4
byte 5	byte 6
⋮	
byte n-2	byte n-1
byte n	–

Full-Precision Numbers

Full-precision numeric values are stored as 12-digit, BCD (Binary Coded Decimal), floating point numbers. They occupy four words each. The first word contains the sign of the exponent, a two's-complement 10-bit exponent, and the sign of the mantissa. The other three words contain the twelve mantissa digits, 4 to each word. The words look like this —

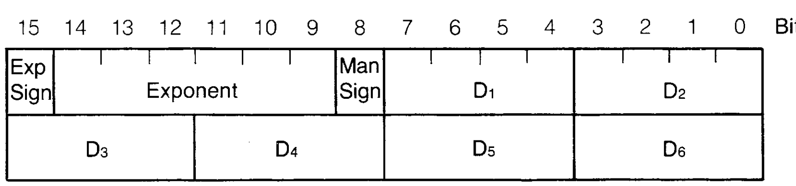


The exponent is always adjusted during arithmetic routines so that there is an implied decimal point following D_1. Thus, every mantissa value looks like —

$$D_1 \,.\, D_2\, D_3\, D_4\, D_5\, D_6\, D_7\, D_8\, D_9\, D_{10}\, D_{11}\, D_{12}$$

Short-Precision Numbers

Short-precision numeric values are stored as 6-digit, BCD floating point numbers. Unlike full-precision, they occupy two words each instead of four. The first word contains a 7-bit exponent, the sign of the mantissa and the two most significant mantissa digits. The second word contains the remaining four mantissa digits. The words look like this —



As with full-precision, the exponent is stored in two's complement form and the implied decimal point follows D_1.

If you are unfamiliar with BCD arithmetic or need a refresher in floating point operations, it is suggested that you refer to Chapter 5.

Machine Instructions

The machine instruction set underlying the assembly language system consists of 92 instructions, divided into eleven groups. The groups are —

Load/Store	Operations placing values into registers or memory.
Integer Math	Operations involving integer arithmetic.
Branch	Operations altering the execution sequence unconditionally.
Test/Branch	Operations altering the execution sequence, dependent upon some condition.
Test/Alter/Branch	Operations altering the execution sequence and a value, dependent upon some condition.
Shift/Rotate	Operations performing re-arrangments of the bits in the A or B register.
Logical	Operations performing logical functions on the A or B registers.
Stack	Operations managing stacks.
BCD Math	Operations involving Binary Coded Decimal arithmetic.
I/O	Operations specifically involving I/O operations.
Miscellaneous	Some unclassifiable operations.

Operands

Most instructions require operands. These operands have general forms which they may assume.

Many instructions contain an operand which is the address on which the function is to be performed. This {location} may be a constant (octal or decimal) or it may be a symbol. It also may be an expression containing any allowable combination of constants and symbols. For a full discussion of allowable expressions and symbols, and the "types" they are allowed to assume, consult "Symbolic Operations" in Chapter 4.

For example, note the operands in the following —

```
LDA 10B
STA Save
JMP Store + 3
AND = 177000B
```

A {location} may be either "relocatable" or "absolute" (see "Relocation" and "Symbolic Operations" in Chapter 4 for a full treatment of these types). If a relocatable {location} is used, the assembler generates machine code which uses "current page" addressing, and thus the {location} must be within − 512 words and + 511 words of the instruction. If an absolute {location} is used, the assembler generates machine code which uses "base page" addressing (meaning it takes the address as an offset from location 0).

An {address} is a {location} the same as above, except the intended location must be relocatable and within − 32 and + 31 words of the current instructions.

A {register} may be specified either through its absolute address or by its pre-defined symbol. The permissible registers are those with addresses between 0 and 7, inclusive. These are registers A, B, P, R, R4, R5, R6, and R7.

A number of instructions are followed by a {value}, which is a numeric expression usually in the range of 1 through 16. This {value} frequently indicates the number of bits involved in the operation. For example —

```
SAR 8
```

right-shifts the A register by 8 bits.

NOTE

Specifying the R4, R5, R6, or R7 registers (absolute locations 4 through 7) in an instruction causes an "I/O bus cycle" to occur. Consult Chapter 7, "I/O Handling", for the proper use of these registers.

Indirect Addressing

Some instructions may also employ "indirect addressing". This is indicated by including the optional indicator , I, such as —

```
LDA 10B,I
STA Save,I
JMP Store+3,I
```

There is only one level of indirect addressing provided with the processor. Of course, if further levels are desired, it is possible to implement them on your own. Some flagging scheme could be adopted, for example. One approach could be to adopt the policy that the sign bit (bit 15) of a word would indicate further indirection, with the remaining bits being the value. In such an approach, a load accumulator instruction would become two instructions —

```
10   ISOURCE   LDA A,I    ! Use current contents as pointer
20   ISOURCE   SAM *-1    ! If bit 15 set, indirection
```

Load/Store Group

This group of instructions allows transfers of data to take place. With the instructions below you can move information to and from the arithmetic accumulators (the A and B registers). You can also transfer the contents of one contiguous set of words in memory to another contiguous set.

Instruction	Description
LDA {location} [, I]	Loads register A with the contents of the specified location.
LDB {location} [, I]	Loads register B with the contents of the specified location.
STA {location} [, I]	Stores the contents of the A register into the specified location.
STB {location} [, I]	Stores the contents of the B register into the specified location.
CLR {value}	Clears (zeroes out) the specified number of words, beginning at the location specified by the A register. {value} must be an integer between 1 and 16.
XFR {value}	Transfers the specified number of words, from one location to another. The starting address of the location being transferred **from** must be stored in the A register. The starting address of the location being transferred **to** must be stored in the B register. {value} must be an integer between 1 and 16.

Integer Math Group

This group of instructions allows you to perform fundamental arithmetic operations on the contents of the arithmetic accumulators (the A and B registers).[1]

Instruction	Description
ADA {location} [, I]	Adds the contents of the specified location to the contents of the A register, leaving the result in A. If a carry occurs, the Extend flag is set in the processor. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag is set in the processor.
ADB {location} [, I]	Adds the contents of the specified location to the contents of the B register, leaving the result in B. If a carry occurs, the Extend flag is set in the processor. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag is set in the processor.
TCA	Performs a two's complement of the A register (i.e., one's complement, incremented by 1). If a carry occurs, the Extend flag in the processor is set. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag in the processor is set.
TCB	Performs a two's complement of the B register (i.e., one's complement, incremented by 1). If a carry occurs, the Extend flag in the processor is set. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag in the processor is set.
MPY	Binary multiply. Uses Booth's Algorithm. The values of the A and B registers are multiplied together with the product placed into A and B. The A register contains the least significant bits and the B register contains the most significant bits and the sign. (An anomaly in the processor results in an improper result whenever B equals − 32 768.)

1 A discussion of integer arithmetic techniques is found in the "Arithmetic" chapter of this manual.

Branch Group

This group of instructions allows you to alter the execution sequence unconditionally. It includes the "jumps" and "returns" from subroutines.

Instruction	Description
JMP {location} [, I]	Unconditionally branches to the specified location.
JSM {location} [, I]	Jumps to a subroutine. The value of the R register is incremented and the current value of the P register (i.e., the location of the JSM instruction itself) is stored into the address pointed to by the R register. Execution then proceeds to the specified location.
RET {value}	Returns from a subroutine. {value} is added to the contents of the address pointed to by the R register. The results are stored in the P register (i.e., specifying the next location for execution) and the R register is decremented. This is, in effect, a return from a JSM instruction to the instruction which is {value} instructions from the JSM itself. The "usual" return is RET 1. {value} must be an integer between −32 and 31.

Test/Branch Group

Similar to the Branch group, this group of instructions allows you to alter the execution sequence, but conditionally upon the result of some test. Most instructions involve tests on all or part of one of the arithmetic accumulators (the A and B registers), but a couple allow a test on a location in memory which you can specify, and a couple test the current activity of the CRT.

Instruction	Description
CPA {location} [, I]	Compares the contents of the A register with the contents of the specified location. Execution skips over the next word if the contents are not equal.
CPB {location} [, I]	Compares the contents of the B register with the contents of the specified location. Execution skips over the next word if the contents are unequal.
SZA {address}	Skips to {address} if register A is 0.
SZB {address}	Skips to {address} if register B is 0.
RZA {address}	Skips to {address} if register A is not 0.
RZB {address}	Skips to {address} if register B is not 0.
SIA {address}	Skips to {address} if register A is 0, then increments A regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
SIB {address}	Skips to {address} if register B is 0, then increments B regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
RIA {address}	Skips to {address} if register A is not 0, then increments A regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
RIB {address}	Skips to {address} if register B is not 0, then increments B regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
SHC {address}	Skips to {address} if CRT is scanning its raster.
SHS {address}	Skips to {address} if CRT is doing vertical retrace.

Test/Alter/Branch Group

Similar to the Test/Branch group, this group of instructions allows you to conditionally alter the execution sequence. In addition to tests, you can also alter the contents of the item being tested (such as set or clear a bit, or increment or decrement a register). Certain bits in the processor (Extend and Overflow) can be tested with some of these instructions, as well as registers and memory locations.

Some instructions may be followed by either of the following —

```
,S
,C
```

indicating that the bit being tested by the instruction will either be set (S) or cleared (C) after the test has been made.

Instruction	Description
ISZ {location} [,I]	Increment the contents of the specified location and skip execution of the next word if the result is 0.
DSZ {location} [,I]	Decrement the contents of the specified location and skip execution of the next word if the result is 0.
SAP {address} [,S] SAP {address} [,C]	Skips to {address} if the A register is positive or zero (bit 15 is 0).
SBP {address} [,S] SBP {address} [,C]	Skips to {address} if the B register is positive or zero (bit 15 is 0).
SAM {address} [,S] SAM {address} [,C]	Skips to {address} if the A register is negative (bit 15 is 1).
SBM {address} [,S] SBM {address} [,C]	Skips to {address} if the B register is negative (bit 15 is 1).
SLA {address} [,S] SLA {address} [,C]	Skips to {address} if the least significant bit of the A register is 0.
SLB {address} [,S] SLB {address} [,C]	Skips to {address} if the least significant bit of the B register is 0.

Instruction	Description
RLA {address} [,S] RLA {address} [,C]	Skips to {address} if the least significant bit of the A register is not 0.
RLB {address} [,S] RLB {address} [,C]	Skips to {address} if the least significant bit of the B register is not 0.
SOS {address} [,S] SOS {address} [,C]	Skips to {address} if the Overflow flag in the processor is set.
SOC {address} [,S] SOC {address} [,C]	Skips to {address} if the Overflow flag in the processor is cleared.
SES {address} [,S] SES {address} [,C]	Skips to {address} if the Extend flag in the processor is set.
SEC {address} [,S] SEC {address} [,C]	Skips to {address} if the Extend flag in the processor is cleared.

NOTE

The Extend and Overflow flags can be cleared only by using the SEC, SES, SOC, and SOS instructions with the ,C option.

Shift/Rotate Group

This group of instructions performs re-arrangements of bits in the arithmetic accumulators (the A and B registers). Circular and non-circular shifts are available.

Instruction	Description
SAR {value}	Shifts the A register right the indicated number of bits with all vacated bit positions becoming 0.
SBR {value}	Shifts the B register right the indicated number of bits with all vacated bit positions becoming 0.
SAL {value}	Shifts the A register left the indicated number of bits with all vacated bit positions becoming 0.
SBL {value}	Shifts the B register left the indicated number of bits with all vacated bit positions becoming 0.
AAR {value}	Shifts the A register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)
ABR {value}	Shifts the B register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)
RAR {value}	Rotates the A register right the indicated number of bits. Bit 0 rotates into bit 15 each time. (Right circular)
RBR {value}	Rotates the B register right the indicated number of bits. Bit 0 rotates into bit 15 each time. (Right circular)
RAL {value}	Rotates the A register left the indicated number of bits. Bit 15 rotates into bit 0 each time. (Left circular)
RBL {value}	Rotates the B register left the indicated number of bits. Bit 15 rotates into bit 0 each time. (Left circular)

Logical Group

This group of instructions performs logical (Boolean) operations upon the contents of an arithmetic accumulator (on A or B register). Logical "and" and "or" operations are available, along with complementing and clearing operations.

Instruction	Description
AND {address} [, I]	Logical "and" operation. The contents of the A register are compared bit by bit, with the contents of the specified location. For each bit-comparison a 1 results if both bits are 1's, a 0 results otherwise. The 16-bit result is left in A.
IOR {address} [, I]	Logical "inclusive or" operation. The contents of the A register are compared, bit by bit, with the contents of the specified location. For each bit-comparison, a 0 results if both bits are 0's, a 1 otherwise. The 16-bit result is left in A.
CMA	Performs a one's complement of the A register (i.e., bit-by-bit inversion of all 16 bits).
CMB	Performs a one's complement of the B register (i.e., bit-by-bit inversion of all 16 bits).
CLA	Clears register A. This instruction is identical to SAR 16.
CLB	Clears register B. This instruction is identical to SBR 16.

Stack Group

The Stack group of instructions provides you with operations for managing stacks. The instructions withdraw items from (also called "pop" or "pull") or push items onto a stack pointed to by either the C or D register. The items are pushed from or withdrawn into a specified register (other than C or D) and the C or D register is incremented or decremented appropriately.

Pushing instructions increment or decrement the C or D register prior to doing the pushing. Withdrawing instructions increment or decrement the C or D register after doing the withdrawal. Consequently, the pointer is always left pointing to the "top" of the stack after the operation.

Decrementing the C or D register is indicated by including `,D` after the operand. For "withdrawing" instructions, D is the default. For example, the following are equivalent —

```
WWC A,D
WWC A
```

Incrementing is specified by including `,I` after the operand. This is also the default for "pushing" instructions if neither I nor D is included. For example, the following are equivalent —

```
PWC A,I
PWC A
```

The instructions for pushing and withdrawing bytes require the ability to address bytes rather than words. This essentially multiplies the memory map by two, requiring an additional address bit. When using the byte oriented stack instructions, the Cb and Db registers provide an additional high order bit to the C and D registers, respectively. A typical set up for pushing items onto a stack is as follows:

```
ISOURCE          LDA=Buffer   ! Get buffer address.
ISOURCE          SAL 1        ! Shift it left.
ISOURCE          ADA =-1      ! Compensate for pre-increment.
ISOURCE          STA C        ! Put it into C register.
ISOURCE          CBL          ! Clear Cb register.
ISOURCE           .
ISOURCE           .
ISOURCE          PBC A,I      ! Push byte unto the stack.
```

A typical set up for popping items off the stack is as follows:

```
ISOURCE          LDA =Buffer  ! Get buffer address.
ISOURCE          SAL 1        ! Shift it left.
ISOURCE          STA C        ! Put it into the C register.
ISOURCE          CBL          ! Clear Cb register.
ISOURCE          WBC R4,I     ! Withdraw byte and output it.
ISOURCE          STA R7       ! Trigger output handshake.
ISOURCE           .
ISOURCE           .
```

Note the use of the CBL instruction in both cases.

One use of the push and withdraw byte instructions is for input and output operations involving strings. Manipulating byte stacks allows byte packing of character data. The first word of the string storage space can be cleared initially and incremented as each character comes in. At the end of the transfer, the first word of the string contains the string length, making the string BASIC compatible. Keep in mind that the push byte instruction increments first, then pushes. The lower bit of the C register determines whether the upper byte or the lower byte is addressed in the manner illustrated here –

C register least significant bit	Byte address
0	Upper
1	Lower

The character string "HELLO" appears in a byte-oriented stack upon input as illustrated here –

. . .	. . .
(string length)	
H	E
L	L
O	. . .
. . .	. . .

NOTE

When using the byte instructions (PBC, PBD, WBC, WBD), the address pointed to by the C or D register must not have an absolute address less than 40B.

Instruction	Description
PWC {register} , D PWC {register} [, I]	Pushes contents of {register} onto the stack pointed to by the C register.
PWD {register} , D PWD {register} [, I]	Pushes contents of {register} onto the stack pointed to by the D register.
PBC {register{ , D PBC {register} [, I]	Pushes the lower byte (right half) of {register} onto the stack pointed to by the Cb and C registers. If the least significant bit of C is a 1, the byte is placed in the lower byte of the word in the stack; if it is a 0, it is pushed into the upper byte.
PBD {register} , D PBD {register} [, I]	Pushes the lower byte (right half) of {register} onto the stack pointed to by the Db and D registers. If the least significant bit of D is a 1, the byte is placed in the lower byte of the word in the stack; if it is a 0, it is pushed into the upper byte.
WWC {register} [, D] WWC {register} , I	Withdraws a word from the stack pointed to by the C register and stores it into {register}.
WWD {register} [, D] WWD {register} , I	Withdraws a word from the stack pointed to by the D register and stores it into {register}.
WBC {register} [, D] WBC {register} , I	Withdraws a byte from the stack pointed to by the Cb and C registers and places it into the lower byte (right half) of {register}. If the least significant bit of C is a 1, the byte is withdrawn from the lower byte of the word in the stack; if it is a 0, it will be withdrawn from the upper byte.
WBD {register} [, D] WBD {register} , I	Withdraws a byte from a stack pointed to by the Db and D registers and places it into the lower byte (right half) of {register}. If the least significant bit of D is a 1, the byte is withdrawn from the lower byte of the word in the stack; if it is a 0, it is withdrawn from the upper byte.
CBL	Clears the Cb register (indicates lower block of memory).
CBU	Sets the Cb register (indicates upper block of memory).
DBL	Clears the Db register (indicates lower block of memory).
DBU	Sets the Db register (indicates upper block of memory).

BCD Math Group

This group of instructions provides you with BCD arithmetic operations using the Ar1 and Ar2 registers.

In general, the instructions associate the Ar1 register with "X" and the Ar2 register with "Y" in the mnemonic for the instruction. Both registers contain values which are considered BCD full-precision values when operated upon by instructions in this group.

The mantissas referred to below consist of 12 BCD digits. All the shifting operations manipulate the digits as units (i.e., 1 digit — or 4 bits — at a time). In addition, shifting operations involve an additional digit in the A register (located in the lower 4 bits, numbered 0 through 3).

All arithmetic is performed in BCD. The values being operated upon are assumed to be normalized BCD floating-point (full-precision) values. Signs and exponents are left strictly alone. There is a flag in the processor, called Decimal Carry, which is set when an overflow occurs during a BCD operation.

A full discussion of BCD arithmetic techniques can be found in Chapter 5.

Instruction	Description
MRX	Mantissa right shift on Ar1. The number of digits to be shifted is specified in the lower 4 bits (0-3) of the B register. The shift is accomplished in three stages — 1. The digit in bits (0-3) of the A register is right-shifted into the first digit of the mantissa, with the twelfth digit being lost. This is the first shift. 2. The mantissa digits are then right-shifted for the remaining number of digits specified. The twelfth digit, except for the last shift, is lost on each shift and the vacated digits are zero-filled. 3. Finally, the last right-shift takes place with the twelfth digit shifting into the A register. The Decimal Carry flag in the processor is cleared along with the upper 12 bits of the A register (4-15).

Instruction	Description
MRY	Mantissa right-shift on Ar2. The number of digits to be shifted is specified in the lower four bits (0-3) of the B register. The shift is accomplished in three stages — 1. The digit in bits (0-3) of the A register is right-shifted into the first digit of the mantissa, with the twelfth digit being lost. This is the first shift. 2. The mantissa digits are then right-shifted for the remaining number of digits specified. The twelfth digit, except for the last shift, is lost on each shift, and the vacated digits are zero-filled. 3. Finally, the last right-shift takes place, with the twelfth digit shifting into the A register. The Decimal Carry flag in the processor is cleared along with the upper 12 bits of the A register (4-15).
MLY	Mantissa left-shift on Ar2 for one digit. This is a circular shift, with the digit in bits (0-3) of the A register forming a thirteenth digit. The non-digit part of the A register is cleared (i.e., bits 4-15), and the Decimal Carry flag in the processor is cleared.
DRS	Mantissa right-shift on Ar1 for one digit. The twelfth digit is shifted into the A register (bits 0-3). The non-digit part of the A register is cleared (i.e., bits 4-15), and the Decimal Carry flag in the processor is cleared. The first digit in the mantissa is set to 0.
NRM	Normalizes the Ar2 mantissa. The mantissa digits are left-shifted until the first digit of the mantissa is non-zero, or until twelve shifts have taken place, whichever comes first. If the original first digit is already non-zero, no shifts occur. The number of shifts required is stored as the first four bits (0-3) of the B register. If twelve shifts were required, the Decimal Carry flag in the processor is set, otherwise it is cleared.
CMX	Ten's complement of Ar1. The mantissa of Ar1 is replaced with its ten's complement and Decimal Carry is cleared.

Instruction	Description
CMY	Ten's complement of Ar2. The mantissa of Ar2 is replaced with its ten's complement and Decimal Carry is cleared.
FXA	Fixed-point addition. The mantissas of Ar1 and Ar2 are added together, and the result is placed into Ar2. Decimal Carry is added to the twelfth digit. After the addition, Decimal Carry is set if an overflow occurred, otherwise Decimal Carry is cleared.
MWA	Mantissa word addition. The contents of the B register are added to the ninth through twelfth digits of the mantissa of Ar2. Decimal Carry is added to the twelfth digit; if an overflow occurs, Decimal Carry is set, otherwise it is cleared.
FMP	Fast Multiply. Performs the multiplication by repeated additions. The mantissa of Ar1 is added to the mantissa of Ar2 a specified number of times. The number of times is specified in the lower 4 bits (0-3) of the B register. The result accumulates in Ar2. If intermediate overflows occur, the number of times they occur appears in the lower 4 bits of the A register after the operation is complete. The upper 12 bits of the A register are cleared along with Decimal Carry.
FDV	Fast divide. The mantissas of Ar1 and Ar2 are added together until the first decimal overflow occurs. The result accumulates into Ar2. The number of additions without overflow is placed into the lower 4 digits of the B register (0-3). The remainder of the B register is cleared, as is the Decimal Carry flag in the processor.
CDC	Clears the Decimal Carry flag in the processor.
SDS {address}	Skips to {address} if Decimal Carry is set. Decimal Carry is a flag in the processor which may be set as the result of certain BCD arithmetic operations (see Chapter 5 for details).
SDC {address}	Skip to {address} if Decimal Carry is cleared. Decimal Carry is a flag in the processor which may be set as the result of certain BCD arithmetic operations (see Chapter 5 for details).

I/O Group

The I/O group of instructions provides you with some of the operations necessary to accessing peripheral devices through the I/O bus. In addition to the instructions contained here, there are instructions in other groups which can have I/O effects (e.g., LDA, STA...).

The techniques useful to the implementation of I/O operations using the instructions in this group and the other groups are discussed in Chapter 7.

Instruction	Description
SFS {address}	Skips to {address} if the Flag line is set (ready). The Flag line is associated with a peripheral on the current select code (see Chapter 7 for details).
SFC {address}	Skips to {address} if the Flag line is clear (busy). The Flag line is associated with a peripheral on the current select code (see Chapter 7 for details).
SSS {address}	Skips to {address} if the Status line is set (ready). The Status line is associated with a peripheral on the current select code (see Chapter 7 for details).
SSC {address}	Skips to {address} if the Status line is clear (busy). The Status flag is associated with a peripheral on the current select code (see Chapter 7 for details).
EIR	Enables the interrupt system. Cancels the DIR instruction.
DIR	Disables the interrupt system. Cancels the EIR instruction.
SDO	Sets DMA outwards. Directs that DMA operations read from memory, write to the peripheral.
SDI	Sets DMA inwards. Directs that DMA operations read from the peripheral, write to memory.
DMA	Enables the DMA mode. Cancels the DDR instruction.
DDR	Disables Data Request. Cancels the DMA instruction.

Miscellaneous

The following instructions cannot be classified into any of the other groups.

Instruction	Description
NOP	Null operation. This is exactly equivalent to LDA A.
EXE {value} [, I]	The contents of any register can be treated as the current instruction and executed. {value} is a numeric expression in the range 0 through 31, indicating the register to be used. The register is left unchanged, unless the instruction code causes it to be altered. The next instruction to be executed is the one in the word following the EXE, unless the code in the executed register causes a branch.

Chapter 4
Assembly Language Fundamentals

Summary: This chapter discusses some of the basic statements and syntaxes used throughout the assembly language system. Program entry, assembling, symbolic operations, module creation, program and variable storage, and utilities are the topics covered.

When writing assembly language programs there are a number of things with which you will be involved constantly. In the beginning, questions arise on how to use the language: How do you enter the source code? What kind of symbolic addressing is there? How do you create and distinguish modules? How do you create the object code and where is it stored? What utilities are available and how do you use them?

The answers to those questions form the underlying capabilities through which you write your applications. These are things which nearly every assembly language program uses. As essential as they are, however, none are difficult to master.

Program Entry

You were introduced early in Chapter 2 to the integrated nature of the assembly language with its host language, BASIC. You know from that chapter how assembly language statements can be intermingled with BASIC statements — that you can employ the usual editing features on the assembly statements. However, there is more to the ISOURCE statement than just its integrated nature with BASIC.

As stated in Chapter 2, all assembly language statements are designated with the keyword "ISOURCE". The keyword is followed by {assembly language source}. So the syntax of the entry line is —

{line number} [{BASIC label} :] ISOURCE {assembly language source}

Here's a simple example of this from Chapter 2 —

```
40   ISOURCE   NAM Example
50   ISOURCE   NOP
60   ISOURCE   END Example
```

The {line number} and {BASIC label} are the same as you are used to in BASIC. However, it should be noted that the statement is not an executable one, so the BASIC label is only useful for documentation and EDIT purposes.

To BASIC, the ISOURCE statement appears as a comment. If you were to change the above so that it read —

```
40 Example:  ISOURCE   NAM Example
50           ISOURCE   NOP
60           ISOURCE   END Example
70           END
```

and then executed a statement "GOTO Example", the result would be to simply execute the END statement in line 70. That is because, to BASIC, the lines appear the same as —

```
40 Example:  REM
50           REM
60           REM
70           END
```

or —

```
40 Example:  !
50           !
60           !
70           END
```

The BASIC label on an ISOURCE line finds its most useful characteristic in being able to be referenced, as any other BASIC label on any other type of line may be, with an EDIT command. Thus, if you were to execute —

```
EDIT Example
```

on the above, you would be working in the editor, starting with line 40. This feature will become useful during program development as will be pointed out shortly.

Assembly Language Source

You may have recognized the assembly language instruction and pseudo-instructions to the right of ISOURCE in the examples above. This is where your instructions and pseudo-instructions appear. However, the source is a little more versatile than that. In general, {assembly language source} has the syntax —

[{label} :] {action} [! {comment}]

Or, the action may be omitted and only a comment appears —

[{label} :] ! {comment}

A label is always optional in the source, but either an {action} or a {comment} must be present in every source line.

Actions

An {action} in assembly language source is —

- A machine instruction, with any operand it may require. These were discussed at some length in Chapter 3.
- A pseudo-instruction, with any operand it may require. These are discussed under the topics to which they relate.

The actions contained in the above example were —

```
NAM Example
NOP
END Example
```

Labels

The {label} in assembly language source is part of the symbolic addressing capability of the assembler. This {label} is used by the **assembler** only. Neither the operating system nor BASIC is aware of its existence.

The label follows the same form and rules as do labels in BASIC —

- Up to 15 characters long.
- First character must be a capital letter (A-Z).
- Only the non-capital letters (a-z), the numerals (0 to 9), or the underscore (_) may be used following the first character.

No two labels are allowed to be the same in a given **module**. If your source consists of two or more modules, then the same label may be defined more than once, provided each definition is in a different module. (Distinguishing between modules is discussed in "Creating Modules", later in this chapter.) So you may not code —

```
Rumpelstiltskin: LDA B
```

in one place in the module and later in the same module code —

```
Rumpelstiltskin: LDB A
```

There are other restrictions as well on the choosing of labels. For instance, there are symbols already defined by the assembler and you are not allowed to choose one of them as a label. This is discussed at length in "Symbolic Operations" in this chapter.

Both a BASIC label **and** an assembly language source label can appear in the same line, and they are distinct from one another. BASIC does not know about the source label and the assembly language system does not know about the BASIC label.

Since neither BASIC nor the operating system is aware of the existence of source labels, actions ouside the assembler cannot reference these labels. Thus, if you had the source line —

```
100  ISOURCE Rumpelstiltskin:  JMP Bail
```

You can say neither GOTO Rumpelstiltskin nor EDIT Rumpelstiltskin. Neither of these can find "Rumpelstiltskin", since only the assembler can know it is there.

This can be a nuisance in some instances during program development. Many programmers use labels almost exclusively and rarely consider the line number when using the editor to change a line. For instance, in the above, they would not be used to saying, "EDIT 100" to get at the line in order to change it. They are more used to saying, "EDIT Rumpelstiltskin". A way for them to do it would be to change the line to —

```
100 Rumpelstiltskin:  ISOURCE Rumpelstiltskin:  JMP Bail
```

Note that, as the example demonstrates, the name can be the same in the BASIC label as in the source. This takes advantage of the fact that BASIC and the assembler are unaware of each other's labels. The names do not have to be the same.

Comments

As with any BASIC line, a comment may be included by simply adding an exclamation point (!) and typing your comment after it. Since you have a total of 160 characters for a line, your comment may fill up the remainder of the 160 characters left after the rest of the statement has been provided (line number, ISOURCE keyword, label, action).

Syntaxing the Source

When you are creating your source program, you are either entering it from the keyboard or retrieving it from mass storage (LINK or GET). In either case, as the statement is entered (the STORE key on the keyboard is pressed or a record is read from mass storage), the operating system takes note of any use of the keyword ISOURCE. When a line has this keyword, the operating system turns over the remainder of the line following the keyword to the assembly system. The assembly system, then and there, checks the syntax of the source.

By checking the syntax at the time of entry of the statement, a considerable amount of processing time is saved when the time comes to assemble the source into object code. In addition, it gives you, as the programmer, immediate feedback when a syntactical error occurs. You do not have to wait until assembly time just to find out that you misspelled NOP.

At syntax time, the assembler takes care of capitalization, lower case, and spacing for the source. It's quite similar to the SPACE DEPENDENT mode of entry for BASIC statements (that mode is not required to get the effect with the assembly system). It follows the following rules in syntaxing the source —

- Everything between the ISOURCE and the colon (if present) is the label. Its initial character is capitalized and the remaining letters are converted to lower-case. This is regardless of whether they were entered in that form.
- The label, if present, is left-justified to the second column following the keyword ISOURCE.
- The first three letters following the colon (or just the first three letters, if there is no label) are considered the machine instruction or pseudo-instruction and are capitalized. The instruction will remain in the same column as it was entered, and, if possible, a space is added after it.
- Everything after the instruction or pseudo-instruction is considered the operand for the instruction, up until the exclamation point before the comment (if any). Any label (symbol) in the operand will have its initial character capitalized and the remaining letters converted to lower case automatically.
- Comments are unchanged and remain in the same columns as entered, whenever possible.

In short, simply enter the statement in your most comfortable fashion and the assembly system automatically assures that what you enter is in the proper form (though it still can't guarantee that you have entered the right instruction for what you mean to do).

As a demonstration of this facility, consider the following line ready for syntaxing —

```
100   ISOURCE      rUMPELSTILTSKIN:jMpbail
```

It becomes —

```
100   ISOURCE Rumpelstiltskin:     JMP Bail
```

Creating Modules

When you were introduced in Chapter 2 to the concept of a module, it was said that a module is given a name through the NAM pseudo-instruction.

So, when you enter a source line which has the following form —

NAM {module name}

you are assigning a name to a module, and you are also delimiting the beginning of the module. By the inclusion of this statement, all source lines which follow are part of the module with the name designated in this source line, that is, all lines until the END pseudo-instruction is encountered in the source. It has the form —

END {module name}

Its {module name} must be the same as in the NAM pseudo-instruction.

A {module name} follows the same rules for naming as do labels.

It is by the use of these two instructions that modules are created. The source lines which appear between them comprise a single module, and the name assigned to the module is the one with which the module is referenced (with the ILOAD and ISTORE statement for example).

When it comes time to assemble the source into object code, the assembler treats the source lines in a module as a unit.

In actuality, therefore, there are **two** modules — a source module and an object module. When you are assembling a module, the name you use refers to the source module and creates the object module. Later, other statements, such as ISTORE and ILOAD, refer solely to the object module.

Storage

Modules

When assembly converts a source module into an object module, there must be a place to keep the object module. That is the function of the ICOM region.

You were introduced to the ICOM region in Chapter 2 in connection with the loading and storing of modules. It is also used to hold modules which are created through assembly. Once a module has been assembled, the object code appears in the ICOM region just as if you had loaded it from mass storage.

Variables

Within a module, you may want to set aside one or more words of memory for your use. For example, you might need a location to store a variable, or keep a counter, or save a register. This is done with the BSS pseudo-instruction —

```
BSS {number}
```

where {number} is the number of words to be set aside. {number} can be any absolute expression, provided the expression evaluates to a positive integer (see "Symbolic Operations", later in this chapter).

This kind of storage is part of the object code and is set aside "in-line". This means that wherever it appears in the source, the storage appears in the same relative location in the object module.

For example, suppose a module contained the following source lines —

```
          .
          .
          .
220   ISOURCE Save_a:  BSS 1
230   ISOURCE Save_4:  BSS 2*2
240   ISOURCE Renras:  BSS Larry
250   ISOURCE Again:   LDA Renras
          .
          .
          .
```

Then, at some appropriate spot in the object module (relative to the other instructions in the module) there would be the following **contiguous** locations —

Save_a	1 word
Save_4	4 words
Renras	some number of words equal to "the absolute symbol, Larry"[1]
Again	1 word

The locations at labels Save_a, Save_4, and Renras are merely reserved by the BSS pseudo-instructions, and their contents are not initialized to any particular value.

It is possible to accidentally execute these locations when the routine is run if you're not careful. Ordinarily, you should place these locations somewhere safely out of the potential execution sequence, since they are used just for storage. Some applications, though, use self-generating code, and a BSS is a way to set aside locations for it.

Data Generators

A "data generator" is very much like a BSS operation. The function, as with the BSS, is to set aside words of memory at a particular location in the object code. But in addition, the words are to be initialized to some value. The initialization occurs at the same time the words are set aside (i.e., at assemble-time).

This is done using the DAT pseudo-instruction which has the form —

DAT {expression} [, {expression} [, ...]]

An {expression} may be any absolute or relocatable expression. The various forms that an expression may take are discussed in "Symbolic Operations" later in this chapter.

As an example, suppose you want the value 100 (a decimal integer) to be located at location "X" in the object module. You can achieve this by identifying the location in the source code (ultimately the object code) where you want the value to be, then placing this instruction at that point —

```
X:  DAT 100
```

1 Such symbols are discussed at length in the "Symbolic Operations" section later in this chapter.

Upon encountering this pseudo-instruction, the assembler generates the words necessary to store the value (in this case, only 1 word is necessary). It then stores the value (100) into the word(s) and proceeds with the remaining assembly. Thus, the location of the words is dependent upon the instruction's relative position in the source module, the same as with any machine instruction.

The number of data words generated for each {expression} is dependent upon the result of the {expression} —

Result	Words
Full-precision	4
Short-precision	2
Decimal integer	1
Octal integer	1
Address[1]	1
Literal	1
String	actual length (2 characters per word)

If more than one {expression} is present, the necessary data words are generated in the order in which they appear in the list. As an example, if you were to include the instructions —

```
ISOURCE   Integers:      DAT 24,24B
ISOURCE   Real:          DAT 2.4E1,-2.4E5
ISOURCE   Short:         DAT 2.4E6S,4.567S
ISOURCE   String:        DAT "HELLO"
ISOURCE   B_string:      DAT 5,"HELLO"
ISOURCE   Character:     DAT ´C
ISOURCE   First_word:    DAT Buffer
ISOURCE   Buffer:        BSS 10
ISOURCE   Last_word:     DAT *-1
ISOURCE   Addr_of int:   DAT =3
ISOURCE   Addr_of_3_int:DAT =3,4,5
```

1 including "external"

Forty words would be set aside and initialized to the appropriate values —

```
000041: 000030   decimal integer 24
000042: 000024   octal integer 24
000043: 000100
000044: 022000   full-precision 2.4 E1
000045: 000000
000046: 000000
000047: 000501
000050: 022000   full-precision -2.4 E5
000051: 000000
000052: 000000
000053: 006044   short-precision 2.4 E6
000054: 000000
000055: 000105   short-precision 4.567
000056: 063400
000057: 044105
000060: 046114   "HELLO" string
000061: 047400
000062: 000005
000063: 044105   BASIC "HELLO" string¹; first value (5) is character count
000064: 046114
000065: 047400
000066: 000103   "C" character
000067: 000070   Address of first word in ten word buffer
000070: 004000
000071: 167356
000072: 100040
000073: 002000
000074: 167312   Ten word buffer (values are meaningless)
000075: 100040
000076: 002000
000077: 167246
000100: 100020
000101: 004000
000102: 000101   Address of last word in ten word buffer
000103: 000105   Address of word containing integer value 3
000104: 000106   Address of first word of an area containing three integers
000105: 000003   Integer value 3
000106: 000003   Integer value 3, first word in a group of three words
000107: 000004   Integer value 4
000110: 000005   Integer value 5
```

1 BASIC strings must be generated for communication between BASIC and assembly language as brought about through the use of the Put_value (Chapter 6) and the Print_string (Chapter 7) utilities.

Repeating Instructions

To help relieve the tedium of writing the same instruction many times (which many applications occasionally require), a "repeat" pseudo-instruction is provided —

REP {expression}

The pseudo-instruction causes the immediately following machine instruction to be duplicated in the object code {expression} number of times.

For example, suppose you are writing a real-time application where timing was critical, and to make things work correctly you need 10 NOPs at a certain location. Ordinarily you would type —

```
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
ISOURCE   NOP
```

But all of this could be replaced with —

```
10   ISOURCE   REP 10
20   ISOURCE   NOP
```

and the same effect would be achieved.

Some pseudo-instructions may not be replicated. They are —

```
COM
END
EQU
EXT
NAM
REP
SUB
```

Assembling

Object code is created by "assembling" the source code. Again, modules are a key factor. The assembly directive is aimed at modules, using the module name as a delimiter in the source code so the assembler can tell which ISOURCE statements to assemble as part of the module. Of course this same name is also used to store the object code using mass storage.

The IASSEMBLE statement is the vehicle for assembling modules. It has the forms —

```
IASSEMBLE {module} [ , {module} [ , ...] ] [ ; {option} [ , {option} [ , ...] ] ]
IASSEMBLE [ ALL ] [ ; {option} [ , {option} [ , ...] ] ]
```

Each {module} indicated is assembled, in the order given by the statement. Only those modules are assembled; any others which may be present in the source at the time are ignored. If the ALL version of the statement is used (with or without the optional word ALL), every module present in the source is assembled.

An {option} falls into one of two categories: listing directives and conditions (for conditional assembly). These are discussed separately below. The options, and their categories, are —

EJECT	
LINES	Listing directives
LIST	
XREF	
A	
B	
C	
D	Conditions
E	
F	
G	
H	
IOF	
ION	Control of indirection

References to multiple-line functions cannot appear in the IASSEMBLE statement. If an IASSEMBLE ALL statement is executed and no source code is present, no error message is given.

Effect of BASIC Environments

To assemble a module, all of its source lines (between the NAM and END pseudo-instructions) must lie within the same BASIC "environment". That is, the NAM and END for a module must lie within the main program or within the same subprogram or multi-line function. For modules where this is not true, an error ("EN" assemble-time error) occurs.

Source Listing Control

Listings of the source code in a module can be obtained during an assembly. These listings contain the line numbers, instructions, and comments from the source lines along with the associated machine addresses and contents of that address.

Here is part of a typical listing —

```
430  01034 002645   LDA =Array_type
440  01035 006645   LDB =Array
450  01036 142645   JSM Get_info     !Info on the array
460  01037 003005   LDA Array_type   !Look at the type
470  01040 012644   CPA =16          !Is it a file number?
480  01041 066003   JMP *+3          !Must be a file number
490  01042 022643   ADA =-12         !Is it an array data
500  01043 172003   SAP *+3          !  type (ie, >12)?
```

line numbers | absolute addresses | contents | actions | comments

The addresses and contents are displayed in octal representation.

Listings are not automatic. They are obtained in one of two ways —

- By using the LIST option in the IASSEMBLE statement. This directs that a listing is desired for all the modules in the statement. The statement would look like the following examples —

```
IASSEMBLE Store;LIST
IASSEMBLE Retrieve,Work;LIST
```

- By using the LST pseudo-instruction in the source code itself.

Modules can be just partially listed, if desired. This kind of control is achieved by using the LST and UNL pseudo-instructions within the source code, placing the LST before any instructions which you want listed, and placing the UNL before any instructions you do not want listed. For example, if the following source lines are assembled —

```
420  ISOURCE   LST
430  ISOURCE   LDA =Array_type
440  ISOURCE   LDB =Array
450  ISOURCE   JSM Get_info        !Info on the array
460  ISOURCE   LDA Array_type      !Look at the type
470  ISOURCE   CPA =16             !Is it a file number?
480  ISOURCE   JMP *+3             !Must be a file number
490  ISOURCE   ADA =-12            !Is it an array data
500  ISOURCE   SAP *+3             !  type (ie, >12)?
510  ISOURCE   UNL
```

only lines 430 through 500 would be listed.

The primary purpose of this capability is to allow as much modularity in the listings as you can get in source code. To implement this purpose, a "listing counter" is used.

Whenever an LST instruction is encountered during an assembly, the listing counter is incremented. Whenever an UNL instruction is encountered during an assembly, the listing counter is decremented. Source lines are listed whenever the counter is greater than 0. Whenever it is equal to 0 or negative, then no lines are listed.

The counter is set to 0 upon execution of the IASSEMBLE statement. This is why there is no automatic listing. However, if the LIST option is included in the IASSEMBLE statement, then the counter is initialized to 1. This is why that option creates a listing. Thus, you could defeat a LIST option by placing an UNL instruction at the beginning of a module. This initialization process occurs for each module assembled, so if you have more than one module indicated in your IASSEMBLE statement, the counter is set at the beginning of the assembly for each.

This capability sees its greatest usefulness during debugging stages and while working with independently written sections of source code. For example, a number of people could be writing different sections of code, each containing their own LST and UNL instructions. These instructions could then be overridden when they were combined into a single module by preceding the sections with an LST instruction (to get a listing) or an UNL (to suppress the listings).

Page Format

Each and every assembly listing page has the following format —

- The word "PAGE" and the current page number of the listing occurs on the first line starting at column 49.
- A heading occurs on the second line, left-justified. The heading always includes —

  `MODULE:` {name}

  where {name} is the name of the module currently being assembled. Additional heading information can be specified for this line (see "Page Headings" below).
- A blank line follows the heading.
- The text follows the blank line. The number of lines printed depends upon the LINES option in the IASSEMBLE statement, the number of source lines encountered, and the SKP pseudo-instructions which may be encountered while assembling the source. LINES and SKP are described in the following sections.
- If the EJECT option is **not** included in the IASSEMBLE statement, then a minimum of three blank lines (carriage return/line feed pairs) will be printed at the end of a page. The number may exceed three if the number of source lines printed on a page is less than the standard length for a listing page.

Page Length

The length of the text in each page of your assembly listings can be specified through the IASSEMBLE statement using the LINES option, which has the form —

`LINES` {numeric expression}

This option directs that any listing of the modules being assembled have pages of the length indicated by the absolute value of {numeric expression}. If {numeric expression} evaluates to a positive number, the listing for each module is printed on a separate page with the indicated number of lines. If {numeric expression} evaluates to a negative number, the pagination at the end of each module listing is suppressed. An error is generated if {numeric expression} evaluates to zero.

It is not necessary that this value be the page length of the printing device being used; however, this is frequently the value selected. If the option is omitted, a value of 60 is used, producing an overall page size of 66 lines.

Printer control characters, such as line-feed and form-feed, in a comment can affect the actual printing length of the pages independent of the length you specify. Thus, a page length of 60 could result in actually 61 lines if one of the comments in your ISOURCE statements contains a line-feed character.

End-of-Page Control

At any time during the assembly of a module, you can force the listing to continue printing at the top of the next page by including —

```
SKP
```

at the desired spot in the module. If a listing is being generated when this pseudo-instruction is encountered in the source code, the printer is sent to top-of-form. This is physically done in one of two ways —

- If the EJECT option was included in the IASSEMBLE statement which is assembling the module, then a form-feed character (ASCII character 14B), is sent to the printer. This feature is intended for perforated paper.
- If the EJECT option was not included, sufficient CR/LF pairs (ASCII characters 15B and 12B) are sent to the printer to fill out the standard length of a listing page (plus three at the end of the page). Thus, if you already have printed 10 lines on a page, and an SKP instruction was encountered, the assembler sends (length − 10 + 3) CR/LF pairs. This feature is intended for non-perforated paper.

The SKP instruction is not required to cause pagination to occur when the standard length of a listing page is exceeded. Thus, if you are working with a default length of 60 for your standard length, then each 60 lines from the last page break forces a new page break.

Page Headings

The heading for each listing page is —

```
MODULE: {name}
```

where {name} is the name of the module currently being assembled. This heading can have additional information added to it through the HED pseudo-instruction. This instruction has the form —

```
HED {comment}
```

When this instruction is encountered, and a listing is being generated, pagination immediately occurs, the same as with the SKP instruction (see above). On the new page, and on all pages after it, the indicated {comment} appears after {name} in the heading, replacing any previous information specified by an earlier HED instruction.

You can change the heading any number of times in a listing. This is frequently done in order to generate documentation by sections, even though all sections may reside in a single module.

The heading appears on the page exactly the same as in {comment}, including the positioning of blanks, control characters, etc.

Blank Line Generation

If occasional blank lines are desired in a listing (usually to set off sections of code, or comments), they may be generated by including —

```
SPC {number}
```

at the desired spot in the source statements. {number} designates the number of blank lines desired. {number} can be any absolute expression, provided the expression evaluates to a positive integer (see "Symbolic Operations" later in this chapter).

Non-Listable Pseudo-Instructions

The following pseudo-instructions do not appear in a listing —

```
LST
UNL
SKP
HED
SPC
```

Conditional Assembly

For reasons of complexity or length, it is occasionally desirable to selectively assemble only parts of a module. This is particularly true during the debugging stage of longer, complex assembly programs. "Conditional assembly" is the ability to designate certain portions of a module for assembly, depending upon conditions established by the IASSEMBLE statement.

You may recall from the description of the IASSEMBLE statement earlier, there are options called "conditions" available with the statement. These conditions —

```
A
B
C
D
E
F
G
H
```

are used to designate which conditions are "set" during the assembly. By including one or more of these conditions, all conditional assembly statements predicated upon that condition are assembled. For example, if the following statement is executed —

```
IASSEMBLE Retrieve; A
```

then any occurrence of conditional assemblies based on "A" are assembled. Also, any conditional assemblies based on B through H are not assembled, since those conditions were not included in the options for the IASSEMBLE statement.

The conditional assembly sections are delimited by pseudo-instructions. A conditional section begins with one of the following —

```
IFA
IFB
IFC
IFD
IFE
IFG
IFH
```

and it concludes with —

```
XIF
```

In addition to the lettered conditions, a numeric condition can be tested by using an IFP pseudo-instruction. It has the form —

`IFP` {absolute expression}

The condition is considered true if {absolute expression} evaluates as a positive value. It should be noted that this is an assembly-time construct, meaning that the variables contained in the expression are evaluated at the time of assembly.

The IFP instruction performs in the same manner as the IFA through IFH instructions. It also terminates with the XIF instruction.

The conditional assembly is based upon a flag. At the beginning of the assembly for a module the flag is set so that object code is generated for all instructions. An IF conditional encountered during the assembly which does not have its condition set turns off the flag so that no further code is generated. Encountering an XIF statement resets the flag so that code generation can resume. For instance, if the source —

```
300   !
310   !
320 IASSEMBLE Retrieve;A
330   !
340   !
350 ICALL Begin
360   !
370  ISOURCE          NAM Retrieve
380  ISOURCE           !
390  ISOURCE          SUB
400  ISOURCE           !
410  ISOURCE           !
430  ISOURCE Begin:   LDA =Array_type
440  ISOURCE          LDB =Array
450  ISOURCE          JSM Get_info      ! Info on array parameter
460  ISOURCE          LDA Array_type    ! Look at the type
470  ISOURCE          IFA                 !
480  ISOURCE          STA Test            ! Debugging section
490  ISOURCE          RET 1               !
500  ISOURCE          XIF                 !
510  ISOURCE          IFB
520  ISOURCE          CPA =16           ! A file number (not an array)?
530  ISOURCE          JMP *+3           ! Must be a file number
540  ISOURCE          ADA =-12          ! Is it an array
550  ISOURCE          SAP *+3           !    data type(i.e., >12)?
560  ISOURCE          XIF
570  ISOURCE          LDA =Test
640  ISOURCE           !
650  ISOURCE           !
660  ISOURCE          END Retrieve
670  !
680  !
690  IASSEMBLE Retrieve
```

is executed, lines 430 through 460, 480, and 490 are assembled, but 520 through 550 are not. Line 570 is assembled.

The XIF pseudo-instruction actually affected both conditions. This effect is more dramatically illustrated if line 320 is changed to —

```
IASSEMBLE Retrieve
```

where neither A nor B is set. In this case 480, 490, 520 through 550 are not assembled. But 570 is assembled!

The effect of the XIF, then, is as a flag for all the conditions. As a consequence, it is not possible to "nest" conditional assemblies. This effect is the same with the IFP conditional.

Control of Indirection

The assembler can generate an indirect instruction, even when you have not specified a ,I after the instruction. The pseudo-instructions IOF (indirect off) and ION (indirect on) control these automatic indirects. While automatic indirection is turned off (by IOF), a range error (RN) is generated for any instruction which the assembler would have generated an automatic indirect for. ION turns automatic indirection back on, restoring the assembler to its normal state. These pseudo-instructions are used in pairs, with IOF first and ION last, to specify an interval for which you wish to control automatic indirection.

Relocation

The code talked about in this section is relocatable. You do not have to worry about the absolute location of your module. The assembler automatically generates the appropriate machine codes for each of your instructions to assure that the correct location is reached when referenced.

Some instructions generate relocatable object code in which the operand address is an offset from the current address and the relocating loader has to make no changes to the object code for them as long as they are within − 512 and + 511 of the current address.

For indirect addressing, and for instructions which are more than 512 words away from the current address, it is required of the loader to adjust the address in the intermediate word to reflect the actual address being referenced. For indirect addressing generated by the assembler, this activity is automatic.

Some instructions permit you to specify an absolute machine address for its operand. In those cases, the assembler generates the code necessary to perform the reference to the absolute location.

For example, if the instruction was assembled —

```
LDA B
```

(which essentially says "load register A with the contents of register B) the result would be a machine instruction which references the B register (absolute address 1). This reference would be independent of the actual location of the instruction itself.

There are a couple of ways to produce an absolute address in an operand. Using pre-defined symbols is one way. There is a type of expression known as "absolute" which is another way. Both of these are dicussed in the next section, "Symbolic Operations".

You should never try to use absolute addressing within the ICOM region, since not only is the location of the region itself not fixed, but modules can also be moved around within the region.

Module Reassembly

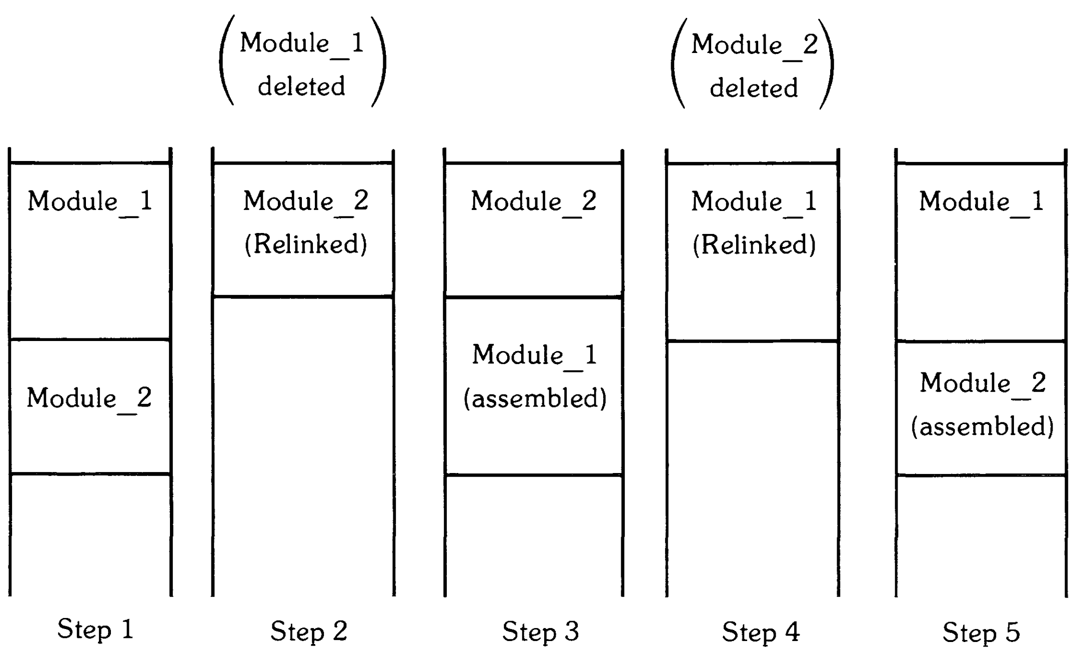
Modules that have been assembled can be reassembled at any time. Debugging a routine often times leads to changes and reassembly. A discussion of this process is in order.

The steps involved in the reassembly of two modules with the statement —

```
IASSEMBLE Module_1, Module_2
```

are the following:

- Step 1 – both modules appear in their original positions in the ICOM region.
- Step 2 – Module_1 is deleted and Module_2 is moved and linked.
- Step 3 – Module_1 is assembled.
- Step 4 – Module_2 is deleted and Module_1 is moved and linked.
- Step 5 – Module_2 is assembled.



The impact of this is that during debugging with the stepping feature (Chapter 9), the lines of the reassembled modules are listed erroneously. The simple solution to this problem is to execute an IDELETE ALL statement before reassembling more than one module.

Symbolic Operations

You have been introduced, in small doses, to symbols throughout the chapters preceding this one. The idea of symbols in an assembly language is the same as it is in a higher language such as BASIC — to make operations simpler and the code more understandable.

Several symbolic tools are provided for you in this assembly language system. You have already seen one described in detail in this chapter — labels. There are some pre-defined symbols the assembly system provides for certain locations in the machine (mostly registers). There are ways to define your own symbols (and give them a "type"). And, there are ways to access symbols in other modules.

Symbols can be used as operands in machine instructions and in some pseudo-instructions. They can be part of expressions in an operand.

Predefined Symbols

The assembler has predefined a number of symbols and has reserved them as references to special locations in memory. Each of the locations has a special meaning and function. The symbols themselves are "reserved", meaning they cannot be re-defined (by using them as labels on something else). The symbols are —

Symbol	Description
A	Arithmetic accumulator
Ar1	BCD arithmetic accumulators
Ar2	BCD arithmetic accumulators
B	Arithmetic accumulator
Base_page	Global temporary area (9 words)
C	Stack pointer
Cb	Address-extension bit for byte pointer in C
D	Stack pointer
Db	Address-extension bit for byte pointer in D
Dmac	DMA count register
Dmama	DMA memory address register
Dmapa	DMA peripheral address register
End_isr_high	Reserved symbols for interrupt service routines
End_isr_low	Reserved symbols for interrupt service routines
Isr_psw	Reserved symbols for interrupt service routines
Oper_1	Arithmetic utility operand address registers
Oper_2	Arithmetic utility operand address registers
P	Program counter
Pa	Peripheral address register
R	Return stack pointer

Symbol	Description
R4 R5 R6 R7	I/O registers
Result	Arithmetic utility result address register
Se	Shift-extend register
Utlcount Utlend Utltemps	Reserved symbols for writing utilities

The meaning of each of these locations is discussed in other chapters. The absolute locations of the registers can be found in Chapter 2. A description of the function of the accumulators and pointers can be found in Chapter 3 as part of the discussion on machine instructions. A discussion of the I/O registers and symbols can be found in Chapter 7. The arithmetic registers are discussed in Chapter 5.

Using a pre-defined symbol in a machine instruction is the same as using its address. For example —

```
ISOURCE   LDA B
```

means simply that register A will be loaded with the contents of register B. The same effect could have been achieved with —

```
ISOURCE   LDA 1
```

except that the symbolic form makes it more obvious what is intended by the operation. This is true with most symbols.

Defining Your Own

You are defining your own symbol each time you specify a label on an instruction or pseudo-instruction. Normally the "value" of the label is the address associated with the instruction. However, in two cases it is possible to create the label and specify what its value is to be. One case is when the label is on the EQU pseudo-instruction; the other case is when the label is on the SET pseudo-instruction.

The EQU is an assembly-time construct. It exists only at the time of assembly to give you value-assigning capability to symbols. It generates no code itself, and it has no implementation or "location" in the object module.

To define a symbol using an EQU, the form is —

{label}: `EQU` {expression}

the resulting symbol ({label}) has the same "type" as the expression (see "Expressions" later in this chapter) and it has the same value as the result of the expression.

As an example, assembling the statement —

```
ISOURCE Three:     EQU 3
```

means that in all references in the module to the symbol "Three", it is the same as referring to the **value** 3. Thus —

```
LDA Three
```

means load A with the contents of location 3.

A common use for this instruction is to assign a symbol an address which is an offset from another address. For example, if this sequence were in a module —

```
ISOURCE Save_registers:     BSS 40B
ISOURCE Save_b:             EQU Save_registers+1
```

then Save_b would refer to the second word in the BSS area "Save_registers", and it would probably be used to store away the contents of the B register sometime —

```
ISOURCE   STB Save_b
```

and later retrieve the value —

The SET pseudo-instruction defines a symbol in identical fashion to an EQU. Consequently, it has the same general form —

{label}: SET {expression}

The difference between the two is that the SET instruction can have its {label} be a symbol which has been previously defined. The effect in that case is to allow a **redefinition** of the symbol. For example, after assembling the following instructions —

```
ISOURCE Three:     EQU 3
ISOURCE Three:     SET 30B
```

the symbol "Three" has the value 30B.

Literals

Literals are a special means of defining your own symbols without actually having to go to the trouble to do so. The result is a form of symbolic addressing without the symbol! The assembler automatically allocates space at the end of each module for the storage of literal values. This area is called a literal pool.

The form of a literal is —

= {expression} [, {expression} [, ...]]

where {expression} may be any absolute or relocatable expression (see "Expressions" below).

Evaluation of Literals

When a literal is encountered in an operand, three things occur —

1. The literal is converted to its binary value. If there is more than one expression in the literal, then they are all converted.
2. The binary value is stored in a literal pool. If there is more than one expression in the literal, then they are stored contiguously in the order specified.
3. The address where the value is stored is then substituted for the literal in the operand.

If the same literal is used in more than one instruction, only one value is generated in the literal pool. All instructions using this literal refer to the same location.

Literals can be part of expressions as well as having expressions as part of them. Since they ultimately are replaced by an address (pointing to a specific location within a literal pool), their "type" is "relocatable". See the section on "Expressions" later in this chapter.

Basically, a literal means "the address of {expression}". An example should help in the understanding of literals. Suppose that you want to store the value 1 into the A register. There are two ways you could accomplish that purpose. You **could** code —

```
One:  DAT 1
       .
       .
       .
      LDA One
```

or, you could use a literal and code —

```
LDA =1
```

Using the literal method is easier and is more self-documenting. While the literal form strictly says "load A with the contents of the **address of** the constant 1", it can also be read as "load A with the constant 1", and this short-hand version can be an excellent way of self-documenting your programs, not to mention the elimination of a lot of unnecessary symbols.

The value of symbols defined with the EQU pseudo-instruction are referenced using the literal specifier. For example —

```
Select_code: EQU 6
             .
             .
             .
             LDA = Select_code
```

Nesting Literals

Since literals use expressions, and literals may be used in expressions, it is possible to have a literal within a literal (nesting). In fact, it may be done to any depth, though the most useful form of nesting is a single level.

Suppose you want to initialize a variable to the value of pi each time you enter a routine. A nested literal would be a way of accomplishing this in a clean, straight-forward fashion —

```
Pi:  BSS 4
      .
      .
      .
     LDA ==3.14159265349
     LDB =Pi
     XFR 4
```

and the locations starting at "Pi" now contains the full-precision value indicated (which is a fair approximation to pi). This would replace coding which could have looked like this (without using literals) —

```
A_init:  DAT Init
Init:    DAT 3.14159265349
A_pi:    DAT Pi
Pi:      BSS 4
          .
          .
          .
         LDA A_init
         LDB A_Pi
         XFR 4
```

Literals are also used to provide an instruction or a utility (e.g., the XFR instruction and the Print_string utility) with the address of the first word of a string, or full-precision or short-precision number. In these cases the "= =" specifier is used. For example —

```
LDA == 3.14159
```

puts the address of the first word of the short-precision number in the A register for the XFR instruction. Likewise —

```
LDA == 7,"EXAMPLE"
```

puts the address of the first word of the BASIC string "EXAMPLE" in the A register for the Print_string utility. (See Chapter 7, I/O Handling, for an explanation of the Print_string utility).

Nonsensical Uses of Literals

A literal, basically, is an address. Since it can be used in an operand wherever an address may be used, it is possible to use it in instructions where the result is a little nonsensical.

For example, consider the result of doing some of the following —

```
STA =2
JSM ="GARBAGE"
DSZ =- 1
JMP =Neverneverland
SZA =Out_to_lunch
```

Caution dictates that you well consider the appropriateness of the action when using the literal. Literals can be a highly useful tool, but only when properly employed.

Literal Pools

Literals are assemble-time constructs, but they eventually resolve to an actual address in the object code. That address points into the literal "pool".

The literal pool is part of your module where the actual values of literals are stored. There is automatically a literal pool assigned at the end of each module where literals are used. As many literal values as possible are stored there by the assembler. However, in some cases, a literal pool is needed earlier in the program (a need indicated by the assembler with the "LT" assembly-time error). In that case a pool should be created using the LIT pseudo-instruction. This instruction has the form —

`LIT` {size}

where {size} is the number of words to be set aside (it may be a positive numeric expression). The instruction acts very much like a BSS. And, like a BSS, it should be placed at a location in your code where it is not likely to be inadvertently executed.

Most modules do not need assignment of an extra literal pool. However, one is needed where there is a literal used beyond 512 words from the first available space in the literal pool at the end of the module. To alleviate the problem, a literal pool must be created with the LIT statement within 512 words of the instruction.

A common cause of this kind of problem is a large BSS assignment between the instruction and the end of the module. Sometimes moving the BSS to some other location is a solution to the problem.

Expressions

Literals, some pseudo-instructions (particularly EQU), and a number of machine instructions, all permit "expressions" to be used as an operand. These expressions take one of two forms — "absolute" or "relocatable". The type of an expression depends upon the type of the individual **elements** in it.

An element is of the type "absolute" if it is any of the following —

- A decimal integer (like 0, 1, 2, 1 024).
- An octal integer (like 10B, 40B, 100000B).
- A string (enclosed by quote marks) (like "ERROR")
- An ASCII character, preceded by an apostrophe (like 'A).
- A label associated with an EQU or SET pseudo-instruction whose expression is also evaluative as type absolute (like EQU 40B).

An element is of the type "relocatable" if it is any of the following —

- A label not associated with an EQU or SET pseudo-instruction (i.e., it is an "address").
- A literal (like =0).
- An asterisk, symbolizing "current address".
- A label associated with an EQU or SET pseudo-instruction whose expression is also evaluative as type relocatable (like EQU *).

An expression is a list of elements each pair of which is separated by one of the following operators —

+ − / *

meaning addition, subtraction, division, and multiplication, respectively, as in BASIC.

The result of an expression is either absolute or relocatable depending upon the following rules:

An absolute expression is any expression which contains —

- Only absolute elements.
- An even number of relocatable elements, paired in sequence and by sign (i.e., for each relocatable element there is another relocatable element adjacent to it, of opposite sign). These pairs may be in combination with absolute elements.

A relocatable expression is any expression which contains —

- An odd number of relocatable elements, paired in sequence and by sign, except the last, which must be positive.
- An odd number of relocatable elements, as above, in combination with any number of absolute elements.

Any combination of absolute or relocatable elements which does not result in either an absolute or relocatable value, by the rules above, results in an error.

These rules and the rules for using * and / can be summarized as —

The expression is —	**The type is —**	**Example**
absolute ± absolute	absolute	`1000B + 10`
absolute + relocatable	relocatable	`1 + Temp`
relocatable ± absolute	relocatable	`Temp - 1`
relocatable − relocatable	absolute	`Temp 1 - (Temp - 1)`
relocatable + relocatable	error	`Temp + Temp 1`
absolute − relocatable	error	`1000B - Temp`
absolute * absolute	absolute	`100 * 3`
absolute/absolute	absolute	`100/3`
absolute * relocatable	error	`Temp * 3`
relocatable * absolute	error	`3 * Temp`
absolute/relocatable	error	`Temp/3`
relocatable/absolute	error	`3/Temp`

Unlike BASIC, there is no precedence among the operators. All are of equal precedence. Where precedence is desired, parentheses must be used. So where BASIC requires —

```
2*16+3*8
```

to result in 56, the same expression in the assembly language results in 280 (assembly language operators are evaluated from left to right). However, 56 would be the result if it were expressed as —

```
(2*16)+(3*8)
```

An expression may be of any length and contain as many operators and parentheses as desired, as long as the result can be evaluated and the parentheses are properly paired. All operators are evaluated from left to right. Multiplication and division can only be used with elements that are of type absolute.

Both operands are considered to be unsigned integers for assembler division (/). Overflows in all assembler arithmetic operations are ignored.

External Symbols and Elements

There is an additional relocatable element, called "external". It behaves in almost all respects as does any other relocatable element, except that only one external item may appear in an expression. Also, the expressions containing —

relocatable − relocatable

are not allowed when one of the relocatable elements is external. Externals are defined as symbols appearing in an EXT pseudo-instruction —

EXT {symbol} [, {symbol} [, ...]]

These are entry points in another module or utility. "Entry points" are merely symbols in a module which are listed in an ENT pseudo-instruction in that module —

ENT {symbol} [, {symbol} [, ...]]

If one module contains —

```
ENT Stage_left
```

then that symbol would be available to another module which contains —

```
EXT Stage_left
```

The EXT instruction should appear before any other instruction using the symbols which are listed in that EXT instruction. At execution time for a module with an EXT instruction, all of the symbols listed in it must be either a utility name or be contained in an ENT or SUB (described in Chapter 6) of another module. It is not necessary that the module be in source form; it may already be an object module assembled from a source module which contained the symbol as an ENT or SUB.

NOTE

When ICALLing an assembly routine, satisfaction of the external symbols specified by an EXT pseudo-instruction is checked only for the first module after the ICALL. The external symbols of modules entered after the first module are not checked. Undesirable results can be obtained if externally referenced modules cannot be found. Be sure that all interrelated modules reside in the ICOM region before an ICALL is executed.

Other Absolute Elements

There are additional **absolute** elements which may be used in expressions. These are "machine addresses", short-precision numbers, and full-precision numbers.

A machine address is one of the following —

- An assembler pre-defined symbol.
- A symbol associated with an EQU or SET pseudo-instruction whose expression is evaluated as a machine address (i.e., it contains a pre-defined symbol or another EQU-associated symbol whose expression contains a pre-defined symbol).

For the most part, machine addresses can be used just like absolutes. However, they remain defined from assembly to assembly. By defining a machine address in one module (with an EQU or SET), it then becomes available to you with the same value in other modules which you assemble.

For example, if you were to assemble a module containing —

```
ISOURCE R100:  EQU A+100
```

then R100 is a machine address following the above rules, just as if the assembler had predefined it. If you don't do any SCRATCH or GET statements in the meantime, then the next assembly you do would also have this symbol available without ever having to define it.

When full-precision numbers (like − 2.5, 3E3, 3.141592) and short-precision numbers (like 1.S, − 2.5S, 3.14159S, 3.E3S) are used in expressions, they become the **entire** expression. This is because these numbers are only intended as simple data-generating devices in literals and in DAT pseudo-instructions. Explicitly, the rules for using full- and short-precision numbers are —

- They may only appear alone in an expression, i.e., they may not be in combination with other elements.
- They may only appear in literals and in DAT pseudo-instructions.

Utilities

A number of utilities have been provided to help make your programming tasks easier and to give you direct access to some of the operating system's capabilities and routines.

Descriptions of the utilities are made in conjunction with those topics where the utilities play a part. The form of the description of a utility is somewhat standardized. Each description will tell you —

- The name of the utility.
- The general procedure for using the utility.
- Any special requirements which must be satisfied for the utility to work properly.
- A step-by-step calling procedure for the utility.
- The exit conditions.

Utilities are a form of subroutine, so to execute them it is necessary to execute a jump-to-subroutine instruction (JSM) if you want the utility to return to the routine which calls it. Most utilities execute a RET 1 instruction to return, so in some cases where you follow a utility call with a RET 1 of your own, you can save the RET instruction by using the JMP (unconditional branch) instruction instead. For example, a typical utility call looks like —

```
 .
 .
 .
LDA =Temp
LDB =Pointer
JSM Get_element
 .
 .
 .
```

but if it happened to be followed by a RET 1 —

```
 .
 .
 .
LDA =Temp
LDB =Pointer
JSM Get_element
RET 1
 .
 .
 .
 .
```

the calling procedure could be changed to —

```
  .
  .
  .
LDA =Temp
LDB =Pointer
JMP Get_element
  .
  .
  .
```

and you save a word of code: the effect is otherwise the same. Check the exit conditions for a utility before using this approach.

Utilities which you use in a module must have their names in an EXT pseudo-instruction for that module. Otherwise, the assembler is unable to tell that you meant a utility and not one of your own labels, causing an "undefined reference" assembly error.

The contents of any or all of the processor registers may be altered after a return from a utility. Be sure to save the contents of registers that you are using before you call a utility.

If you are using interrupts, the interrupt system may or may not be enabled upon return from a utility. Use the EIR and DIR instructions to ensure the proper state of the interrupt system upon return from a utility. A system utility cannot be called from an interrupt service routine (ISR).

Appendix F contains a short description of the utilities.

The utilities currently available are —

Utility	Description
Busy	Tests the busy bits of a BASIC variable
Error_exit	Aborts an ICALL statement with a particular error number
Get_bytes	Accesses substrings (or parts of parameters)
Get_elem_bytes	Same as "Get_bytes", but used for array elements
Get_element	Same as "Get_value", but used for array elements
Get_file_info	Accesses the file-pointer of an assigned file
Get_info	Returns the characteristics of a variable passed as a parameter or existing in common
Get_value	Returns the value of a BASIC variable
Int_to_rel	Data type conversion from integer to full-precision
Isr_access	Establishes hardware linkages for interrupts
Mm_read_start	Prepares to read a physical record from mass storage
Mm_read_xfer	Reads a physical record from mass storage
Mm_write_start	Writes a physical record to mass storage
Mm_write_test	Verifies a physical record was written to mass storage
Printer_select	Changes or interrogates select-code for standard printer
Print_no_lf	Outputs a string with no CR-LF sequence
Print_string	Outputs a string to the standard printer
Put_bytes	Replaces substrings (or parts of parameters)
Put_elem_bytes	Same as "Put_bytes", used for elements in an array
Put_element	Same as "Put_value", used for elements in an array
Put_file_info	Manipulates the file-pointer of a file
Put_value	Changes the value of a BASIC variable
Rel_math	Provides access to all the arithmetic routines
Rel_to_int	Data type conversion from full-precision to integer
Rel_to_sho	Data type conversion from full-precision to short
Sho_to_rel	Data type conversion from short-precision to full
To_system	Allows immediate printing with printing utilities

Chapter 5
Arithmetic

Summary: Arithmetic operations are reviewed and the arithmetic utilities are discussed. Floating point and BCD arithmetic are explained, as well as integer arithmetic.

Numerical calculations are a large part of any computer's operations. Implemented within the System 45's processor are both integer and primitive Binary Coded Decimal (BCD) floating-point arithmetic operations. These operations are needed because three of the four BASIC variable data types (explained in Chapter 3) are represented either as BCD floating point numbers or as integer (binary) values. To be specific, full-precision numbers are presented as 12-digit, BCD floating point numbers, short-precision numbers are represented as 6-digit, floating point numbers, and integers are represented as binary numbers. This chapter deals with integer and floating point operations and is intended for those readers who may have no acquaintance with this topic, or perhaps only a passing one. The particular machine instructions involved with such arithmetic are reviewed.

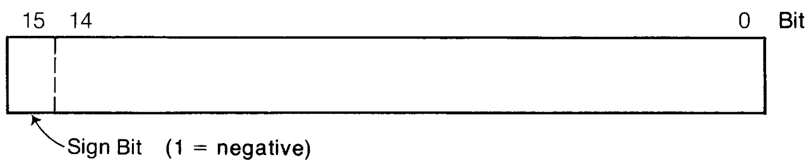
Because the processor provides only rudimentary floating-point operations and because complete floating-point operations (e.g., subtract, divide) are not easy to write, BCD arithmetic utilities have been provided to perform these calculations and are discussed later in this chapter. Integer arithmetic operations are less complex; thus utilities can be written by you, as described in the following section. If you are not interested in doing your own BCD or integer arithmetic, it is recommended that you skip immediately to ''Arithmetic Utilities''.

Due to its speed increases over BCD floating point arithmetic, integer arithmetic is recommended when you are performing the addition, subtraction, or multiplication of integers.

Integer Arithmetic

Representation of Integers

Recall from Chapter 3 that integers are represented as –



The range of integers represented by 16 bits in the 9845 is –

–32 767 to +32 767

This is further illustrated in the following table –

Decimal	Bit 15	Binary Integer Representation				Bit 0
−32 768	1	000	000	000	000	000
−32 767	1	000	000	000	000	001
•						
•						
•						
−1	1	111	111	111	111	111
0	0	000	000	000	000	000
1	0	000	000	000	000	001
•						
•						
•						
32 767	0	111	111	111	111	111

Notice that negative integers have their sign bit (bit 15) equal to one. There is another important fact concerning negative numbers – they are represented in two's complement form. This is done so that subtraction can be implemented by complementing and adding. There are two instructions (TCA and TCB) which enable you to form the two's complement of an integer. An example of the use of two's complement is shown –

```
10    ISOURCE Absolute:    LDA Integer    ! Find the absolute value
20    ISOURCE              SAP *+2        ! of an integer.
30    ISOURCE              TCA            ! If not positive, two's
40                                        ! complement it.
50    ISOURCE              STA Abs_value  ! Save absolute value.
```

Integer Arithmetic[1]

The addition of integers is accomplished very easily. Two instructions (ADA and ADB) are provided to do integer addition. A special situation to be aware of is the overflow condition. It is possible to add two valid 16-bit integers and produce an answer which cannot be represented in 16 bits.

1 For the purposes of this manual, the terms binary arithmetic and integer arithmetic are synonomous.

For example, 15 000 + 25 000 = 40 000, and 40 000 is greater than 32 767 (the upper limit).

The following example illustrates how to detect this condition –

```
ISOURCE                   SOC *+1,C        ! Clear overflow indicator.
ISOURCE                   LDA Number_1
ISOURCE                   ADA Number_2
ISOURCE                   SOC Ok           ! Test for overflow.
ISOURCE Overflow:         LDA =20
ISOURCE                   JMP Error_exit   ! Give integer precison
                                           !  overflow error.
ISOURCE Ok:               STA Answer       ! Continue processing.
```

Of course, if you know that the result will be in the range −32 768 to +32 767, there is no need to check for the overflow condition.

The subtraction of integers is handled almost exactly like addition. The following example computes (X-Y) –

```
ISOURCE                   SOC *+1,C        ! Clear overflow indicator.
ISOURCE                   LDA Y
ISOURCE                   TCA              ! Form -Y.
ISOURCE                   ADA X            ! Compute X+(-Y)=X-Y.
ISOURCE                   SOC Ok           ! Test for overflow.
ISOURCE Overflow:         LDA =20
ISOURCE                   JMP Error_exit   ! Give integer precison
                                           !  overflow error.
ISOURCE Ok:               STA Answer       ! Continue processing.
```

The processor contains an integer multiply instruction. There are two special considerations concerning integer multiplication –

- When you multiply two 16-bit integers, the resulting product can always be represented as a 32-bit integer. Hence, the processor's MPY instruction produces a 32-bit answer, and no overflow condition is possible. However, if you would like to restrict products to a valid 16-bit integer, you must provide your own 16-bit integer overflow check.
- An anomaly exists in the MPY instruction. If the B register contains −32 768, the MPY instruction yields the wrong answer.

The following example multiplies two 16-bit integers (X and Y) and tests the result to see if it is a valid 16-bit integer –

```
ISOURCE Test:       LDB X          ! Multiplication routine.
ISOURCE             CPB =-32768
ISOURCE             JMP Anomaly
ISOURCE             LDA Y
ISOURCE Mpy_1:      MPY
ISOURCE             SAP *+2        ! Detect overflow when all
ISOURCE             CMB            !  bits of B differ from upper
                                   !  bit of A.
ISOURCE             RZB Overflow
ISOURCE             STA Answer     ! Save result.
ISOURCE             RET 1          ! Exit.
ISOURCE Overflow:   LDA =20        ! Give integer overflow error.
ISOURCE             JSM Error_exit
ISOURCE Anomaly:    LDA B
ISOURCE             LDB Y          ! Exchange A and B.
ISOURCE             CPB =-32768    ! Is B now -32768?
ISOURCE             JMP Overflow   ! If yes, give overflow message.
ISOURCE             JMP Mpy_1      ! If not, multiply.
```

The processor does not contain an integer divide instruction. However, integer division can be implemented quite easily. The following program implements integer division (X/Y) analogous to the BASIC DIV operator with integer operands –

```
ISOURCE             LDA X
ISOURCE             STA Dividend
ISOURCE             LDB Y
ISOURCE             STB Divisor
ISOURCE             RZB Not_zero        ! Skip if not dividing by 0.
ISOURCE             LDA =31
ISOURCE             JMP Error_exit      ! Give division by 0 error.
ISOURCE Not_zero:   LDA =0
ISOURCE             STA Quotient        ! Initialize quotient.
ISOURCE             SOC *+1,C           ! Necessary for -32768.
ISOURCE             SBP Pos_divisor
ISOURCE             CMA                 ! Toggle sign saver.
ISOURCE             TCB                 ! Force positive divisor.
ISOURCE             STB Divisor
ISOURCE Pos_divisor:LDB Dividend
ISOURCE             SBP Pos_dividend
ISOURCE             CMA                 ! Toggle sign saver.
ISOURCE             TCB                 ! Force positive dividend.
ISOURCE             STB Dividend
ISOURCE Pos_dividend:STA Sign           ! Save sign of quotient.
ISOURCE             SOC *+3
ISOURCE             LDA =19             ! Give improper value error-
ISOURCE             JMP Error_exit      !  one operand was -32768.
ISOURCE             LDA =1              ! Initialize quotient update.
ISOURCE !  FIND MOST SIGNFICANT DIGIT IN QUOTIENT.
```

```
ISOURCE Div1:       LDB Divisor
ISOURCE             TCB
ISOURCE             ADB Dividend
ISOURCE             SBM Div2          ! Skip if divisor>dividend.
ISOURCE             SAL 1             ! Increase what to add
ISOURCE             LDB Divisor       !  to quotient; the divisor
ISOURCE             SBL 1             !  must be updated in the
ISOURCE             STB Divisor       !  same manner.
ISOURCE             JMP Div1
ISOURCE !  EXAMINE ALL REMAINING LEAST SIGNIFICANT BIT POSITIONS TO
ISOURCE !   DETERMINE IF THEY ARE PART OF THE QUOTIENT.
ISOURCE Div2:       SAR 1             ! See if next bit is to be
ISOURCE             SZA Do_sign       !  included.
ISOURCE             LDB Divisor       ! Maybe; first, it is
ISOURCE             SBR 1             !  necessary to adjust the
ISOURCE             STB Divisor       !  divisor for bit position.
ISOURCE             TCB
ISOURCE             ADB Dividend
ISOURCE             SBM Div2          ! Skip if bit should be off.
ISOURCE             STB Dividend      ! Bit should be on; adjust
ISOURCE             LDB Quotient      !  dividend and quotient
ISOURCE             ADB A             !  to account for it.
ISOURCE             STB Quotient
ISOURCE             JMP Div2          ! Check all bit positions.
ISOURCE Do_sign:    LDA Quotient
ISOURCE             LDB Sign
ISOURCE             SBP *+2
ISOURCE             TCA               ! Complement sign of quotient.
ISOURCE             STA Quotient
ISOURCE             LDA =Quotient
ISOURCE             LDB =Quotient
ISOURCE             JMP Put_value     ! Save results, return to BASIC.
ISOURCE             END Division
```

Multi-Word Integer Arithmetic

The processor does not directly support multi-word arithmetic. However, it does provide a register (the E register) which facilitates multi-word addition. The E register indicates whether there is a "carry" from bit 15 when an add instruction (ADA or ADB) is executed.

The following program segment illustrates how 2-word integers can be added –

```
ISOURCE            SEC *+1,C      ! Clear E register.
ISOURCE            LDA X_right
ISOURCE            ADA Y_right    ! Form least significant
                                  !  word of answer; set E
                                  !  register if carry out
                                  !  of bit 15.
ISOURCE            SOC *+1,C      ! Clear overflow register.
ISOURCE            LDB X_left
ISOURCE            SEC Normal
ISOURCE            ADB =1         ! Add in carry from least
                                  !  significant part.
ISOURCE            SOC Normal,C   ! Skip if no overflow.
ISOURCE            ADB Y_left
ISOURCE            SOS Ok         ! If there was another overflow,
                                  !  the answer is correct.
ISOURCE Err20:     LDA =20        ! Give integer overflow error.
ISOURCE            JSM Error_exit
ISOURCE Normal:    ADB Y_left
ISOURCE            SOS Err20
```

Subtraction can also be handled, by forming the two's complement. The general algorithm is –

1. Form the two's complement of the least significant, non-zero word.
2. Form the one's complement (using CMA or CMB) of all more significant words.

The following program segment illustrates how to compute the two's complement of a two word integer –

```
ISOURCE            LDA Right_word
ISOURCE            SZA Next_word
ISOURCE            TCA
ISOURCE            LDB Left_word
ISOURCE            CMB
ISOURCE            RET 1
ISOURCE Next_word: LDB Left_word
ISOURCE            TCB
ISOURCE            STA Answer_right
ISOURCE            STB Answer_left
ISOURCE            RET 1
```

Binary Coded Decimal

Binary Coded Decimal (BCD) uses four-bit binary codes to represent decimal digits. Thus, the 12-digit mantissa of a full-precision number is represented by 48 bits. The BCD digits are as follows —

DECIMAL	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

A BCD number within this manual has its digits represented as D_1, D_2, D_3, etc., with each digit corresponding to some BCD digit. D_1 is the most significant digit in a number. Since full-precison numbers within the 9845 contain 12-digit BCD mantissas, 12-digit BCD numbers are used as the most frequent examples in this discussion. In that case, D_{12} is the least significant digit in a number.

Arithmetic Machine Instructions

There are some machine instructions which specifically operate upon the BCD registers. The discussions in this chapter will make use of the capabilities of these instructions to develop the techniques to write BCD arithmetic routines. If you have not done so already, you should familiarize yourself with the instructions before moving on in this chapter. A description of the instructions can be found in ''Arithmetic Group'' in Chapter 3.

BCD Registers

There are two registers in the machine used for BCD arithmetic — Ar1 and Ar2. These symbols are pre-defined by the assembly language to the registers' locations in memory (see Chapter 3). The mnemonics for some instructions occasionally refer to these registers as X and Y respectively (see Chapter 3).

BCD Arithmetic

To understand BCD arithmetic in the context of the 9845, recall from Chapter 3 that a full-precision value is represented in four words which contain its information as follows —

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Bit

Exp Sign	Exponent	0 0 0 0 0	Man Sign
D_1 (most significant digit)	D_2	D_3	D_4
D_5	D_6	D_7	D_8
D_9	D_{10}	D_{11}	D_{12} (least significant)

The exponent is stored in two's complement form. The exponent and the mantissa are always adjusted by arithmetic routines so that there is always an implied decimal point following D_1. Thus, the mantissa of every value stored looks like —

$$D_1 . D_2\ D_3\ D_4\ D_5\ D_6\ D_7\ D_8\ D_9\ D_{10}\ D_{11}\ D_{12}$$

Except possibly for intermediate results within the individual arithmetic algorithms, the most significant digit of a full-precision value (D_1) will never be 0 unless the entire number is 0. Sometimes, after an individual arithmetic operation, the answer needs to be **normalized**, that is, the digits of the answer shifted to the left until D_1 is no longer 0. The exponent then needs to be adjusted to reflect the change.

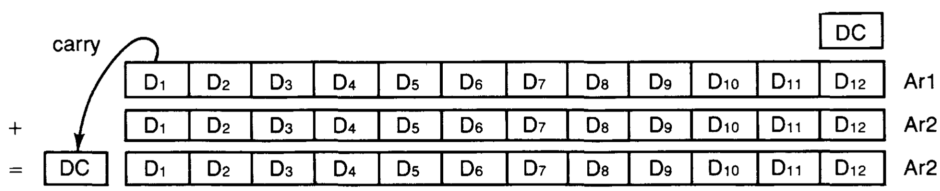
An important thing to keep in mind when examining BCD arithmetic, as implemented by the processor, is that mantissas are represented in a "sign-magnitude" format. This means that the absolute value is stored as the actual mantissa, and the sign of the mantissa is maintained separately.

Addition

There is a one-bit Decimal Carry (DC) flag within the processor which serves a BCD function similar to the Extend flag for binary addition.

DC is set to a one or zero, depending upon the occurrence or absence of a carry from the addition of the two D_1's of the two BCD numbers being added. Since mantissas are represented in a sign-magnitude form (with the sign in the exponent word rather than part of what gets added), DC represents an overflow for 12-digit mantissa additions.

DC itself is part of the addition in the D_{12} position. This gives it potential use with multiple-precision floating point arithmetic. The addition process looks like this —



There are three instructions which concern themselves exclusively with DC. They are — SDS (Skip if DC set), SDC (Skip if DC clear), and CDC (Clear DC).

Ten's Complement for BCD

The addition of the ten's complement of a number is used in lieu of a subtraction mechanism. If the signs of the two numbers to be summed are different, one of the numbers is complemented (it doesn't really matter which one), before the addition.

The ten's complement of a number with n digits to the left of the decimal point is —

$$X = 10^n - X$$

The ten's complement of a floating-point number has the same exponent as the original number. Since the mantissa (M) of a full-precision number can be assumed to have the decimal point implied after D_1, then the number must be less than 10 (but greater than 0) and the ten's complement of a mantissa becomes —

$$M = 10 - M$$

Accordingly, all that is necessary to complement a floating-point number is to complement the mantissa. It is immaterial whether the mantissa is treated as a 12-digit integer or as a number between 0 and 10; the same sequence of digits results.

There are two instructions for doing ten's complements — CMX and CMY. The only difference between them is that CMX operates on the Ar1 register and CMY operates on the Ar2.

CMX and CMY leave the exponent word of a full-precision number completely alone. This means that the sign of the mantissa and the entire exponent are left unchanged in a ten's complement by CMX and CMY.

Ten's complement helps to accomplish addition, too. Rather than go into all of the nuances and subtleties of the arithmetic process, there is a simple rule for accomplishing decimal summations using ten's complements. Assuming the exponents are the same for the numbers to be added —

- If the signs of the numbers are the same, simply add them and leave the signs alone. If DC occurs, the result (Ar2) must be shifted to the right one place, and the exponent adjusted.
- If the signs of the numbers are different, complement, then add. A further complementing action may be necessary: if DC occurs, then the result necessarily has the same sign as the number which was not complemeted; if DC does not occur, then the result must be complemented and then given the sign of the number which was complemented.

The FXA instruction is used to add mantissas. Here is a routine to implement the rule —

```
ISOURCE   LDA Ar1          ! Check the sign
ISOURCE   ADA Ar2
ISOURCE   SLA Just_add     ! Skip if they are the same
ISOURCE   CMX              ! Complement Ar1
ISOURCE   FXA              ! Add the mantissas
ISOURCE   LDB Ar2
ISOURCE   SDS *+3          ! Was there an overflow?
ISOURCE   CMY              !   No, so complement result
ISOURCE   LDB Ar1          !   and switch exponents and signs
ISOURCE   STB Ar2          ! Store the larger sign
ISOURCE   JMP Done
ISOURCE Just_add:          ! Do the addition
ISOURCE   FXA
ISOURCE   SDC Done         ! Was there an overflow?
ISOURCE   LDA =1           !   Yes, so shift in a 1
ISOURCE   LDB =1           !   into the most
ISOURCE   MRY              !   significant digit
ISOURCE   LDA Ar2          ! Adjust exponent
ISOURCE   ADA =100B
ISOURCE   STA Ar2
ISOURCE Done:   ! CONTINUE ON
```

Floating Point Summations

In the example just completed, you may have noted that to copy the sign the entire exponent word was copied. What if the exponents were different? The answer is — the exponents must have been the same. In fact, the only reason the example worked at all was that the exponents were the same.

If exponents are different, addition of mantissas cannot proceed properly. To add the numbers it is necessary to make the exponents the same by shifting one of the mantissas an amount equal to the exponent difference.

This difference is easily found by subtracting the smaller exponent from the larger. If the difference is eleven or less (the precision of the 12-digit mantissa), it is possible to offset the mantissa of the number with the **smaller** exponent.

For example suppose there are two numbers to be added —

```
X.XXXXXXXXXXX E6
Y.YYYYYYYYYYY E4
```

By shifting the smaller one to the right by 2 digits (the difference between 6 and 4), it is possible to align the exponents —

```
X.XXXXXXXXXXX   E6
0.0YYYYYYYYYYYY E6
-----------------------
Z.ZZZZZZZZZZZ   E6
```

As can be readily seen from the example, a shift of more than 11 digits would cause the smaller value to be all zeroes in the significant 12 digits.

The digits to the right of the 12 most significant digits are lost in the action of shifting. That is, all except the left-most one. When using the MRX or MRY instructions, this digit is retained in the A register (bits 0-3) so that it can be used later for rounding purposes.

To use the MRX or MRY instructions, the number of digits to be shifted must be present in the B register.

The process for this "justification" of exponents can be summed up as follows:

- Subtract one exponent from the other storing the absolute value of the difference in the B register.
- Execute the MRX shift if the Ar1 register is smaller; execute the MRY shift if the Ar2 register is smaller.

Normalization

The raw result of an arithmetic operation (such as FXA) might not be a floating-point number that fits the standard form. It might have a leading DC needing to be incorporated into the number, as was seen in the "Addition" section earlier. Another possible deviation is a resulting D_1 of zero and no overflow. There could also be several zero-valued digits as left-most digits of the mantissa.

Such situations call for "normalization". One type of normalization is accomplished with the NRM instruction. This instruction shifts register Ar2 left, leaving the number of shifts required in the B register as a binary number. The maximum number of shifts NRM performs is 12. If NRM must do all twelve shifts, Ar2 must have been 0. This is indicated by a value of 12 left in B and DC being set. For any other shift-count, NRM will leave DC at 0.

The rules for the normalization process are —

- Execute the NRM instruction.
- Follow this instruction by adding the complement of the contents of B (shifted left 6 bits) to the Ar2 exponent unless DC is set. If DC is set, store 0 into Ar2.
- Test the exponent result for an underflow.

Rounding

The addition operation (FXA) does not automatically round a result, and there is no instruction which does rounding in one step. Instead, it is necessary that a series of instructions be established to accomplish the result.

Recalling from "Floating Point Summations" (above) that the rightmost digit for rounding purposes (if any) is typically deposited in the A register by an MRX or MRY instruction, this digit can be checked to determine if rounding is required.

The process of rounding, then, would have the following steps —

- Determine from register A if rounding is required (i.e., if it's greater than or equal to 5).
- If rounding is not required, take no further action. If rounding is required, then load register B with 1 and execute an MWA instruction. This has the effect of incrementing the mantissa in Ar2 by 1. This action is an easier method than setting Ar1 to 1 and executing an FXA and it's faster, too. Don't forget to check DC for an overflow.
- One way the sequence of rounding could appear is —

```
10   ISOURCE   ADA =-5   ! Scale A down
20   ISOURCE   SAM *+3   ! If less than 5, no rounding
30   ISOURCE   LDB =1    ! Get ready to add 1 to Ar2
40   ISOURCE   MWA       ! Add 1 to least significant digit of Ar2
```

Floating Point Multiplication

Twelve-digit BCD floating-point multiplication is partially accomplished using the FMP instruction. This instruction effectively multiplies the value in the Ar1 register by a digit contained in B and adds the result to a partial product in Ar2.

Since, in the full multiplication process, exponents are merely added together, that part of the process is trivial. The ultimate sign of the product is also a trivial matter, determined by inspection of the signs of the original operands. Then the only matter of difficulty in the process is the actual multiplication of the mantissas. By way of explanation, assume that there are two mantissas to be multiplied —

multiplicand = A B C D
multiplier = W X Y Z

Just four digits are used to reduce the amount of symbolism required of the example. The same procedures and conclusions are applicable to a full twelve BCD digits.

One symbolic way to indicate how this multiplication is done is —

$$
\begin{array}{cccccccccl}
 & & & & A & B & C & D & & \\
 & & & \text{x} & W & X & Y & Z & & \\
\cline{4-8}
 & & & & 0 & 0 & 0 & 0 & = & \text{partial product 0} \\
 & & & Z_{ov} & Z_1 & Z_2 & Z_3 & Z_4 & = & Z\ (ABCD) \times 10^{0} \\
\cline{4-8}
 & & & P_4 & P_5 & P_6 & P_7 & P_8 & = & \text{partial product 1} \\
 & & Y_{ov} & Y_1 & Y_2 & Y_3 & Y_4 & 0 & = & Y\ (ABCD) \times 10^{1} \\
\cline{3-8}
 & & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & = & \text{partial product 2} \\
 & X_{ov} & X_1 & X_2 & X_3 & X_4 & 0 & 0 & = & X\ (ABCD) \times 10^{2} \\
\cline{2-8}
 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & = & \text{partial product 3} \\
W_{ov} & W_1 & W_2 & W_3 & W_4 & 0 & 0 & 0 & = & W\ (ABCD) \times 10^{3} \\
\cline{1-8}
P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & = & \text{partial product 4 (result)}
\end{array}
$$

Notice that at each stage the multiple of ABCD, such as X(ABCD), must be multiplied by an increasing power of ten in order that the digits of the multiple line up appropriately with the digits of the last partial product. An equivalent procedure is to have the partial product shifted right one digit at each stage.

Now, consider for a moment what is necessary within the assembly language to generate partial product 1 = 0 + Z (ABCD). Ar2 must be cleared and Ar1 is loaded with ABCD. Z is stored into B in bits 0 to 3. Then the FMP instruction is executed. Ar1 is added to Ar2 Z times, producing Z (ABCD) in Ar2. The overflow digit, Z_{ov}, ends up in the A register (bits 0 to 3). The overflow digit could be any value from 0 to 9 (each add could cause a carry, and there can be up to nine additions).

To create the next partial product, a mantissa right-shift on Ar2 must occur. Notice that mantissa right-shifting instructions (MRX and MRY) also shift bits 0 to 3 of the A register into D_1. Thus, the right-shifting of the partial product (which must occur to prepare Ar2 for the next partial product) also automatically takes care of retaining the overflow digit.

Next, ABCD is added to Z_{ov} Z_1 Z_2 Z_3 a total of Y times (again by use of the FMP instruction). Partial product 2 is created. The process is repeated for the X and W digits, producing the result in Ar2.

After the final partial product has been calculated by the final execution of the FMP instruction, it is possible that a non-zero digit may be present in bits 0-3 of the A register. Such a digit is necessarily the most significant digit of the final product. In this case, another MRY execution is required. Further, the exponent of the product (which was initially estimated as the sum of the operand's exponents) must be incremented by one to reflect this power-of-ten shift.

Upon each step of partial product summation, a significant digit is lost due to the shift. This can't be helped. In general, the product of two 12-digit numbers has 24 digits of precision, but the bottom 12 digits must be discarded since only 12 BCD digits are stored in a mantissa. An error analysis of the algorithm discloses that dropping these digits causes the answer, on average, to be slightly smaller than it should be. However, rounding introduces a similar error, but in the other direction. Note that the process did not round each partial product.

The discarded digits can be inspected before they are permanently lost. The MRY instruction causes the digit to be placed in the A register (in bits 0 to 3). This provides an easy way for a rounding mechanism to check on those digits as they are discarded. The rounding routine needs to save the last digit discarded for use in rounding in the event the last use of FMP produces no overflow digit.

Finally, it should be noted that you can put WXYZ into B at the very start of the process and simply shift B right 4 bits (with an SBR 4 instruction) between each execution of FMP. After all, FMP uses only bits 0 to 3 of the register as the number of times to add Ar1 and Ar2.

Floating Point Division

There are many possible algorithms to accomplish floating-point division. The one presented here was chosen because of its effective use of the machine instructions and data structures employed by the processor and operating system.

Remembering that full-precision numbers consist of both a signed mantissa and a signed exponent, use can be made of the mathematical properties of both to reduce the division problem to manageable proportions. Suppose that you have two full-precision values to divide —

$$-4.8E3 \div 1.5E-2$$

The mathematical properties of exponents can be utilized and the second exponent can be subtracted from the first giving the exponent of the answer (subject to possible later adjustment). This is the first (and easiest) step in the division algorithm.

Secondly, the mathematical properties of signs within a division process can be used to determine the sign of the quotient from the signs of the divisor and dividend (negative quotient if the signs are different, positive quotient otherwise).

Thus, the problem can be reduced to the division of the mantissas —

$(-4.8 \div 1.5)$ E5

As long as the full-precision numbers have been normalized, this adjustment of the exponents works for any pair of exponents. The normalization of the numbers also assures that the division of the mantissas under the following algorithm is sufficient to produce the mantissa of the result.

Since the decimal point of each mantissa is in the same place, they can be dropped altogether. For example —

$-4.8 \div 1.5 = -48 \div 15$

The algorithm can then consider both the divisor and the dividend as 12-digit integers.

The algorithm begins by placing the normalized values into the BCD arithmetic registers. The divisor (1.5E−2 in the example) is transferred to register Ar1. The dividend (- 4.8E3 in the example) is transferred to register Ar2. Basically, the algorithm subtracts the absolute value of the mantissa of Ar1 from the absolute value of the mantissa of Ar2 until Ar2 is smaller than Ar1. The number of subtractions required for that to occur becomes the first digit in the quotient (it'll be some value between 0 and 9 because the mantissas are normalized). If there is a (non-zero) remainder, then it is shifted left (multiplied by 10) and the subtraction process is repeated to calculate another digit in the quotient. The process is repeated until either a zero remainder occurs , or sufficient digits have been calculated, whichever occurs first. The resulting digits are merged, in order, to form the complete mantissa of the quotient.

There are some points to keep in mind in following the algorithm —

- Suppose you have a divisor whose normalized mantissa is larger than the normalized mantissa of the dividend, for example —

  $15 \div 48$

  then the first digit of the quotient's mantissa could easily be zero. If calculation of only twelve digits were made, the first digit being zero would mean a loss of a significant digit. To guarantee that there are always at least 12 significant digits calculated for the quotient, it is necessary (and sufficient) to calculate 13 digits. The 13th digit can always be thrown away, or used for rounding, if the first digit is not zero. Thirteen digits are always sufficient because you can never have a quotient with **two** leading zeroes, if the divisor and the dividend are both normalized.

- The number of subtractions during the calculation of any digit in the quotient is always nine or less. Again, this is true because the divisor is normalized and its first digit is always non-zero.
- At times during the algorithm, it is necessary to left-shift the mantissa of Ar2 (the mantissa at this point is the remainder). When shifting the remainder to the left (multiplying it by 10), you are shifting the first digit out of Ar2. If this digit is zero, this is not a problem. But, if the digit is non-zero, you can't ignore it during subtractions of the divisor. This in effect means that you are dealing with a 13-digit dividend! Since the machine instructions deal in 12-digit arithmetic, it is necessary that the algorithm handle the thirteenth.

The FDV Instruction

The FDV instruction provided by the processor is the primary tool used to implement the algorithm in assembly language. The instruction works by accomplishing the equivalent of automatically repeated subtractions of Ar1 (the divisor) from Ar2 (the dividend) until Ar2 is smaller than Ar1. The instruction actually adds the divisor to the ten's complement of the dividend until an overflow occurs. However, this is equivalent to subtracting until an "underflow" occurs. It is easier to understand the procedure if the discussion is in terms of "subtractions", but it should be kept in mind that what is really occurring with the instruction is repeated "complement-additions" until overflow. This process is what is meant by the term "subtractions until overflow".

The FDV instruction returns the number of subtractions without overflowing as a binary number in the B register (bits 0-3). The remaining bits in the B register (4-15) are cleared.[1] In effect, then B contains the next digit in the quotient.

This process is repeated for the number of digits to be calculated. After each FDV execution, the result of the overflow subtraction is left in Ar2. Since Ar2 does not contain the remainder, it is necessary to patch Ar2 so that it will contain the proper value for the next calculation. To get the proper value it is necessary to add Ar1 back into Ar2 to undo the results of the last subtraction (which caused the overflow).[2]

There is one case, however, where Ar2 does not need to be patched up, and this is when the remainder (Ar2) is zero. This situation implies not only that no patching up is needed, but also that the quotient is complete — no further digits need be calculated. It should be noted that the number of subtractions (which has been stored in the B register) is one count too small, thus B has to be incremented in this case so that it can be used as the last digit in the quotient.

1 Since bits 4-15 of the register are cleared during execution of the FDV instruction, you can't accumulate quotient digits there. After each digit is calculated, it is necessary that you store the digit as part of a quotient which you keep stored in another location.

2 This is equivalent to complementing Ar2, adding in Ar1, then complementing Ar2 again.

Thirteen-Digit Dividends

The largest difficulty in the algorithm is attempting to deal with those instances where the dividend has thirteen digits. This situation arises when you shift the remainder left a place. The most significant digit must be retained when it is non-zero so that the subtractions are subtracted from the proper amount.

This shifting can be accomplished with the MLY instruction. With the way that the MLY instruction operates, the left-most digit (D_1) ends up being shifted out of Ar2 into register A (in the lower 4 bits, 0-3). Thus, the thirteen-digit algorithm must accomodate the most significant digit residing in the A register and the twelve least significant digits in the Ar2 register. The use of FDV must now take this modified situation into account.

When the FDV instruction is executed, Arl is subtracted from Ar2 until an overflow occurs. When this overflow occurs, it is necessary to decrement A and keep subtracting (without patching up Ar2). Each time an overflow occurs, A must be decremented until finally an overflow occurs when A is 0. This can be handled very neatly within a small loop.

Another aspect of dealing with thirteen-digit dividends is the count placed in B with each execution of FDV. Since each overflow is a "successful" subtraction in the sense that is part of a proper count of subtractions (at least until A is 0), then that subtraction must be counted, too. The difficulty with this is that FDV does not count this last (overflowing) subtraction. The solution obviously is to add 1 to the value in the B register each time FDV causes an overflow. However, with the last overflow, being the "real" overflow, the 1 shouldn't be added in, so after adding it in (during the loop), you have to subtract it back out again (after leaving the loop). To further complicate matters, if you have a zero remainder, you have to add it right back in again.

For example, if there happened to be three uses of FDV for a certain quotient digit, you form the quotient digit as —

$$Q_n = \underset{\substack{\text{value after 1st}\\ \text{use of FDV}}}{(B + 1)} \quad + \underset{\substack{\text{value after 2nd}\\ \text{use of FDV}}}{(B + 1)} \quad + \underset{\substack{\text{value after final}\\ \text{use of FDV}}}{B}$$

If the same general situation produced a zero remainder, then the quotient digit is formed as —

$$Q_n = \underset{\substack{\text{value after 1st}\\ \text{use of FDV}}}{(B + 1)} \quad + \underset{\substack{\text{value after 2nd}\\ \text{use of FDV}}}{(B + 1)} \quad + \underset{\substack{\text{value after final}\\ \text{use of FDV}}}{(B + 1)}$$

Floating-Point Division Example

An example of a 13-digit division routine follows. The rules which it implements are —

1. Always increment the value returned in B after an FDV operation.
2. After incrementing B, check the contents of A. If non-zero, loop immediately, performing no other tests or activities.
3. When a quotient digit has been found (i.e., A is zero), check to see if the remainder is 0. If so, exit the division loop. Save the last digit found as part of the answer.
4. If the remainder is not 0, decrement the value of the last quotient digit found and save it as part of the answer. Then add back the divisor to the remainder.

The example does not include routines for testing and handling —

- signs
- division by zero
- exponents
- overflow
- rounding

These have to be handled in a real program before or after the division algorithm itself (as appropriate).

```
ISOURCE ! Some useful symbols
                .
                .
                .
ISOURCE Ar21:   EQU Ar2+1     ! First mantissa word
ISOURCE Ar22:   EQU Ar2+2     ! Second mantissa word
ISOURCE Ar23:   EQU Ar2+3     ! Third mantissa word
                .
                .
                .
ISOURCE ! Working area
ISOURCE Quotient:           BSS 5   ! Working storage for quotient
ISOURCE Quotient_1:         EQU Quotient+1   ! for quotient word 1
ISOURCE Quotient_2:         EQU Quotient+2   ! for quotient word 2
ISOURCE Quotient_3:         EQU Quotient+3   ! for quotient word 3
ISOURCE Quotient_4:         EQU Quotient+4   ! for quotient word 4
ISOURCE Quotient_ptr:       BSS 1   ! for quotient word 1
ISOURCE Digit_counter:      BSS 1   ! total digits (1-13)
ISOURCE Within_word_ctr:    BSS 1   ! digit counter (1-4)
                .
```

```
               URCE ! Dividend already in Ar2, divisor already in Ar1
ISOURCE Divide:                   ! START OF DIVISION LOOP
ISOURCE     LDA =Quotient_1
ISOURCE     STA Quotient_ptr
ISOURCE     CLR 4                 ! In case of early termination, zero
ISOURCE     CMY                   ! Complement the dividend
ISOURCE     LDA =13
ISOURCE     STA Digit_counter     ! Initializes digit count to 13
ISOURCE     LDA =0                ! Initialize FDV repetition counter to 1
ISOURCE !
ISOURCE Next_word:                ! WORKS ON NEXT SET OF 4 BCD DIGITS
ISOURCE     LDB =4
ISOURCE     STB Within_word_ctr   ! Initialize intermediate counter
ISOURCE !
ISOURCE Next_digit:               ! WORKS ON NEXT QUOTIENT DIGIT
ISOURCE     SBL 4                 ! Clear lower bits of B
ISOURCE     STB Quotient_ptr,I    ! Clear next storage word
ISOURCE !
ISOURCE        ! QUOTIENT CALCULATION
ISOURCE     FDV                   ! Ar2=Ar2+Ar1 until overflow
ISOURCE     ADB Quotient_ptr,I    ! Merge new digit with rest of answer
ISOURCE     ADB =1                ! Increment the new digit
ISOURCE     STB Quotient_ptr,I    ! Save this state of the answer
ISOURCE     RIA Fdv_loop          !   Decrement and loop if non-zero
ISOURCE !
ISOURCE ! Check for a zero remainder
ISOURCE !
ISOURCE     LDA Ar21
ISOURCE     IOR Ar22
ISOURCE     IOR Ar23
ISOURCE     SZA Zero_remainder
ISOURCE !
ISOURCE ! No zero remainder, so divide acaij* Bqp birst reqtore dividend,
          shift it left, and then find new FDV repetition count.
ISOURCE !
ISOURCE     CMY                   ! Decomplement remainder (Ar2)
ISOURCE     FXA                   ! Add back in divisor (Ar1)
ISOURCE     ADB =-1               ! Undo the increment
ISOURCE     STB Quotient_ptr,I    ! Save the corrected partial answer
ISOURCE     CMY                   ! Complement the dividend
ISOURCE     LDA =0                ! Clear A
ISOURCE     MLY                   ! Shift dividend left
ISOURCE     ADA =-9               ! Determine next repetition count

ISOURCE !
ISOURCE ! Bottom of loop maintenance follows
ISOURCE !
ISOURCE     DSZ Digit_counter     ! Decrement number of digits
ISOURCE     JMP Within_word
ISOURCE     JMP Done
ISOURCE !
ISOURCE Within_word:              ! DECREMENT POSITION WITHIN WORD
ISOURCE     DSZ Within_word_ctr
ISOURCE     JMP Next_digit
ISOURCE     ISZ Quotient_ptr
ISOURCE     JMP Next_word
ISOURCE !
ISOURCE Zero_remainder:           ! ZERO REMAINDER BEFORE 13th DIGIT?
ISOURCE     DSZ Digit_counter
ISOURCE     JMP Shift
ISOURCE     JMP Done
```

```
ISOURCE !
ISOURCE Shift_left:   SBL 4
ISOURCE Shift:        DSZ Within_word_ctr    ! Shift digits as necessary
ISOURCE               JMP Shift_left
ISOURCE !
ISOURCE Done:                          ! STORE AWAY THE RESULT
ISOURCE     STB Quotient_ptr,I         ! Store last digits of quotient
ISOURCE     LDA =Quotient
ISOURCE     LDB =Ar2
ISOURCE     XFR 4                      ! Transfer quotient from working storage
                        to Ar2
                                         to Ar2
ISOURCE     NRM
ISOURCE     SZB Continue               ! Go on, if all is OK
ISOURCE !
ISOURCE ! If leading digit of quotient was a zero, then old digit 13 must

          be saved as new digit 12
ISOURCE !
ISOURCE     LDA Quotient_4             ! Get digit 13
ISOURCE     AND =17B                   ! Lower 4 bits only (in case Quotient_4
                         is used elswehere for other thi
                                         is used elswehere for other things
ISOURCE     ADA Ar23                   ! Add in new digit (old digit 12 was 0)
ISOURCE     STA AR23                   ! Save the corrected quotient
ISOURCE ! Proceed to adjust exponent accordingly
            .
            .
            .
ISOURCE Continue:  ! Compute sign, etc.
```

Arithmetic Utilities

Now that you have been introduced to the complexities of BCD arithmetic and floating-point operations, this is the time to present an easier way of accomplishing these operations — the arithmetic utilities.

In order to make BASIC a useful programming tool, the operating system already contains a number of floating-point routines. Recognizing that BCD and floating-point arithmetic can be a difficult and laborious task to implement, the assembly language provides a utility by which the operating system mathematical routines can be accessed. There are also utilities for the conversion of numerical data types.

UTILITY: Rel_math

The Rel_math utility provides access to all of the system floating point routines and functions.

General Procedure: The utility is told the execution address of the desired routine or function and is also told the number of parameters. The parameters are floating-point values stored in full-precision form (4 words each). The result is a full-precision value.

Special Requirements:

- If one operand is passed to the utility, the **address** of the operand is stored in register Oper_1.
- If two operands are passed to the utility, the **address** of the **first** operand is stored in register Oper_1 (as above), and the **address** of the **second** operand is stored in register Oper_2.
- The **address** where the result should be stored must be stored in the register Result.
- All operands and the result are full-precision values and require 4 words each.
- Values passed must make sense for the routine or function being called (e.g., Oper_2 should not point to a value of 0 when calling the division routine), or else an error results.
- The storage areas for the operands and the result must reside either in the ICOM region or in the Base_page register. Specifically, they cannot be specified as Ar1 or Ar2.

Calling Procedure:

1. Assure that Oper_1, Oper_2, and Result contain the proper addresses as above.
2. Load register A with the number of parameters required for the routine or function (see the table on next page). Note that some routines require this number to be complemented.
3. Load register B with the execution address of the routine or function (see the table on the next page).
4. Call the utility.

Exit Conditions:

- The result is placed into the 4 words starting at the address pointed to by the Result register.
- Register A contains 0 if no error is encountered during execution of the utility.
- Register A contains the error number should an error be encountered during execution of the utility.

Rel_math Utility Routines, Addresses, and Parameters[2]

Operands (LDA =)	Octal Execution Routine	Address (LDB =)
Addition (Oper_1 + Oper_2)	146721B	2
Subtraction (Oper_1 − Oper_2)	146717B	2
Multiplication (Oper_1 * Oper_2)	147037B	2
Division (Oper_1 / Oper_2)	147155B	2
Exponentiation (Oper_1 ∧ Oper_2)	34276B	2
Oper_1 DIV Oper_2	33026B	2
Oper_1 MOD Oper_2	33157B	2
SQR	31450B	1
INT	33071B	1
FRACT	33262B	1
EXP	34173B	1
LOG	34203B	1
LGT	34263B	1
PROUND (Oper_1, Oper_2)	32225B	−2
DROUND (Oper_1, Oper_2)	32247B	−2
ABS	33054B	1
SGN	33651B	1
PI	36267B	0
RND	33607B	0
RES	36307B	0
TYP[1]	6753B	1
SIN	34213B	1
COS	34224B	1
TAN	34151B	1
ASN	34235B	1
ACS	34250B	1
ATN	34161B	1
ERRL[1]	61765B	0
ERRN[1]	61753B	0
DECIMAL[1 3]	162067B	1
IADR (Oper_1, Oper_2)[3]	162230B	−2
IMEM (Oper_1, Oper_2)[3]	162211B	−2
OCTAL[3]	162146B	1
Oper_1 AND Oper_2	32042B	2
Oper_1 OR Oper_2	32057B	2
Oper_1 EXOR Oper_2	32025B	2
NOT	32071B	1
Oper_1 < Oper_2	32077B	2
Oper_1 < = Oper_2	32105B	2
Oper_1 < > Oper_2	32137B	2
Oper_1 = Oper_2	32127B	2
Oper_1 > = Oper_2	32121B	2
Oper_1 > Oper_2	32113B	2
MAX (Oper_1, Oper_2)	33744B	−2
MIN (Oper_1, Oper_2)	33704B	−2

Table 1. Routines, Addresses, and Parameters for Rel_Math Utility

1 These functions return an integer value which is stored in the second word of the four words reserved by Result.

2 See the System 45 Operating and Programming manual for a detailed explanation of the function of each of these routines.

3 See the appropriate section of this manual for a detailed explanation of the function of each of these routines.

By way of example, suppose you have established two full-precision values which need to be multiplied. The call to the Rel_math utility to accomplish the multiplication would look similar to this —

```
ISOURCE  ! Working storage
             .
             .
             .
ISOURCE Operand_1:  BSS 4
ISOURCE Operand_2:  BSS 4
ISOURCE Product:    BSS 4
             .
             .
             .
ISOURCE Multiply:  ! MULTIPLY THE OPERANDS
ISOURCE     LDA =Operand_1
ISOURCE     STA Oper_1
ISOURCE     LDA =Operand_2
ISOURCE     STA Oper_2
ISOURCE     LDA =Product
ISOURCE     STA Result
ISOURCE     LDA =2            ! Call the multiply routine
ISOURCE     LDB =147037B
ISOURCE     JSM Rel_math
ISOURCE     SZA *+2           ! Test for any errors
ISOURCE     JSM Error_exit    ! Error encountered, so leave
             .
             .
             .
```

Note in the last line of the example the call to the Error_exit utility is made when register A is not zero. When this occurs, A contains the error number of the error encountered — ready-made for calling the Error_exit utility.

UTILITY: Rel_to_int

The Rel_to_int utility provides for the conversion of a full-precision value into an integer.

General Procedure: The utility is given the address of the location of the full-precision value and the address of the location where the integer is to be stored.

Special Requirements: The full-precision value must be within the range of integers (− 32 768 to + 32 767).

Calling Procedure:

1. Store the address of the full-precision value into register Oper_1.
2. Store the address where the integer is to be stored into register Result.
3. Call the utility.

Exit Conditions: The overflow bit in the processor is set if the result is outside the range of integers.

An example —

```
ISOURCE ! Working storage
ISOURCE Operand:  BSS 4    ! Contains a full-precision value
ISOURCE Value:    BSS 1    ! Contains an integer value
                   .
                   .
                   .
ISOURCE           LDA =Operand
ISOURCE           STA Oper_1
ISOURCE           LDA =Value
ISOURCE           STA Result
ISOURCE           JSM Rel_to_int  ! Convert real to integer
ISOURCE           SOC *+3         ! Check for overflow
ISOURCE           LDA =20         ! Set error number to 20
ISOURCE           JSM Error_exit  !   and take error exit
                   .
                   .
                   .
```

UTILITY: Rel_to_sho

The Rel_to_sho utility provides for the conversion of a full-precision value into a short-precision one.

General Procedure: The utility is given the address of the location of the full-precision value and the address of the location where the short-precision value is to be stored.

Special Requirements: A short-precision value requires 2 words to be stored.

Calling Procedure:

1. Store the address of the full-precision value into register Oper_1.
2. Store the address of the storage area for the short-precision value into register Result.
3. Call the utility.

Exit Conditions: The overflow bit in the processor is set if the result is outside the range of integers.

An example —

```
ISOURCE ! Working storage
ISOURCE Operand:  BSS 4    ! Contains full-precision value
ISOURCE Value:    BSS 2    ! Contains short-precision value
                   .
                   .
                   .
ISOURCE           LDA =Operand
ISOURCE           STA Oper_1
ISOURCE           LDA =Value
ISOURCE           STA Result
ISOURCE           JSM Rel_to_sho  ! Convert full to short
                   .
                   .
                   .
```

UTILITY: Int_to_rel

The Int_to_rel utility provides for the conversion of an integer into a full-precision value.

General Procedure: The utility is given the address of the location of the integer and the address where the full-precision value is to be stored.

Calling Procedure:

1. Store the address of the integer into register Oper_1.
2. Store the address of the storage area for the full-precision value into register Result.
3. Call the utility.

Exit Conditions: The overflow bit in the processor is set if the result is outside the range of integers.

An example —

```
ISOURCE ! Working storage
ISOURCE Operand:  BSS 1    ! Contains an integer
ISOURCE Value:    BSS 4    ! Contains full-precision value
                   .
                   .
                   .
ISOURCE           LDA =Operand
ISOURCE           STA Oper_1
ISOURCE           LDA =Value
ISOURCE           STA Result
ISOURCE           JSM Int_to_rel  ! Convert integer to real
                   .
                   .
                   .
```

UTILITY: Sho_to_rel

The Sho_to_rel utility provides for the conversion of a short-precision value into a full-precision one.

General Procedure: The utility is given the address of the location of the short-precision value and the address of where the full-precision value is to be stored.

Calling Procedure:

1. Store the address of the short-precision value into register Oper_1.
2. Store the address of the storage area for the full-precision value into register Result.
3. Call the utility.

Exit Conditions: No special exit conditions.

An example —

```
ISOURCE ! Working storage
ISOURCE Operand:  BSS 2   ! Contains short-precision value
ISOURCE Value:    BSS 4   ! Contains full-precision value
                  .
                  .
                  .
ISOURCE           LDA =Operand
ISOURCE           STA Oper_1
ISOURCE           LDA =Value
ISOURCE           STA Result
ISOURCE           JSM Sho_to_rel   ! Convert short to real
                  .
                  .
                  .
```

Chapter 6
Communication Between BASIC and Assembly Language

Summary: This chapter discusses the techniques used to pass information to and from assembly language programs. Calling assembly language routines and passing parameters are presented, along with issues involved in using common. Applicable utilities are also discussed.

Once assembly language programs have been written, they are executed using the ICALL statement. This statement is very similar to BASIC's CALL statement for subroutines. In fact, the function it performs is nearly identical in effect — the only difference is that the target subroutine has been written in assembly language instead of in BASIC. The ICALL statement also provides a means to pass data between BASIC and assembly programs through its argument list. Data can also be passed through common.

The ICALL Statement

There are two ways to execute an assembly language routine. One way is as an interrupt service routine when an interrupt occurs on the select code to which the service routine has been linked. This technique is discussed in Chapter 7. The other way is through executing an ICALL statement, either in a BASIC program or from the keyboard.

The syntax of the statement is —

```
ICALL {routine name} [ ( {argument} [, {argument} [,...] ] ) ]
```

{routine name} is the name of the assembly language routine to be executed. {argument} is a data item that has the same characteristics as an argument in BASIC's CALL statement — there may be constants, variables, or expressions. (How these items correspond to instructions in the assembly language will be discussed shortly.)

By way of example, suppose that you have an ICALL that is being used to call a sort routine and the routine was written in such a way as to require two arguments be passed to it — an array to be sorted and the number of elements to be sorted (in that order). Then the following would be valid calls to that routine —

```
ICALL Sort(Test(*),100)
ICALL Sort(Test$(*),Number)
ICALL Sort(Value(*),Events DIV 2)
```

Upon executing the ICALL statement, execution in a program transfers to the routine named. Upon executing a RET 1 instruction from the main assembly language program, execution returns to the BASIC statement which follows the ICALL. This is identical in effect to the CALL statement in BASIC.

In executing the statement from the keyboard, the routine named is executed just as if it were used in a program. Upon return from the routine, control is passed back to the keyboard. This is unlike BASIC's CALL statement, which cannot be executed from the keyboard.

To execute a routine, whether it be from a program or from the keyboard, its object code must currently reside in the ICOM region.

Corresponding Assembly Language Statements

When the ICALL is executed, it references a routine in the object code. When the module containing the routine was assembled, it declared that routine name as a "subroutine" entry point. ("Subroutine" and "routine" are synonymous in this context.) This is done with a SUB pseudo-instruction and a label.

When a SUB pseudo-instruction appears in the source code, it is a signal to the assembler that a subroutine entry point follows. Then the first machine instruction must have a label and that label becomes the routine name. If the label is missing, an error results (assembly-time "SQ" error).

For example, in the above examples of ICALL, the Sort routine could have been defined by the sequence —

```
ISOURCE        SUB
ISOURCE Sort:  LDA =Array_info
```

except that there are arguments involved. (That exception is discussed in a moment.) The joint use of these two statements results in the label "Sort" being identified as a routine name, referenceable with an ICALL statement.

In general, no machine instructions or code-generating pseudo-instructions can be inserted between a SUB pseudo-instruction and the instruction containing the routine name. An exception to this exists when arguments are involved in a call.

Arguments

When a value is placed into an ICALL statement to be sent down to an assembly language routine, that value is called an "argument" (like the argument of a mathematical function). The corresponding structure on the assembly language side is called a "parameter". A parameter "declaration" is an assembly pseudo-instruction by which a parameter is created.

When a routine is to be called with arguments, a parameter declaration pseudo-instruction is required for each one of the arguments. These declarations appear between the SUB pseudo-instruction and the instruction containing the routine name.

Thus, when there is a call like —

```
ICALL Sort(Test$(*),100)
```

the corresponding assembly language entry looks like —

```
ISOURCE        SUB
ISOURCE        STR (*)
ISOURCE        REL
ISOURCE Sort:  LDA =Array_info
```

To accommodate the two arguments, two parameter declarations had to appear between the SUB instruction and the entry point. (In this example, they were the STR and REL declarations.) These declarations may even have labels of their own —

```
ISOURCE               SUB
ISOURCE Parameter_1:  STR (*)
ISOURCE Parameter_2:  REL
ISOURCE Sort:         LDA =Array_info
```

The appearance of these labels does not affect the fact that "Sort" is the name of the routine.

Parameter declarations have "types" just like variables. These types have to correspond to the "types" of the arguments used in the ICALL. The declarations and their types are —

```
INT        meaning integer
REL        meaning full-precision
SHO        meaning short-precision
STR        meaning string
FIL        meaning a file number
```

In the above example, STR had to be used as the first parameter declaration because the first argument was a string. Similarly, REL had to be the second declaration because the second argument was a numeric expression (which is always full-precision).

When an array is to be passed, the declaration is followed by an "array identifier" — (*). Thus, when arrays are involved, the declarations appear as —

```
INT(*)     meaning an integer array
REL(*)     meaning a full-precision array
SHO(*)     meaning a short-precision array
STR(*)     meaning a string array
```

File numbers are not passed in arrays, so that the declaration FIL cannot be followed by an array identifier. When passing file numbers to assembly language routines, the file number must be preceded by a "#" character.

ICALL Sort (#File_number,Entries,Type)

Failure to include the "#" before the file number or file number variable results in an error.

Since the example call above uses a string array as the first argument, the corresponding assembly language parameter declaration uses an array identifier after STR.

The parameter declarations are associated with the arguments in the ICALL in the same order. If the types do not match when the ICALL is executed, an error occurs (number 8).

So, if the subroutine entry looks like —

```
ISOURCE          SUB
ISOURCE          STR (*)
ISOURCE          REL
ISOURCE Sort:    LDA =Array_info
```

then this ICALL executes properly —

```
ICALL Sort(Test$(*),100)
```

but these ICALLs result in run-time errors —

```
ICALL Sort(Test$,100)
ICALL Sort(Test(*),100)
ICALL Sort(Test$(*),"ASCENDING")
```

Each declaration reserves three words in the object code upon assembly. As a result of the ICALL execution, these words contain a descriptor of the corresponding argument. These descriptors are used by the utilities for fetching and storing values. Thus, in the Sort calling example above, when the ICALL is executed, a descriptor for Test$(*) is stored in the three words starting at Parameter_1. Similarly, a descriptor for the constant 100 is stored in the three words starting at Parameter_2.

The types discussed here do not apply just to simple variables, arrays, and constants. They also apply to single elements of arrays and expressions. If you have a STR parameter declaration, for example, any of the following would be valid as arguments in the ICALL statement —

```
Test$(1)
CHR$(127)&Test$
RPT$("A",20)
Test$[1,Stop]
```

It is similar for numerical expressions.

The number of arguments passed by an ICALL statement must be no more than the number of parameter declarations in the subroutine entry. There may be fewer, however. The actual number passed is stored in the word reserved by the SUB pseudo-instruction.

Unlike the CALL statement in BASIC, the ICALL statement can be executed from the keyboard. In doing so, any variables used as arguments pass their current values to the routine.

"Blind" Parameters

With explicit parameter declarations, an error occurs if a different type of variable or expression is passed. In many cases, the error is desirable — you do not want different types of arguments corresponding to a single parameter declaration. But in other cases, the error might not be as desirable. Take the example of a sort. You might want the sort to have the capability of sorting any type of array. You have two choices in that case — you can make different routines, each with the appropriate declarations, or you can use a single entry point and the ANY parameter declaration.

The ANY declaration —

```
ANY
```

is "blind" to the type of the corresponding argument in the ICALL statement. When used, it accepts any type of argument as valid — string, full-precision, short-precision, integer, file number, array. The descriptor for the argument is stored in the three words set aside, just as in the other declarations.

Now, if your entry looks like —

```
ISOURCE        SUB
ISOURCE        ANY
ISOURCE        REL
ISOURCE Sort:  LDA =Array_info
```

then any of the following calls would be valid —

```
ICALL Sort(Test$(*),100)
ICALL Sort(Test(*),100)
ICALL Sort(Test$,100)
ICALL Sort(Test,100)
ICALL Sort(#1,100)
```

When using the ANY declaration, it becomes the responsibility of your assembly language routine to determine what is a valid parameter and what is not. You lose the automatic type-checking available with explicit declarations. Techniques for doing this are discussed in the next section.

Getting Information on Arguments

When an ICALL is executed with an argument, and the corresponding parameter is blind, then it may be necessary for the purposes of your routine to know what type of argument is actually passed. This need can be present even when one of the explicit type declarations is used, since an expression or constant can be passed as easily as a variable.

A utility has been provided for obtaining this information, along with other "vital statistics" which may be useful to know during the execution of your routine. Before describing the utility itself, let's look at the information which it can provide you about an argument.

The information returned by the utility is stored in an area which you set aside for it. The size of the area can vary from 3 words to 39. The information, when returned, is in the following form —

Word #	Description
0	Argument type (see description later)
1	Number of dimensions (0 for non-arrays)
2	Size, in number of bytes (dimensioned length, for strings)
(for arrays only:)	
3	Total number of elements in array
4	Two's complement of the lower bound of first dimension
5	Absolute size of first dimension (upper bound − lower + 1)
6	Two's complement of the lower bound of second dimension (if any)
7	Absolute size of second dimension
8	Two's complement of the lower bound of third dimension (if any)
9	Absolute size of third dimension
10	Two's complement of the lower bound of fourth dimension (if any)
11	Absolute size of fourth dimension
12	Two's complement of the lower bound of fifth dimension (if any)
13	Absolute size of fifth dimension
14	Two's complement of the lower bound of sixth dimension (if any)
15	Absolute size of sixth dimension
16	Element offset (from the first element)
17	Size, in words, of each element (dimensioned length, for strings)
(dependent upon memory size of your machine:)	
18-20	Pointer parameters
21-23	Pointer parameters (only for machines over 64K bytes)
24-26	Pointer parameters (only for machines over 128K bytes)
27-29	Pointer parameters (only for machines over 192K bytes)
30-32	Pointer parameters (only for machines over 256K bytes)
33-35	Pointer parameters (only for machines over 320K bytes)
36-38	Pointer parameters (only for machines over 384K bytes)

The argument type returned in word 0 is as follows —

Value	Type
0	String expression
1	Full-precision expression
2	Short-precision expression
3	Integer expression
4	String simple variable
5	Full-precision simple variable
6	Short-precision simple variable
7	Integer simple variable
8	String array element
9	Full-precision array element
10	Short-precision array element
11	Integer array element
12	String array
13	Full-precision array
14	Short-precision array
15	Integer array
16	File number

The size, in bytes, will be one of the following values —

For an integer	2
Short-precision	4
Full-precision	8
String variables	dimensioned length
String expressions	actual length

The utility which retrieves all this information is called "Get_info".

UTILITY: Get_info

General Procedure: The utility is given the address where the information is to be returned and the address of the parameter declaration. It returns with the information on the argument in the ICALL corresponding to the parameter declaration.

Special Requirements:

- The location where it is to store the information must be adequate to hold all that may be returned. For non-arrays, 3 words will suffice. For arrays, up to 39 words may be required (as above). If you are writing a general routine, it may be wise to play it safe by setting aside a full 39 words.
- An argument must have been passed by the ICALL (in the case of parameters) or a corresponding BASIC COM declaration must exist (in the case of common declarations).[1]

Calling Procedure:

1. Load register A with the address of the storage area for the information to be returned.
2. Load register B with the address of the parameter declaration corresponding to the desired argument.
3. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the JSM. Since there are no error exits, and there is no requirement that there be as many arguments as there are parameter declarations, an argument must actually have been passed by the ICALL in order for the utility to work correctly.

Following up on the example in the previous section, suppose the first thing that the Sort routine does is check to see if the first parameter passed is an array. Then, by using the Get_info utility, it is possible to have the instructions look as follows —

```
ISOURCE Array_info:  BSS 39
ISOURCE          SUB
ISOURCE Array:   ANY
ISOURCE Number:  REL
ISOURCE Sort:    LDA =Array_info    ! Get info on argument
ISOURCE          LDB =Array
ISOURCE          JSM Get_info
ISOURCE          LDA Array_info     ! Get the argument's type
ISOURCE          CPA =16            ! Is it a file number?
ISOURCE          JMP Error_8        !   Yes, indicate error 8
ISOURCE          ADA =-12
ISOURCE          SAP *+2            ! An array (types 12-15)?
ISOURCE          JMP Error_8        !   No, indicate error 8
```

1 This and the following utilities are also used to access variables in the common area. An explanation of BASIC COM declaration is found in the section of this chapter entitled "Using Common".

The array information returned by the Get_info utility is used for accessing elements in arrays passed as arguments. It is used by the element-retrieval utilities described in a later section of this chapter. Once retrieved, the information is usable any number of times for accessing the array associated with it. It is not necessary to retrieve the information every time you access an array, as long as you have not altered the information (except the pointer) between accesses.

The seventeenth word of the array information (word 16 on the chart) is reserved to hold the offset from the start of the array of the element to be accessed. Therefore, it is permissible (indeed, it is **necessary**) to alter the contents of that location to indicate which element in the array you wish to retrieve. None of the other words returned by the utility should be changed.

An example of how to calculate array offsets is given here. It is convenient to give labels to some of the words of information returned by the Get_info utility.

```
ISOURCE Storage:    BSS 4
ISOURCE Lower1:     BSS 1
ISOURCE Size1:      BSS 1
ISOURCE Lower2:     BSS 1
ISOURCE Size2:      BSS 1
ISOURCE Lower3:     BSS 1
ISOURCE Size3:      BSS 1
ISOURCE Lower4:     BSS 1
ISOURCE Size4:      BSS 1
ISOURCE Lower5:     BSS 1
ISOURCE Size5:      BSS 1
ISOURCE Lower6:     BSS 1
ISOURCE Size6:      BSS 1
ISOURCE Element:    BSS 1
ISOURCE Remainder:  BSS 22
```

In addition, space is reserved for up to six array subscript indices.

```
ISOURCE Index1:     BSS 1
ISOURCE Index2:     BSS 1
ISOURCE Index3:     BSS 1
ISOURCE Index4:     BSS 1
ISOURCE Index5:     BSS 1
ISOURCE Index6:     BSS 1
```

For a six-dimensional array, the computation of the element offset (word 16 returned by "Get_info") is –

```
ISOURCE    LDA Index1      ! For all dimensions.
ISOURCE    ADA Lower1      !
ISOURCE    LDB Size2        ! For 2- and higher dimensions.
ISOURCE    MPY              !
ISOURCE    ADA Index2       !
ISOURCE    ADA Lower2       !
ISOURCE    LDB Size3         ! For 3- and higher dimensions.
ISOURCE    MPY               !
ISOURCE    ADA Index3        !
ISOURCE    ADA Lower3        !
ISOURCE    LDB Size4          ! For 4- and higher dimensions.
ISOURCE    MPY                !
ISOURCE    ADA Index4         !
ISOURCE    ADA Lower4         !
ISOURCE    LDB Size5           ! For 5- and higher dimensions.
ISOURCE    MPY                 !
ISOURCE    ADA Index5          !
ISOURCE    ADA Lower5          !
ISOURCE    LDB Size6            ! For 6 dimensions.
ISOURCE    MPY                  !
ISOURCE    ADA Index6           !
ISOURCE    ADA Lower6           !
ISOURCE    STA Element     ! For all dimensions.
```

For an array with a smaller number of dimensions, the operations involving the higher subscripts can be omitted.

Note that the indices in this example were not checked against the array bounds. Following is an example of a program segment which checks the index against the upper and lower bounds of a one-dimensional array:

```
ISOURCE    LDA Index1
ISOURCE    ADA Lower1
ISOURCE    SAM Lower_bound_err
ISOURCE    LDB Size1
ISOURCE    TCB
ISOURCE    ADB A
ISOURCE    SBP Upper_bound_err
ISOURCE    STA Element
```

There is no need to check for overflow, since the element offset is never greater than 32 767.

When making multiple accesses with the same information, caution should be taken if an array is involved. The information returned by Get_info is a copy of the system information and as such remains valid for as long as the ICALL lasts. However, as soon as an ICALL completes, the system has an opportunity to change its own information (via REDIM or subprogram recursion). This renders the original data returned by Get_info invalid.

Thus, while it is sufficient to call Get_info only once during an ICALL (independent of the number of times the information is used), it is advisable to use Get_info during each ICALL rather than attempting to retain the information from one ICALL to the next.

Retrieving the Value of an Argument

At some point during execution of your assembly language routine, you may want to retrieve the value of an argument so that you can use it in your processing. By doing so, you accomplish one of the methods of communicating with assembly language — namely, passing a value **to** the assembly language routine from BASIC.

There are a number of utilities for this purpose. The one to use is dependent upon the type of argument passed. The utilities available are —

Name	Used For	Example Parameters
Get_value	Simple variables, expressions, individual elements of arrays passed as arguments, and file numbers	Alpha,Z*SIN(Z),A$,"ABC",B$(10),Array(2,3),#5
Get_element	Elements (from arrays passed as arguments)	Array(*),Z$(*)
Get_bytes	Substrings of strings passed as arguments either as simple string variables, expressions, or individual elements of arrays passed as arguments	"DEF",String$,B$&C$,Z$(2,3),Z$[5,6]
Get_elem_bytes	Substrings of individual elements (from string arrays passed as arguments)	Z$(*)

How each of these utilities is used is described in the immediately following pages.

UTILITY: Get_value

General Procedure: The utility is given the address of the parameter declaration and the address where the value of the argument is to be stored. It returns with that value stored in the indicated area. It works on simple variables, expressions, strings, and individual elements of arrays (passed as arguments) of any type.

Special Requirements:

- The storage area set aside for the value must be large enough to hold the value. The size of the storage area must be —

for a file number	1 word
for an integer value	1 word
for a short-precision value	2 words
for a full-precision value	4 words
for a string	maximum length in bytes ÷ 2 + 1 word (+ 1 additional word if the maximum string length is odd)

- An argument must have been passed by the ICALL (in the case of parameters) or a corresponding BASIC COM declaration must exist (in the case of common declarations).
- The storage area must lie within the ICOM region.

Calling Procedure:

1. Load register A with the address of the storage area for the value.
2. Load register B with the address of the parameter declaration.
3. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

In the case that it is used to pass a string value, the Get_value utility returns the entire dimensioned string (which includes all characters between the current length and the dimensioned length of the string).

Here is an example call to the utility, retrieving information from a full-precision argument —

```
ISOURCE Value:      BSS 4
ISOURCE             SUB
ISOURCE Parameter:  REL
ISOURCE Entry:      LDA =Value
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_value
```

UTILITY: Get_element

General Procedure: This is similar to the "Get_value" utility. This utility retrieves a value from an element of an array passed as an argument. It works on arrays of any type.

Special Requirements:

- The storage area set aside for the value must be large enough to hold the value. The size of the storage area must be —

for an integer	1 word
for a short-precision value	2 words
for a full-precision value	4 words
for a string	maximum length in bytes÷2 + 1 word (+ 1 additional word if the maximum string length is odd)

- The array information must be retrieved with the "Get_info" utility before calling this utility.
- The offset of the element in the array must be correct in the array information (word 16 returned by "Get_info"). It should be remembered that the offset of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays. The offset is in terms of number of elements.[1]
- The storage area must lie within the ICOM region.

Calling Procedure:

1. Store the element offset within the array information (word 16 returned by "Get-info").
2. Load register A with the address of the storage area for the value.
3. Load register B with the address of word 0 of the information returned by the "Get_info" utility (see description of that utility).
4. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

[1] See the description of calculating array offsets under the "Get_info" utility.

Here is an example call, retrieving the third element (relative element 2) of an integer array and placing it into Value —

```
ISOURCE Value:      BSS 1
ISOURCE Array_info: BSS 39
ISOURCE Element:    EQU Array_info+16 ! Element offset
ISOURCE             SUB
ISOURCE Parameter:  INT (*)
ISOURCE Entry:      LDA =Array_info   ! Get the array info
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_info
ISOURCE             LDA =2            ! Set element offset to 2
ISOURCE             STA Element
ISOURCE             LDA =Value        ! Get the value
ISOURCE             LDB =Array_info
ISOURCE             JSM Get_element
```

UTILITY: Get_bytes

General Procedure: This is similar to the "Get_value" utility. This utility retrieves a substring of a string passed as an argument, having been given the starting byte and the number of bytes to be retrieved.

Special Requirements:

- The storage area set aside for the substring must be large enough to hold all of the substring. This includes not only the string itself, but also two extra words. Remember, a word holds two characters.
- A string must have been passed by the ICALL for the utility to work properly.
- The storage area must lie within the ICOM region.

Calling Procedure:

1. Store the number of the starting **byte** of the substring desired into the first word of the storage area set aside for the substring. (Note that bytes 0 and 1 are the length word of the string.)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Load register A with the address of the storage area.
4. Load register B with the address of the parameter declaration.
5. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call. The substring is returned starting with the third word of the storage area. (Note: Since the second word contains the length of the substring, you have a string data structure starting with the second word!)

For example —

```
ISOURCE Value:     DAT 2    ! 1st character (ignore length)
ISOURCE            DAT 10   ! Transfer 10 characters
ISOURCE            BSS 5    ! Substring storage area
ISOURCE            SUB
ISOURCE Parameter: STR
ISOURCE Entry:     LDA =Value   ! Info already stored
ISOURCE            LDB =Parameter
ISOURCE            JSM Get_bytes
```

In this example, Value is the storage area. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string would be transferred. Of course, the original string must contain at least 10 characters — or the bytes which are returned may be nonsense. Why was the value 2 stored as the byte number? Because bytes in a string are numbered starting with 0, and bytes 0 and 1 contain the length of the string (see "Data Structures" in Chapter 3).

UTILITY: Get_elem_bytes

General Procedure: This is a combination of the "Get_element" and "Get_bytes" utilities. This utility retrieves a substring of an element of a string array passed as an argument. The utility is given the starting byte and the number of bytes to be retrieved.

Special Requirements:

- The storage area set aside for the substring must be large enough to hold all of it. This includes not only the string itself, but also two extra words. Remember, a word holds two characters.
- The array information must be retrieved with the "Get_info" utility before calling this utility.
- The offset of the element in the array must be correct in the array information (word 16 returned by "Get_info"). It should be remembered that the offset of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays. The offset is in terms of number of elements.[1]
- The storage area must lie within the ICOM region.

1 See the description of calculating array offsets under the "Get_info" utility.

Calling Procedure:

1. Store the number of the starting byte of the substring desired into the first word of the storage area set aside for the substring. (Note that bytes 0 and 1 are the length word of the string.)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Store the offset within the array information.
4. Load register A with the address of the storage area for the value.
5. Load register B with the address of word 0 of the information returned by the "Get_info" utility (see description of that utility).
6. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call. The substring is returned starting with the third word of the storage area. (Note: since the second word contains the length of the substring, you have a string data structure starting with the second word!)

For example —

```
ISOURCE Value:      DAT 2     ! 1st character (ignore length)
ISOURCE             DAT 10    ! Transfer 10 characters
ISOURCE             BSS 5     ! Substring storage area
ISOURCE Array_info:BSS 39     ! Array information
ISOURCE Element:    EQU Array_info+16 ! Element offset
ISOURCE             SUB
ISOURCE Parameter: STR (*)
                     .
                     .
                     .
ISOURCE Entry:      LDA =Array_info    ! Get the array pointer
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_info
ISOURCE             LDA =2             ! Set element offset to 2
ISOURCE             STA Element
                     .
                     .
                     .
ISOURCE             LDA =Value         ! Info already stored
ISOURCE             LDB =Array_info
ISOURCE             JSM Get_elem_bytes
```

In this example, Value is the storage area. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string element are transferred. Of course, the string element must contain at least 10 characters — or the bytes which are returned may be nonsense.

Changing the Value of an Argument

At some point during the execution of your assembly language routine, you might want to accomplish the other half of this method of communication with BASIC — namely, changing the value of a BASIC variable which is used as an argument, in effect changing the value of a BASIC variable from the assembly language routine.

As with retrieving a value, there are a number of utilities available for changing a value. The one to use is dependent upon the type of argument passed. The utilities available are —

Name	Used For	Example Parameters
Put_value	Simple variables, strings and individual elements of arrays passed as arguments	Alpha,A$,B$(10),Array(2,3)
Put_element	Elements (from arrays passed as arguments)	Array(*),Z$(*)
Put_bytes	Substrings of strings passed as arguments either as simple variables or as individual elements of arrays passed as arguments.	String $,Z$(2,3)
Put_elem_bytes	Substrings of elements (from string arrays passed as arguments)	Z$(*)

Note that these utilities modify variables existing in the BASIC environment. They do not modify the length of the variables as dimensioned in BASIC.

How each of these utilities is used is described in the immediately following pages.

UTILITY: Put_value

General Procedure: The utility is given the address of the parameter declaration and the address of the value. It changes the value of the BASIC variable associated with the parameter. It works only on simple variables, expression strings, and individual elements of arrays (passed as arguments) of any type.

Special Requirements:

- The value must have the appropriate data structure for the data type of the argument (see "Data Structures" in Chapter 3).
- An actual argument must have been passed by the ICALL for the utility to work properly.

Calling Procedure:

1. Load register A with the address of the storage area of the value.
2. Load register B with the address of the parameter declaration.
3. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

Here is an example call to the utility, passing information to an integer argument —

```
ISOURCE Value:      BSS 1
ISOURCE             SUB
ISOURCE Parameter:  INT
                     .
                     .
                     .
ISOURCE             LDA =Value
ISOURCE             LDB =Parameter
ISOURCE             JSM Put_value
```

Here is an additional example demonstrating string passing –

```
 COM S$
  .
  .
  .
ISOURCE              EXT Put_value
ISOURCE Value:       DAT 6,"STRING"
ISOURCE              COM
ISOURCE Com1:        STR
                      .
                      .
                      .
ISOURCE              LDA =Value
ISOURCE              LDB =Com1
ISOURCE              JSM Put_value
                      .
                      .
```

UTILITY: Put_element

General Procedure: This is similar to the "Put_value" utility. This utility changes the value of a single element in an array passed as an argument. It works on arrays of any type.

Special Requirements:

- The value must have the appropriate data structure for the data type of the argument (see "Data Structures" in Chapter 3).
- The array information must be retrieved with the "Get_info" utility before calling this utility.
- The offset of the element in the array must be correct in the array information for the array (word 16 returned by "Get_info"). It should be remembered that the relative element number of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays.
- The storage area must lie within the ICOM region.

Calling Procedure:

1. Store the element offset into the array information (word 16).
2. Load register A with the address of the storage area for the value.
3. Load register B with the address of word 0 of the information returned by the "Get_info" utility (see description of that utility).
4. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

Here is an example call, storing information from Value into element 0 of an integer array —

```
ISOURCE Value:      BSS 1
ISOURCE Array_info:BSS 39
ISOURCE Element:    EQU Array_info+16  ! Element offset
ISOURCE             SUB
ISOURCE Parameter:  INT (*)
                     .
                     .
                     .
ISOURCE             LDA =Array_info    ! Get array info
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_info
ISOURCE             LDA =0             ! Set offset to 0
ISOURCE             STA Element
                     .
                     .
                     .
ISOURCE             STA Value          ! Change the value
ISOURCE             LDA =Value
ISOURCE             LDB =Array_info
ISOURCE             JSM Put_element
```

UTILITY: Put_bytes

General Procedure: This is similar to the "Put_value" utility. This utility changes the value of a substring which is part of a string variable or an individual element of a string array, having been given the starting byte and the number of bytes to be changed as well as the new characters.

Special Requirments:

- The bytes to be transferred are preceded by two words in the storage area. The two words contain the starting byte for the substring and the number of bytes to be transferred.
- A string variable or an element of a string array must have been passed as an argument for the utility to work properly.

Calling Procedure:

1. Store the number of the starting **byte** of the substring to be changed into the first word of the storage area. (Note that bytes 0 and 1 are the length word of the string)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Load register A with the address of the storage area.
4. Load register B with the address of the parameter declaration.
5. Call the utility.

Exit Conditions: There are no error exits from the utility, so it always returns to the instruction following the call.

For example —

```
ISOURCE Value:     DAT 2      ! 1st character (ignore length)
ISOURCE            DAT 10     ! Transfer 10 characters
ISOURCE            BSS 5      ! Substring storage area
ISOURCE            SUB
ISOURCE Parameter: STR
                    .
                    .
                    .
ISOURCE            LDA =Value     ! Other info already saved
ISOURCE            LDB =Parameter
ISOURCE            JSM Put_bytes
```

In this example, Value is the storage area containing the string to be transferred. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string are changed. Why was the value 2 stored as the byte number? Because bytes in a string are numbered starting with 0, and bytes 0 and 1 contain the length of the string (see "Data Structures" in Chapter 3).

UTILITY: Put_elem_bytes

General Procedure: This is a combination of the "Put_element" and "Put_bytes" utilities. This utility changes a substring of an element in a string array which has been passed as an argument. The utility is given the starting byte and the number of bytes to be transferred.

Special Requirements:

- The bytes to be transferred are preceded by two words in the storage area. The two words contain the starting byte for the substring and the number of bytes to be transferred.
- The array information for the array must be retrieved with the "Get_info" utility before calling this utility.
- The offset of the element in the array must be correct in the array information for the array (word 16 returned by "Get_info"). It should be remembered that the offset of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays. The offset is in terms of number of elements.[1]

Calling Procedure:

1. Store the number of the starting **byte** of the substring to be changed into the first word of the storage area. (Note that bytes 0 and 1 are the length word of the string.)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Store the element offset into the array information (word 16).
4. Load register A with the address of the storage area for the string to be transferred.
5. Load register B with the address of word 0 of the information returned by the "Get_info" utility (see description of that utility).
6. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

1 See the description of calculating array offsets under the "Get_info" utility.

For example —

```
ISOURCE Value:      DAT 2     ! 1st character (ignore length)
ISOURCE             DAT 10    ! Transfer 10 characters
ISOURCE             BSS 5     ! Substring storage area
ISOURCE Array_info:BSS 39
ISOURCE Element:    EQU Array_info+16 ! Element offset
ISOURCE             SUB
ISOURCE Parameter: STR (*)
                     .
                     .
                     .
ISOURCE             LDA =Array_info   ! Get array info
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_info
ISOURCE             LDA =2            ! Set offset to 2
ISOURCE             STA Element
                     .
                     .
                     .
ISOURCE             LDA =Value        ! Info already saved
ISOURCE             LDB =Array_info
ISOURCE             JSM Put_elem_bytes
```

In this example, Value is the storage area for the string to be transferred. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string element are changed. It is the responsibility of the software (not shown) to assure that 10 characters of valid data are stored in the remainder of the storage area.

Using Common

A faster way to pass information between BASIC and assembly language routines is through BASIC's common area.

You may recall from subprograms in BASIC that if you have a COM statement in the main program, the locations named therein can be accessed by other BASIC subprograms and functions through their own COM statements. Though the subprograms may change the names, the locations are the same. The order of appearance in a COM statement is all-important. If a main program has the statement —

```
COM A,B,C
```

and a subprogram has the statement —

```
COM X,Y,Z
```

then X and A are the same storage location, B and Y are the same, and C and Z are the same.

The same kind of operation is available in your assembly language routines with the COM pseudo-instruction —

```
COM
```

As with the SUB pseudo-instruction, the COM only serves as a preface. It is followed by one or more parameter declarations of the same types as in the SUB —

```
ANY
INT
REL
SHO
STR
```

The FIL is not permitted, since there is no corresponding item within BASIC's COM syntax.

Each pseudo-instruction used after an assembly language COM corresponds to an item in the COM declaration in the main BASIC program. Just as in a BASIC subprogram, the types must agree.[1] However, the ANY pseudo-instruction fulfills the same function here as it does with the SUB pseudo-instruction — to allow any type of item to be passed.

As with SUB, arrays are designated by following the type with an array identifier — `(*)`. If the type is ANY, the array identifier is not allowed.

Each pseudo-instruction reserves three words of memory when assembled. And, like SUB, the words are used to contain a descriptor. The descriptors are used by the variable retrieval utilities for fetching and storing values in the common area. The same utilities used in fetching and storing argument values are used for the same purposes for values in the common area. These utilities are —

Get_info
Get_value
Get_element
Get_bytes
Get_elem_bytes
Put_value
Put_element
Put_bytes
Put_elem_bytes

1 If the types do not correspond, an error results (number 198). This matching is checked only for the module containing the routine which was ICALLed.

The utilities are called in the same fashion and are subject to the same restrictions. See the description of the utilities in the preceding sections of this chapter to determine how they are used.

The item pseudo-instructions used with the COM pseudo-instruction can have their own labels, just as the parameter declarations used with a SUB may have. And just as in a BASIC subprogram, they need not have the same names as were given the corresponding items in BASIC. For example, suppose the following BASIC common statement exists at the time of a call to an assembly language routine —

```
COM Q(20),Z$[10]
```

then you could access Q(*) and Z$ by using these pseudo-instructions —

```
ISOURCE      COM
ISOURCE X:   REL (*)
ISOURCE Y:   STR
```

Note the differences in names.

If the number of item pseudo-instructions in the assembly language routine exceeds the number of items in common at the time the routine is called, an error results (number 199).

Similar to BASIC, a common declaration can contain more than one COM sequence. All the COM sequences are treated together as a single common area. For example —

```
BASIC:      COM REAL A1,B1,INTEGER,C1,D1
ASSEMBLY:   COM
            A1: REL
            B1: REL
                 .
                 .
                 .
            COM
            C1: INT
            D1: INT
```

NOTE

If a BASIC COM statement is changed, modules containing the COM pseudo-instruction should be re-IASSEMBLEd or re-ILOADed before executing an ICALL statement.

Busy Bits

Overlapped processing in the 9845 is partially implemented through the facility of "busy bits".

Each variable located in the BASIC value or common areas has associated with it two bits which are independent of the value — a "read" busy bit, and a "write" busy bit. Each time an I/O operation is executed that cannot be buffered, one of the busy bits is set. If a variable is having its value changed by the I/O operation, then the read busy bit is set. If the variable is outputting its value in the I/O operation, then its write busy bit is set. If a variable is not involved in a pending I/O operation both bits are cleared. When the I/O operation is completed, the busy bits for the variables involved are cleared.

When an I/O operation is encountered during execution of BASIC statements, the appropriate busy bits are set and a request is made by the operating system for the resources to satisfy the operation. Until that operation is complete, BASIC (in OVERLAP mode), continues to execute succeeding lines in the program until it encounters a statement which contains variables with busy bits that are set.

If the statement is attempting to use the value of a variable and its read busy bit is set, then the further execution of the statement waits until the busy bit is cleared. The same is true for a statement attempting to change the value of a variable when either its read or write busy bit is set. When the I/O operation completes, the busy bits are cleared and the waiting statement is executed.

In short, overlapped processing uses busy bits as a signal as to whether a statement can be executed or not.

If an ICALL statement is executed with overlapped processing, it is possible that a BASIC variable may be "busy" when the routine wants to access it. Although it is still possible to access the variable without regard to the status of the busy bits, frequently that is not a desirable programming approach. You may on occasion want to check the value of the busy bits when you suspect the user of the routine may be using overlapped processing.

Busy bits are checked from an assembly program using the "Busy" utility to be described shortly. If you are checking the bits for a busy condition, and the busy condition is set, it remains set throughout the time you are in the assembly routine. For it to become un-busy, you must give the operating system a chance to perform the I/O operation and clear the busy bits. One way to do this is to exit the ICALL and return to BASIC.

For example —

```
330  ICALL Sort(Busy)
340  IF Busy THEN 330
```

If the Sort routine exits, setting Busy to 0 if a busy condition is not encountered, and to non-zero otherwise, this keeps trying to execute Sort until the common variables which are busy become un-busy and it can proceed on its way. By exiting the routine after each unsuccessful attempt, the operating system is given an opportunity to perform the I/O operation which has the variable(s) tied up.

UTILITY: Busy

The Busy utility checks the status of the busy bits of a variable.

General Procedure: The utility is given the location of the declaration for the variable. It returns the value of the busy bits for that variable into the A register.

Special Requirements: This utility should be used for all variables involved in overlapped I/O operations.

Calling Procedure:

1. Load register B with the address of the pseudo-instruction of the declaration to be checked.
2. Call the utility.

Exit Conditions: The utility returns the busy bits in the A register. The "read" busy bit is in bit 0 and the "write" busy bit is in bit 1. The other bits are cleared.

In the following example, if any of the busy bits among three common variables is set, a flag is set and the routine is exited —

```
ISOURCE                COM
ISOURCE  Variable1:    INT
ISOURCE  Variable2:    SHO
ISOURCE  Variable3:    REL
            .
            .
            .
ISOURCE                SUB
ISOURCE  Busy_bits:    INT
ISOURCE  Sort:         LDB =Variable1
ISOURCE                JSM Busy
ISOURCE                RZA Is_busy
ISOURCE                LDB =Variable2
ISOURCE                JSM Busy
ISOURCE                RZA Is_busy
ISOURCE                LDB =Variable3
ISOURCE                SZA Go_ahead
ISOURCE Is_busy:       LDA ==1
ISOURCE                LDB =Busy_bits
ISOURCE                JSM Put_value
ISOURCE                RET 1
ISOURCE Go_ahead:        ! Continue processing.
ISOURCE Work:   ! Continue processing
```

The overhead of exiting and re-entering the ICALL statement while waiting for a variable to become unbusy can be avoided. It is sufficient to allow the operating system to perform an I/O operation without having to go back to BASIC. A special utility, To_system, is provided for this purpose.

UTILITY: To_system

The To_system utility gives the operating system a chance to move toward completion of any I/O operation which has not already completed.

General Procedure: Each call to the utility gives the operating system one chance to perform an I/O operation.

Calling Procedure: Call the utility.

Exit Conditions: The utility always returns the instruction following the JSM To_system instruction. There are no error exits from the utility.

In the following example, the Sort routine waits until all busy bits in the three common variables are cleared before proceeding with execution:

```
ISOURCE              COM
ISOURCE Var1:        INT               !
ISOURCE Var2:        SHO               ! Common declarations.
ISOURCE Var3:        REL               !
                      .
                      .
ISOURCE              SUB
ISOURCE Sort:        LDB =Var1         ! Check busy bits.
ISOURCE              JSM Busy
ISOURCE              RZA Is_busy
ISOURCE              LDB =Var2
ISOURCE              JSM Busy
ISOURCE              RZA Is_busy
ISOURCE              LDB =Var3
ISOURCE              JSM Busy
ISOURCE              SZA Go_ahead
ISOURCE Is_busy:     JSM To_system     ! Allow system to do some I/O.
ISOURCE              JMP Sort          ! Check busy bits again.
ISOURCE Go_ahead:     !                ! Continue processing.
```

Chapter 7
I/O Handling

Summary: This manual should be used in conjunction with "BASIC Language Interfacing Concepts" which covers the specifics of different interface cards. This chapter describes the various techniques of handling the receiving and sending of information to peripheral devices. Topics are: a review of I/O machine instructions, registers, applicable utilities, interrupts and interrupt service routines, handshake I/O, direct memory access, and mass storage devices.

A major usage for assembly language programs is to improve or customize the performance of the 9845 with respect to data transfers with peripheral devices. The types of devices dealt with are those which communicate via the various interface cards (e.g., HP 98032, HPIB, etc.). The types of I/O which the assembly language supports are **programmed** (handshake-type), **interrupt**, and **direct memory access** (or DMA).

A number of detailed examples have been provided demonstrating the various types of I/O using different interfaces. These examples can be found in Appendix H.

Peripheral-Processor Communication

All I/O, except for that to the internal devices (tape cartridges, keyboard, printer, CRT, or Graphics), necessarily takes place through the "backplane". The backplane is that physical area of the machine where the interface cards are inserted (also known as the I/O "slots").

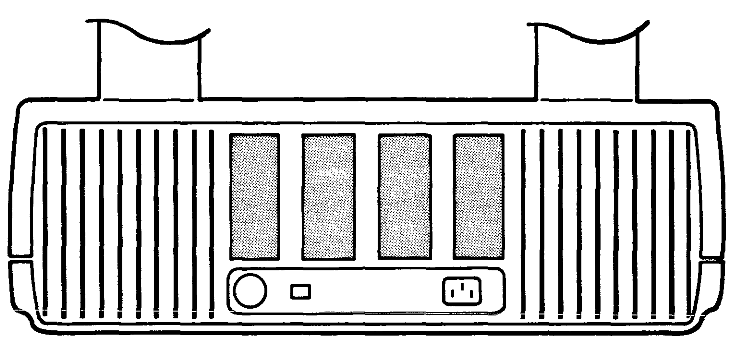


Figure 8. Location of I/O Slots (Backplane)

Interfaces

The processor does all its talking, through the backplane, to peripheral interfaces, never directly to a peripheral itself. An interface is a complex electronic circuit which provides mechanical, electrical, data format, and timing compatibility between the 9845 and the peripheral device to which it is connected. From a programmer's point of view, the primary task of an interface is to provide a means of exchanging data between the 9845 and the peripheral. An interface isolates the programmer from the details of electronics and timing, appearing as a simple "black box" through which information is exchanged.

The processor can talk to as many as 12 peripheral interfaces through the backplane. Each can be talked to individually, and there may be a mix of peripherals using programmed, interrupt, or DMA types of transfers.

Individual I/O operations (i.e., exchanges of single words) occur between the processor and one interface at a time, although interrupt and DMA modes of operation can be programmed to allow automatic interleaving of individual operations.

A peripheral is addressed through a select code and a transfer occurs through four special registers reserved for the purpose. These will each be discussed shortly.

Discussion of the techniques and methods presented in this chapter uses the common HP interfaces as examples. A full discussion of the operation of these interfaces can be found in the BASIC Language Interfacing Concepts manual (HP part number 09835-90600) and also from your Sales and Service office.

Example programs utilizing various I/O techniques with a number of the standard interfaces can be found in Appendix H.

Registers

All I/O operations go through a set of four registers maintained by the 9845. The four registers named R4, R5, R6, and R7 are the sole means of communicating data between the processor and peripheral interfaces. While the registers are actually on the interface cards, they may be thought of as being in the computer memory. This makes the cards themselves accessible by simple memory referencing instructions.

The 9845 sees the registers as single-words and always sends or receives a full word of data when it references one of them. If a particular interface utilizes less than the full sixteen bits (when exchanging 8-bit extended ASCII data bytes, for example), then the most significant bits (8 through 15) are received as zeroes. On output, if fewer than 16 bits are utilized by the interface, it ignores the most significant bits. The value of these bits, in this case, is a "don't care" (i.e., may be any pattern of ones or zeroes).

All of the HP 9803X series of interface cards use the registers as follows —

Register	On Input	On Output
R4	Primary Data In	Primary Data Out
R5	Primary Status In	Primary Control Out
R6	Secondary Data In	Secondary Data Out
R7	Secondary Status In	Secondary Control Out

The R4 register, then, is almost always used for data transfers. R5 is always used for status and control information. The "secondary" registers — R6 and R7 — perform the indicated functions only nominally. The exact interpretation as to how the register is used depends upon the interface card being used (see the BASIC Language Interfacing Concepts manual for details).

In order to give some specific examples for using the registers, the 98032 16-Bit Parallel Interface (sometimes called General Purpose Input/Output — GPIO) is used. This card defines the secondary registers as —

Register	On Input	On Output
R4[1]	Low-Byte Data In	High Bype Data Out
R5	Status In	Control Out
R6[1]	High-Byte Data In	High-Byte Data Out
R7	(unused)	Trigger

Select Codes

As mentioned earlier, more than one interface card may be connected to the 9845. It becomes necessary, then, that there be a mechanism whereby a particular interface can be chosen to respond when an I/O register is referenced for either input or output. This mechanism is the Peripheral Address Register (Pa).

Pa holds a binary number in the range 0 to 15 (utilizing only the lower four bits of the word, 0 to 3). Each interface has an externally-settable select code switch which can also be set to a value between 0 and 15. However, since select codes 0, 13, 14 and 15 are reserved for the internal printer, Graphics and tape cartridge units, respectively, the permissible select code settings are 1 through 12.

Whenever an operation to one of the I/O registers is performed, the System 45 makes the contents of the Pa register available to all the interfaces connected to the backplane. Each card compares the value with its own select code. If they match, the interface responds to the operation.

1 These registers contain the same data if the 98032 card is not jumpered for byte mode. See BASIC Language Interfacing Concepts.

So, for example, if the following statements are executed in turn —

```
ISOURCE   LDA =8    ! Choose peripheral on select code 8
ISOURCE   STA Pa
ISOURCE   LDA R4    ! Read from the interface
```

then a status byte is read from the interface card set to select code 8.

The label "Pa" is reserved by the assembler for the Peripheral Address register.

Status and Control Registers

The primary purpose of any interface is to allow data to be exchanged between the computer and the peripheral device to which it is connected. But HP's 9803X series of interface cards are even more versatile, possessing a programmable capability of their own. This in turn provides optional capabilities with the card that can be set and changed by control instructions from the System 45. (For details on what capabilities are provided, consult the BASIC Language Interfacing Concepts manual.)

The programming of the interface is done by the 9845 using the R5 register. Some of the interfaces use other registers for extended control bits (these are also described in the BASIC Language Interfacing Concepts manual).

Interface cards can also return information to the 9845 about which optional programming features are currently selected. This information, called the status byte, is obtained through an input operation using register R5. The status byte (8 bits) is determined solely by the characteristics of the interface card being addressed in the Pa register. (Again, information on particular cards can be found in BASIC Language Interfacing Concepts).

Remembering that these registers are not really memory locations, but instead are registers on the card being addressed by the Pa register, storing information to these locations is not the same as storing to other memory locations or registers. For example, storing a value in R5 to set the control register sends the information to the addressed interface. Later, if you were to read a value from R5, the information you sent would not be what is returned. Instead, the contents of the status register in the interface would be returned.

Status and Flag Lines

Whenever an I/O register is accessed, the interface with the same select code as is in the Pa register responds. The primary response depends upon the nature of the interface and which register is accessed (see discussion above). However, in all cases there is a secondary effect. Part of every interface's response is to set or clear the Status and Flag lines.

The **Status** line (not to be confused with the status register discussed above), is a single bit indicating whether the interface is operational or not. By inclusion, this can also mean the status of the actual peripheral to which the interface is connected. For example, if a peripheral device has a line coming from it that indicates its power is on, it could be connected to the Status line in the interface. Then the program could quickly determine whether the device is turned on or off. As another example, a printer might have the Status line connected to the out-of-paper indicator (should it have one) to indicate to the program when it is inoperable because of lack of paper.

The **Flag** line is a momentary "busy/ready" indicator used to keep the computer from getting ahead of the peripheral. The line shows that the interface is busy processing the last task given it by the 9845 or that it is ready for another operation. If the line is set, it indicates "ready"; if the line is cleared, it indicates "busy". For example, if the computer has a sequence of ASCII characters to send to a slow printer, it sends one character (making the Flag line "busy") and then waits for the Flag line to go "ready" again before sending the next character.

There are four instructions, part of the I/O group, which can check these lines —

SFS Skip if Flag line is set (i.e., "ready")
SFC Skip if Flag line is cleared (i.e., "busy")
SSS Skip if Status is set (i.e., "operational")
SSC Skip if Status is cleared (i.e., "non-operational")

These instructions have the capability of skipping up to 31 locations in a forward branch, up to 32 locations in a backward branch, or to the same instruction.

Programmed I/O

Programmed I/O is the process whereby software controls the transfer of information between memory and an interface. In the process the program must decide when and where to make the transfer, how to make it, and how much information to transfer. The decision even to originate the transfer comes under program control.

The Status line can be used to determine the availability of an interface. The interface is selected, under program control, by the contents of the Pa register. Then the Status line is checked to see if the interface (and by inclusion its associated peripheral) is operational.

After an operational interface has been chosen, the Flag line can be used to determine when the interface (i.e., peripheral) is ready for a transfer and when it has not finished with the previous transfer.

With sufficient checks of Flag and Status before and between I/O operations, it is possible to eliminate initiating an I/O operation to an interface which isn't ready for it. The following example checks the status (status bit set) of an interface card:

```
ISOURCE                STA Pa          ! Choose the peripheral.
ISOURCE                SSS Status_ok   ! Check for operational device.
ISOURCE                LDA =164        ! Not operational, error 164.
ISOURCE                JSM Error_exit  ! Inform user.
ISOURCE Status_ok:     SFC *           ! Wait until card is ready.
ISOURCE                 .              !
ISOURCE                 .              ! I/O operation done here.
ISOURCE                 .              !
```

The instruction sequence for a software controlled output transfer differs slightly from that of an input transfer. An output transfer involves waiting for the interface flag, outputting the data and then starting the output handshake. The following is an illustration of this sequence. The essential instructions are preceded by an asterisk in the comments column.

```
ISOURCE                 !
ISOURCE                LDA Select_code      ! Grab select code.
ISOURCE                STA Pa               ! Put it in Pa.
ISOURCE Again:         SSS *+3              ! Check device status.
ISOURCE                LDA =164             ! Flag error,
ISOURCE                JSM Error_exit       ! if device down.
ISOURCE                LDA Buffer_pointer,I ! Grab word from buffer.
ISOURCE                ISZ Buffer_pointer   ! Increment pointer.
ISOURCE                SFC *                ! * Wait for flag set.
ISOURCE                STA R4               ! * Output the word.
ISOURCE                STA R7               ! * Trigger the handshake.
ISOURCE                JMP Again            ! Do it again.
ISOURCE                 !
```

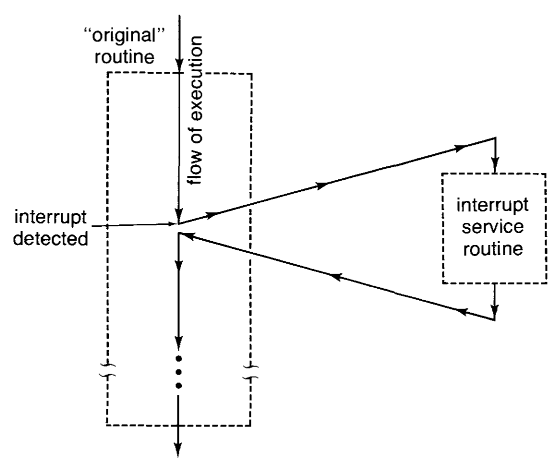
An input transfer involves signalling an input operation, triggering the input handshake, waiting for the interface flag and then inputting the data. This sequence is illustrated here with the essential instructions preceded by asterisks in the comments column.

```
ISOURCE              !
ISOURCE              LDA Select_code     ! Grab select code.
ISOURCE              STA Pa              ! Put it in Pa.
ISOURCE Again:       SSS *+3             ! Check device status.
ISOURCE              LDA =164            ! Flag error,
ISOURCE              JSM Error_exit      ! if device down.
ISOURCE              SFC *               ! * Wait for flag.
ISOURCE              LDA R4              ! * Signal input operation.
ISOURCE              STA R7              ! * Trigger input handshake.
ISOURCE              SFC *               ! * Wait for flag.
ISOURCE              LDA R4              ! * Grab input data.
ISOURCE              JMP Again           ! Do it again.
ISOURCE              !
```

Interrupt I/O

Interrupt I/O is a means of allowing control to pass temporarily to an assembly language routine other than the routine (BASIC or assembly language) currently executing. The "interrupt", which causes the control to be passed, is detected through the backplane and is associated with a particular interface. After the "interrupt service" routine completes its tasks, control is passed back to the original routine.

The process looks something like this —



The sequence of events in interrupt I/O can be detailed as follows —

1. The interface sends a request for service to the backplane which passes it along to the processor. Conditions which generate this request for service are different for each I/O card. See BASIC Language Interfacing Concepts.
2. The processor alters the flow of execution so that the routine associated with that interrupting source can be executed. The processor saves its place in the interrupted routine so that it can later return to it. The current contents of the Pa register are saved internally in the processor and the Pa is then set to the select code of the device causing the interrupt.
3. The interrupt service routine is executed, performing whatever functions are desired. Frequently these functions involve some form of programmed I/O or direct memory access. The service routine may signal an end-of-line BASIC branch, indicating to BASIC that some condition occurred (discussed below).
4. The service routine returns the processor to the interrupted routine so that the "original" process can resume.

The uses for interrupt I/O are so diverse that it is difficult to generalize about them. However, one particular use is fairly well-defined and of general applicability — data transfers.

Interrupt I/O is normally used in data transfers whenever a particular data device has a transfer rate which is significantly slower than that of the computer. Peripheral devices with transfer rates less than 7000 characters per second are candidates for interrupt I/O.

The usual approach is to transfer a word to or from the peripheral device, then go away to do some other processing while waiting for the device to interrupt by becoming "ready" for another transfer. An example illustrating the general procedure for an interrupt I/O transfer is presented following some more background information concerning priorities, ISR linkage, access, preservation and indirect addressing.

Priorities

Select codes are assigned hardware "priority" levels to control what should be processed when an interrupt service routine is executing and another interrupt is received, or when two or more simultaneous interrupts are received.

There are two priority levels —

High	for select codes 8 to 15
Low	for select codes 0 to 7

An interrupt received from a high-priority select code may interrupt a service routine which is executing for an interrupt from a low-priority select code. But an interrupt from a low-priority select code may not interrupt any other service routine.

Interrupt Service Routines and Linkage

An interrupt service routine is associated, or "linked", with a select code by the Isr_access utility described later. This linkage establishes where the interrupt service routine resides, and to which select code it applies. An interrupt service routine typically does one or more of the following —

- Talks to the interface (i.e., satisfies or acknowledges the interface's interrupt).
- Passes data to (or retrieves data from) the rest of the program, when appropriate.
- Breaks the linkage, if desired.

The method of talking to the interface depends upon the type of interface. Some devices or applications do not require the passage of data; the acknowledgement of the interrupt is usually the desired effect in such cases.

Interrupt service routines are always exited with a RET 1 instruction.

Breaking Interrupt Service Routine Linkage

The interrupt service routine-select code linkage can be broken from within the interrupt service routine by executing one of two statements. If the linked select code is high priority, the statement is —

```
JSM End_isr_high,I
```

If the linked select code is low priority, the statement is —

```
JSM End_isr_low,I
```

After execution of one of these linkage-breaking statements, the interrupt service routine is exited with a RET 1 instruction.

Several important facts to keep in mind concerning the JSM End_isr_low,I and JSM End_isr_high,I statements are the following:

- The names, End_isr_low and End_isr_high, do **not** represent utilities or routines. Therefore, they should not be declared as externals.
- Neither statement may appear outside of the appropriate interrupt service routine.
- These linkage-breaking statements should only be executed inside the appropriate interrupt service routine when you no longer need select code linkage to the ISR. In most cases, this is when the ISR is no longer needed because the data transfer is complete.
- The contents of the Pa register are used by End_isr_high and End_isr_low to determine what resources to free and what interrupt linkages to break. Upon entry to the ISR the Pa register contains the select code of the interrupting interface, but you can change Pa during execution of the ISR. If this is done, you must ensure that Pa is set to the desired value before calling End_isr_high or End_isr_low.

Here is an example of a short interrupt service routine which simply reads and processes words from the interface and terminates when it encounters a linefeed.

```
ISOURCE Lf:         EQU 10
                     .
                     .
                     .
ISOURCE Isr:        LDA R4     ! Retrieve character from interface.
ISOURCE             CPA =Lf    ! Is it a line feed?
ISOURCE             JMP Terminate  ! Yes; go to terminate routine.
                     .         ! If not,
                     .         ! process the
                     .         ! character.
ISOURCE             STA R7     ! Trigger another handshake.
ISOURCE             RET 1      ! Return to background program.
                     .
ISOURCE Terminate:  JSM End_isr_low,I  ! Break ISR linkage.
ISOURCE             RET 1      ! Return to BASIC.
```

NOTE

Utilities cannot be called from an interrupt service routine. Attempts to do so lock up the machine.

Access

The operating system (OS) contains a mechanism to regulate requests for hardware capabilities in order to eliminate conflicting uses of these capabilities. For instance, since there is only one DMA[1] channel, it is necessary that there be a mechanism to prohibit two simultaneous DMA transfers.

1 DMA (Direct Memory Access) is explained further in later sections of this chapter.

The OS mechanism which regulates the use of DMA (and also interrupt) transfers either grants or does not grant what is called "access". Before starting either an interrupt or DMA operation, access should be requested from the operating system.

Another example — suppose a device operating on a high priority select code has a relatively slow data rate. This is an ideal situation in which to use interrupt driven I/O. Suppose further that the device operates in such a fashion that the data must be transferred within a fixed time period following its issuance of an interrupt or the data is lost (the internal tape drive is such a device.) If there are other interrupt type transfers operating concurrently on other high priority select codes, it may not be possible to service our slow device within the necessary time frame. When the operating system grants access, this type of conflict is impossible.

Users of the assembly language system are required to request access from the operating system. The OS grants access if granting this access does not compromise any previously granted access.

Devices such as that discussed above which require interrupt service within a specified time frame are called "synchronous", and should use "synchronous" access. Devices with no such time constraints are called "asynchronous", and should use "asynchronous" access.

Abortive access is intended to be used by routines that will be executed only extremely infrequently. For instance, if the System 45 is monitoring a potentially dangerous manufacturing process, it may be necessary to have an interrupt service routine to shut down the process when something goes awry. This could be accomplished with an abortive routine. The advantages of access code 0 (abortive access) is that no other modes of access are prohibited by its use. Thus, the infrequently used routine will not prevent another routine from getting the type of access it needs.

Access code 0 should be used with caution. An interrupt routine with abortive access can exist on the same priority level as an interrupt routine with synchronous access. If the abortive routine is in progress when an interrupt occurs requiring the synchronous routine, the abortive routine will finish before the synchronous routine can be serviced. The timing requirements of the sychronous routine might thus be violated.

Access code 0 is also used to release access in a particular type of DMA transfer to be explained later in this chapter.

The regulation of access incorporates the following points —

- When the operating system grants synchronous access to an operation, it is guaranteeing that the requesting process will have its interrupts serviced with maximum priority.
- DMA conflicts with synchronous access since DMA's cycle stealing causes the processor to run slower and could thus compromise a synchronous process.
- Synchronous access on a low priority select code conflicts with asynchronous access on a high priority select code since the asynchronous device could interrupt the synchronous ISR, thus compromising the timing requirements of the synchronous device. Synchronous access conflicts with asynchronous access on the same priority level. Remember an interrupt request on the same priority level as a currently executing ISR will not be processed until the executing ISR completes.

The following table summarizes the granting of access —

Access Already Granted

Access Requested		Abortive L	Abortive H	ASYN L	ASYN H	DMA	SYN H	SYN L
Abortive	Low	y	y	y	y	y	y	d
Abortive	High	y	y	y	y	y	d	d
ASYN	Low	y	y	y	y	y	y	n
ASYN	High	y	y	y	y	y	n	n
DMA		y	y	y	y	n	n	n
SYN	High	y	d	y	n	n	n	n
SYN	Low	d	d	n	n	n	n	n

n = Not granted
d = Dangerous, but granted
y = Granted

BASIC statements also obtain and release access as I/O is performed. The following table lists some of the ways access is used by the system —

Use	Access
Cartridge Operations	SYNC (HIGH select code)
Flexible Disk Operations	DMA
PRINT, PRINT USING	ASYNC
Plotter Drivers	ASYNC
CARD ENABLE	ASYNC
ENTER/OUTPUT INT	ASYNC
ENTER/OUTPUT DMA	DMA
ENTER/OUTPUT FHS[1]	DMA

In general, single BASIC statements could cause access to be granted and released several times. For example, the cartridge operations obtain and release synchronous access once for each physical record transferred.

It is imperative that access be released after an interrupt service routine has been executed for the last time or a DMA transfer is complete. Such occurrences as tape drive lockout, can occur if access is not released. Use the JSM End_isr_high,I or JSM End_isr_low,I instructions to free access, depending on the select code used.

UTILITY: Isr_access

This utility is used to request access and, if the access is granted, to create the linkage between an interrupt service routine (ISR) and a select code. Valid select codes are 1 through 13. Pressing RESET (CONTROL STOP) during execution of the utility may cause a SCRATCH A to be issued.

General Procedure: The utility is told where the ISR resides and what kind of access is required. If access is granted, it returns successfully. If access is not granted immediately, it keeps trying periodically until it is successful or until a specified number of attempts have been made (in which case it returns unsuccessfully).

1 In addition to obtaining DMA access (which in this case is used just to ensure there is no synchronous access granted), the FHS (Fast Handshake) drivers disable all interrupts during the actual transfer loop.

Special Requirements: The B register must contain information as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits	Description
0–3	Select code to be linked to the ISR
4–5	Access code
8–14	Number of attempts to be made before aborting

The access codes are —

0	Abortive access
1	Asynchronous access
2	DMA with asynchronous access
3	Synchronous access

Calling Procedure:

1. Load register A with the address of the ISR.
2. Load register B with the information described above.
3. Call the utility.

Exit Conditions:

RET 2 If the attempt at linkage is successful, the utility returns to the second word following its call. Register Pa is set to the select code; if access code 2 was specified then Dmapa has also been set to the select code.

RET 1 If the attempt at linkage is unsuccessful, the utility returns to the first word following the call. Register A contains an indication of the type of difficulty encountered —

– 1 Access couldn't be obtained after specified number of attempts.

– 2 Select code is still linked to an assembly language ISR.

As an example of the use of the Isr_access utility, suppose an ISR is to be linked to select code 2 for asynchronous access. The following would be a sequence to establish such a linkage —

```
ISOURCE         EXT Isr_access
                 .
                 .
                 .
ISOURCE         LDA =Read
ISOURCE         LDB =(64*256)+(1*16)+2   ! 64 trials, asynch, SC 2
ISOURCE         JSM Isr_access
ISOURCE         JMP Error
                 .
                 .
                 .
ISOURCE Error:  ISZ A
ISOURCE         JMP Nested_isr     ! Handler for SC busy
ISOURCE         JMP No_resources   ! Handler for time-out
                 .
                 .
                 .
```

NOTE

Access must be released after the execution of an interrupt or DMA transfer is complete with a JSM End_isr_high,I or a JSM End_isr_low,I instruction, depending on the select code used.

Disabling Interrupts

At times it is necessary to disable all interrupts in order to execute a particular sequence of instructions. This is typically necessary for one of two reasons:

- The instructions are modifying some data used by an ISR, and the ISR would become confused if it happened to occur when this data was in a transitory state.
- All ISRs are prohibited in order to minimize the execution time for some task (i.e. fast handshake transfers).

In general, it is allowable to disable interrupts (using the DIR instruction) for up to 100 μs without "notifying" the operating system. (Interrupts are re-enabled using the EIR instruction.) Attempts to disable for more than 100μs without this notice could compromise any synchronous transfers that may be in progress. Specifically, it could cause loss of data if a tape operation were in progress.

If is necessary to disable interrupts for more than 100μs, the Isr_access utility should be used to acquire an access which ensures that no synchronous transfers are jeopardized. Typical access code requests to do this are DMA, ASYNC HIGH, or SYNC (high or low). The one to choose depends on the application.

For example, suppose you would like to minimize the execution time for a segment of code. The segment takes longer than 100μs, but you need to disable interrupts for the duration. The ideal access to request may be DMA. Once DMA has been granted there can be no DMA transfers (which might slow the processor) and there can be no synchronous transfers in progress. Therefore, interrupts can be disabled for as long as necessary.

When Isr_access is used for this purpose (i.e. to get access rather than to set-up ISR linkages), the entry and exit conditions are as previously described except that the A register must contain a zero.

When access is obtained in this manner, it is freed by calling Isr_access a second time with the A register containing a zero. However, the access code requested in bits 4 and 5 of the B register must be zero. This technique of freeing access can be used only if the original access was granted without interrupt linkage (i.e. the A register was 0). Attempts to do otherwise cause Isr_access to give the fail return (RET 1).

The following example illustrates the technique for a fast handshake transfer to a 98032 interface on select code 6.

```
                         .
                         .
                         .
ISOURCE Keep_trying:LDA =0              ! Get DMA access without
ISOURCE             LDB =(127*256)+(2*16)+6! interrupt linkage.
ISOURCE             JSM Isr_access
ISOURCE             JMP Keep_trying
ISOURCE             DIR                 ! Now disable interrupts.
ISOURCE             LDA =Buffer
ISOURCE             STA C
ISOURCE             LDA =-Count+1
ISOURCE             SFC *                 !
ISOURCE             WWC R4,I              ! Fast handshake loop.
ISOURCE             STA R7                !
ISOURCE             RIA *-3               !
ISOURCE             EIR                 ! Re-enable interrupts.
ISOURCE             LDA =0              ! Release access.
ISOURCE             LDB =(127*256)+6    !
ISOURCE             JSM Isr_access
ISOURCE             JMP Error_ret_1     ! Should never get RET 1.
                         .
                         .
                         .
```

State Preservation and Restoration

When an interrupt is detected and an interrupt service routine is called, the operating system automatically saves the state of some of the registers so that their values can be restored upon return from the ISR. Other registers are left alone and if your service routine uses them, it is up to your ISR to save them and restore them before returning from the ISR.

The registers which are automatically preserved are —

A
B
C
Cb
P
Pa

Also, the state of the Overflow and Extend processor flags are preserved and restored before the return from the interrupt.

The D and Db registers are not automatically saved. Saving and restoring location Db is not trivial due to the fact that this location is a read-only location. The following program segment saves and restores D and Db:

```
ISOURCE D_save_low: BSS 1
ISOURCE Db_save_low:BSS 1
                    .
                    .
ISOURCE Save_d:     LDA D            !
ISOURCE             STA D_save_low   ! Save D and Db.
ISOURCE             LDA Db           !
ISOURCE             STA Db_save_low  !
                    .
                    .
ISOURCE Restore_d:  DBL              ! Clear Db.
ISOURCE             LDA Db_save_low  ! Get Db_save_low.
ISOURCE             SAL 1            ! Test most
ISOURCE             SAP *+2          ! significant bit.
ISOURCE             DBU              ! If minus, set Db.
ISOURCE             LDA D_save_low   !
ISOURCE             STA D            ! Restore D.
                    .
                    .
```

If your ISR contains any of the following types of instructions —

Indirect addressing
Stack group
CLR
XFR

and the operand of the instruction(s) is an address in the ICOM region, then it is necessary that the following instruction sequence be executed in the ISR before any such instruction is executed —

```
Save35_low:  BSS 1
              .
              .
              .
             LDA 35B
             STA Save35_low
             LDA 34B
             STA 35B
              .
              .
              .
```

Then, before the ISR exits, and after the affected instructions have been executed, the following sequence must be executed —

```
              .
              .
              .
             LDA Save35_low
             STA 35B
              .
              .
              .
```

Indirect Addressing in ISRs

Indirect addressing in ISRs can produce anomalies unless the following rules are followed —

1. If indirect addressing is employed with the operand being an address in the ICOM region, one of the processor registers must be preserved. For the method of doing this, consult the "State Preservation and Restoration" section immediately above.
2. If indirect addressing is used in a JMP or JSM (including any jumps to external symbols or symbols more than 512 words away from the current instruction, both of which have implied indirect addressing), then the most significant bit must be set in the address. For example, instead of —

```
EXT Sub
 .
 .
 .
JSM Sub
```

in an ISR the procedure must be —

```
EXT Sub
 .
 .
 .
JSM (=Sub+100000B),I
```

The assembler can generate an indirect instruction when you have not specified a ,I after the instruction. These indirect instructions lock the machine if executed within an ISR, and therefore must be re-written. IOF (indirect off) and ION (indirect on) are used to find those instructions for which the indirection is done automatically by the assembly. At the beginning of ISR use the IOF instruction. At the end of ISR use the ION instruction to restore the assembler to its normal state. Between an IOF/ION pair, any instruction for which the assembler would have generated an automatic indirect, a range error (RN) is generated.

Enabling the Interface Card

The particular interface card that you are using must be enabled for interrupts. The 98032 Interface card is used for illustration purposes. Setting bit 7 of the R5 OUT register enables this particular card for interrupts. The R5 OUT register is represented here —

98032 — R5 Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ENABLE INT	ENABLE DMA	RESET	ENABLE AUTO HAND-SHAKE	X	X	CTL1	CTL0

Bit 7: Logical 1 enables card to interrupt
Bit 6: Logical 1 enables DMA
Bit 5: Logical 1 resets interface card
Bit 4: Logical 1 enables auto handshake
Bit 3: (Don't card)
Bit 2: (Don't card)
Bit 1: Optional peripheral control bit 1
Bit 0: Optional peripheral control bit 0

Control bits 0 and 1 are used to drive interface lines CTL0 and CTL1, respectively.CTL0 and CTL1 are optional peripheral control lines.

(Representations of the I/O registers for each interface are provided in the Assembly Language Quick Reference manual.)

The 98032 card is enabled for interrupts with the instructions —

```
ISOURCE  LDA = 200B
ISOURCE  STA R5
```

and disabled with —

```
ISOURCE  LDA = 0
ISOURCE  STA R5
```

The interface card is typically enabled for the first data transfer, disabled at the beginning of the ISR and re-enabled before the ISR is exited.

Interrupt Transfer Example

An example of setting up an interrupt service routine for inputting character data is given in the example below. This example should bring together the information presented in the previous five sections of this manual. Note the procedures for requesting and giving up access, enabling and disabling the interface card for interrupts and processor register preservation and restoration because of indirect addressing in the ISR.

```
10    ! This program illustrates an interrupt transfer.
20                        !
30      IDELETE ALL        ! Clear ICOM region.
40      ICOM 200           ! Reserve 200 words in ICOM region.
50      IASSEMBLE ALL      ! Assemble.
60      ICALL Set_up       ! Call the set up routine.
70                        !
80    !
90    ! BASIC background routine.
100   !
110   END
120                       !
130     ISOURCE              NAM Interrupt        ! Module name.
140     ISOURCE              EXT Isr_access       ! Declare externals.
150     ISOURCE Select_code:EQU 2                 ! Select code is 2.
160     ISOURCE Lf:          EQU 10               ! ASCII for line feed.
170     ISOURCE Buffer:      BSS 81               ! Character buffer.
180     ISOURCE Buf_last:    EQU *-1              ! End of buffer.
190     ISOURCE Buf_point:   DAT Buffer           ! Buffer pointer.
200     ISOURCE Save35:      BSS 1
210     ISOURCE              SUB
220     ISOURCE Set_up:      LDA =Select_code     ! Routine entry point.
230     ISOURCE              STA Pa               ! Put select code in Pa.
240     ISOURCE              LDA =Isr             ! Get asynchronous access,
250     ISOURCE              LDB =(64*256)+(1*16)+Select_code
260     ISOURCE              JSM Isr_access       ! with Isr_access.
270     ISOURCE              JMP Set_up           ! Try again.
280     ISOURCE              SFC *                ! Wait for flag.
290     ISOURCE              LDA R4
```

```
300     ISOURCE             STA R7            ! Trigger the input.
310     ISOURCE             LDA =200B         ! Send interrupt enable
320     ISOURCE             STA R5            ! mask to R5.
330     ISOURCE             RET 1             ! Return to BASIC.
340     ISOURCE              !
350     ISOURCE Isr:        LDA =0            ! Interrupt service routine.
360     ISOURCE             STA R5            ! Disable interupts.
370     ISOURCE             LDA R4            ! Input the character.
380     ISOURCE             CPA =Lf           ! Is it a line feed?
390     ISOURCE             JMP Terminate     ! Yes; go to Terminate.
400     ISOURCE             LDB 35B           ! No; save processor register
410     ISOURCE             STB Save35        ! because on indirection.
420     ISOURCE             LDB 34B           ! Magic code.
430     ISOURCE             STB 35B           !
440     ISOURCE             STA Buf_point,I   ! Put character in buffer.
450     ISOURCE             LDB Save35
460     ISOURCE             STB 35B           ! Restore processor register.
470     ISOURCE             LDA Buf_point     ! Get buffer pointer.
480     ISOURCE             ADA =1            ! Increment it.
490     ISOURCE             CPA =Buf_last     ! End of the buffer?
500     ISOURCE             JMP Terminate     ! Yes; got to terminate.
510     ISOURCE             STA Buf_point     ! No; update pointer.
520     ISOURCE             SFC *             ! Wait for flag.
530     ISOURCE             LDA R4
540     ISOURCE             STA R7            ! Trigger next input.
550     ISOURCE             LDA =200B         ! Card enable mask
560     ISOURCE             STA R5            ! enables interrupts.
570     ISOURCE             RET 1             ! Retrun to BASIC.
580     ISOURCE Terminate:  LDA =0            ! Access freeing routine.
590     ISOURCE             STA R5            ! Disable interrupts.
600     ISOURCE             JSM End_isr_low,I ! After last transfer, give
610     ISOURCE             RET 1             ! up access and return.
620     ISOURCE              !
630     ISOURCE             END Interrupt
```

Direct Memory Access (DMA)

Direct memory access (DMA) is a means to exchange entire blocks of data between memory and peripherals. A block is a series of consecutive memory locations. Once started, the process is automatic; it is done under processor control, regulated by the interface. Since only the 98032 Interface supports DMA, the following discussion is in terms of that interface.

To the peripheral, the DMA operation appears as programmed I/O. The transfer, however, is actually performed by special DMA hardware. Information regarding the transfer is stored in the DMA registers for the DMA hardware to use. This information is the select code, the initial memory location, and the number of words to be transferred. The memory location register and the count register are successively adjusted after each word transferred until the transfer is complete. Upon completion of the transfer, the interface and the DMA hardware stop automatically.

The direction of the transfer is specified before the transfer takes place. It can be specified as either "inward" (i.e., from the peripheral to memory), or "outward" (i.e., from the memory to the peripheral). To set the direction outwards, the instruction —

```
SDO
```

is used. To set the direction inwards, the instruction —

```
SDI
```

is used.

DMA Registers

There are three registers which contain information used by the DMA hardware — Dmapa, Dmama, and Dmac. Before any DMA transfer takes place, the appropriate values must be loaded into these registers.

Dmapa contains the peripheral address of the device requesting DMA. Only the least significant bits of the register specify the select code which is to be the peripheral side of the DMA activity. During DMA transfers, the address bus takes its address from the Dmapa register rather than Pa as in other I/O transfers. The value is supplied to Dmapa by the Isr_access utility when it grants DMA access.

Dmama contains the address of the first word in memory (i.e., lowest address) where the data transferred is (or will be) stored. After each word transferred, this register is automatically incremented. Note that the entire block to be transferred must reside within the ICOM region.

Dmac is the count register for a DMA transfer. Before the transfer begins, it should be set to n-1, where n is the number of words to be transferred. After each word transfer, the count is decremented. If, during a word transfer, the value of Dmac is 0 (meaning that this is the last word to be transferred), the processor automatically informs the interface that the DMA operation will be complete after the present word is transferred.

DMA Transfers

There are two techinques for using DMA. Both initiate the DMA transfer in a similar manner but differ in how the end of the transfer is detected. The more commonly used method uses an interrupt generated by the interface. The second method uses a programmed test.

DMA transfers using interrupt are initiated with a sequence of six distinct actions.

Step 1: The Isr_access utility is used to obtain access to the DMA channel and to set up the ISR linkage used when the transfer terminates.

Step 2: The direction is set for input using an SDI instruction or for output using an SDO instruction.

Step 3: The appropriate values are stored into the Dmama and Dmac registers. (Dmapa is set by the Isr_access.)

Step 4: For input, the first handshake is initiated with these instructions:

```
SFC *
LDA R4
STA R7
```

For output, this step is deleted.

Step 5: The interface is enabled for DMA and interrupt by setting bits 4, 6, and 7 of R5 OUT to one. (i.e. 320B→R5)

Step 6: The DMA requests are enabled using the instruction DMA.

At this point you can do other processing if desired since data is being transferred automatically by the hardware. When all words have been transferred the interface interrupts the processor, causing the previously linked ISR to be executed. This ISR should:

- Disable the interface (bits 4, 6, and 7 of R5 OUT set to 0).
- Free the DMA acces by using End_isr_high or End_isr_low.

The following is a program segment to input 1024 words of data into an internal buffer area using interrupt to terminate the transfer.

```
10      ISOURCE Buffer:     BSS 1024
20      ISOURCE Sc:         EQU 2
30  !   ISOURCE              .
31  !   ISOURCE              .
32  !   ISOURCE              .
40      ISOURCE Linkage:    LDA =Isr          ! Step 1: Link ISR.
50      ISOURCE             LDB =(10*256)+(2*16)+Sc
60      ISOURCE             JSM Isr_access    ! Get DMA access.
70      ISOURCE             JMP Error_ret_1   ! Should never get here.
80      ISOURCE             SDI               ! Step 2: Set DMA inward.
90      ISOURCE             LDA =1023         ! Step 3: Load DMA registers.
100     ISOURCE             STA Dmac          ! Specify DMA count.
110     ISOURCE             LDA =Buffer
120     ISOURCE             STA Dmama         ! Specify buffer address.
130     ISOURCE             SFC *             ! Step 4: Wait for flag.
140     ISOURCE             LDA R4            ! Set up first transfer.
150     ISOURCE             STA R7            ! Trigger.
160     ISOURCE             LDA =320B         ! Step 5: Enable DMA.
170     ISOURCE             STA R5            ! and interrupt.
180     ISOURCE             DMA               ! Step 6: Notify DMA hardware.
190     ISOURCE             RET 1             ! Return.
200 !   ISOURCE              .
210 !   ISOURCE              .
220 !   ISOURCE              .
230     ISOURCE Isr:        LDA =0            ! Interrupt service routine.
240     ISOURCE             STA R5            ! Disable interrupts.
250     ISOURCE             DDR               ! Disable DMA.
260     ISOURCE             LDA Pa            ! Depending on select code,
270     ISOURCE             ADA =-8           ! terminate ISR linkage.
280     ISOURCE             SAP *+3
290     ISOURCE             JSM End_isr_low,I
300     ISOURCE             JMP *+2
310     ISOURCE             JSM End_isr_high,I
320     ISOURCE             RET 1             ! Return.
330 !   ISOURCE              .
340 !   ISOURCE              .
350 !   ISOURCE              .
```

In the previous example, the end of the DMA transfer is signaled by an interrupt which causes execution of an ISR. The ISR, in turn, gives up the DMA access and terminates the ISR linkage with End_isr_low,I or End_isr_high,I.

DMA transfers **without** interrupt are initiated with a sequence of six steps.

- Isr_access is used to obtain the DMA channel, but **not** to set-up an ISR (A register has a 0 value).
- The direction is set for input using an SDI instruction or for output using an SDO instruction.
- The appropriate values are stored in Dmac and Dmama.
- For input, the first handshake is initiated with these instructions:

```
SFC *
LDA R4
STA R7
```

- The interface is enabled for DMA by setting bits 4 and 6 of R5 OUT (i.e. 120B→R5).
- The DMA requests are enabled using the instruction DMA.

At this point you can do other processing if desired since data is being transferred automatically by the hardware. To determine if the transfer is complete, the Dmac register is tested. If it is negative, the transfer is complete and you should:

- Disable the interface (bits 4, 6 and 7 of R5 OUT set to 0).
- Free the DMA access by using Isr_access with the A register containing a 0 and an access code of 0.

The following is a program segment to output 1024 words of data from an interrupt buffer area without using interrupt to terminate the DMA.

```
10      ISOURCE Buffer:      BSS 1024
20      ISOURCE Sc:          EQU 2
30    ! ISOURCE               .
31    ! ISOURCE               .
32    ! ISOURCE               .
40      ISOURCE Dma:         LDA =Sc
50      ISOURCE              STA Pa
60      ISOURCE              SSS *+2          ! Check for operational card.
61      ISOURCE              JMP Card_down
62      ISOURCE Dma_no_int:  LDA =0           ! Step 1: No end-of-transfer
63      ISOURCE              LDB =(10*256)+(2*16)+Sc  ! interrupt.
64      ISOURCE              JSM Isr_access   ! Get DMA access.
70      ISOURCE              JMP Error_ret_1  ! Should never get here.
```

```
80        ISOURCE             SDO               ! Step 2: Set DMA outward.
90        ISOURCE             LDA =1023         ! Step 3: Load DMA registers.
100       ISOURCE             STA Dmac          ! Specify DMA count.
110       ISOURCE             LDA =Buffer
120       ISOURCE             STA Dmama         ! Specify buffer address.
130       ISOURCE             LDA =120B         ! Step 5: Enable DMA only.
140       ISOURCE             STA R5
150       ISOURCE             DMA               ! Step 6: Notify DMA hardware.
190       ISOURCE Check:      LDA Dmac          ! See if DMA is done.
200       ISOURCE             SAP *+2
210       ISOURCE             JSM Terminate
211 !     ISOURCE              .
212 !     ISOURCE              .                ! Other processing.
213 !     ISOURCE              .
230       ISOURCE             JMP Check         ! Go back and check DMA.
240       ISOURCE Terminate:  LDA =0
250       ISOURCE             STA R5            ! Clear R5.
251       ISOURCE             LDB =(64*256)+Sc  ! Ask for 0 access.
260       ISOURCE             JSM Isr_access
270       ISOURCE             JMP Error_ret_1   ! Should never get here.
271       ISOURCE             RET 1             ! Return.
272       ISOURCE Card_down:  LDA =167
273       ISOURCE             JSM Error_exit    ! Give error message.
350       ISOURCE             RET 1
351 !     ISOURCE              .
352 !     ISOURCE              .
353 !     ISOURCE              .
```

BASIC Branching on Interrupts

The handling of interrupts can be integrated into BASIC programs by using the ON INT statement. The object is to allow the flexibility of combining the high-level features of BASIC with the capabilities of assembly language in asynchronous I/O applications. And since ISRs cannot use the system utilities, in particular those that access a BASIC variable, a means of taking action on an interrupt after completion of the ISR is a necessity.

ON INT Statement

The ON INT statement is an executable BASIC statement which acts in a similar fashion to the ON KEY statement (see the System 45 Operating and Programming Manual). The statement allows the BASIC programmer to specify where, in his BASIC program, to branch whenever an End-of-line branch is signalled for the select code he specifies.

As with the ON KEY statement, there are three ways these branches can be taken —

ON INT #{select code} [, {priority}] CALL {subprogram name}
ON INT #{select code} [, {priority}] GOSUB {line identifier}
ON INT #{select code} [, {priority}] GOTO {line identifier}

Whenever an interrupt is signalled from an ISR for a particular select code, if ON INT has been executed for that select code, then at the end of execution of the BASIC line which was executing when the signal came, the indicated branch in the ON INT is taken.

In the GOTO version, the branch is "absolute", which is to say that the program goes to the line indicated and picks up its execution there, forgetting where it was before. This has the effect of an "abortive" type of branch, and should only be used by the BASIC programmer when he wants the program to resume execution at some pre-determined point after handling an interrupt, without regard to where the program was before the interrupt occurred.

In the CALL and GOSUB versions, the branch is only temporary. After the subprogram or subroutine has been executed and the SUBEXIT, SUBEND, or RETURN (as appropriate) has been executed, then the program returns to the line following the one where it was interrupted. This is the same as if the CALL or GOSUB was in between the interrupted line and the one following it.

The {line identifier} and {subprogram name} in the CALL, GOSUB, and GOTO statements are the same as elsewhere in BASIC, except that a CALL may not have any parameters.

The {select code} specified in an ON INT statement restricts the branching action to occurring only when the assembly language triggers the ON INT condition for that select code. The interrupt may have occurred in actuality on another select code. This can be a way of allowing more than one branch for interrupts from a single interrupting device.

As an example —

```
100 ON INT#3 GOSUB Print_result
110 ON INT#5 GOSUB End_data
```

Should an interrupt occur anywhere in the program, causing an assembly language interrupt service routine to be executed, that assembly language ISR has the capability to cause either the branch of line 100 or the branch of line 110 to be taken. Thus, an assembly language ISR signals BASIC either to print an intermediate result or to note that all data has been processed.

Signalling

The {select code} specified in an ON INT statement restricts the branching action to occurring only when a branch is "signalled" for that select code. In actuality, an interrupt may not have occurred on that select code at all. Conversely, an interrupt may occur on the select code, but BASIC and its ON INT condition may never hear about it. It is necessary for the ISR which does the actual handling of an interrupt to inform, or "signal", the operating system that the interrupt occurred and trigger the ON INT conditions which may be set up at the time.

The responsibility of the ISR to signal the ON INT is also an opportunity. This signalling allows you in an ISR to decide whether or not you want BASIC to know about the interrupt. If you do not want BASIC to know, simply do not signal the condition. The signalling also allows you to signal different interrupt conditions. An example of doing this might be a case where, after an interrupt, a peripheral indicates whether it wants to input or output data. Your routine could signal one select code to execute an input routine and signal another select code to execute an output routine.

To signal an ON INT, your ISR must execute the following instructions —

```
ISOURCE      LDB Isr_psw
ISOURCE      LDA =103B
ISOURCE      STA B,I
ISOURCE      ADB=3
ISOURCE      LDA Mask      ! Determines which SC to signal
ISOURCE      STA B,I
```

Mask necessarily contains the select code to be signalled. Rather than containing the number of the select code, however, it has the bit set for the appropriate select code. For example, if you are signalling select code 2, you set bit 2 to 1 in Mask and leave the others 0. Similarly, if you are signalling select code 5, you set bit 5. Thus, the statement containing Mask in the above could just as easily be a literal. For example —

```
LDA = 32
```

would signal select code 5.

If the select code is not known at assembly time or if the ISR is shared by more than one select code, the following segment of code can be used to build the appropriate mask. (Pa cannot be zero, because zero is not a valid select code for the Isr_access utility.)

```
ISOURCE Mask:   BSS 1               ! Storage word for the mask.
ISOURCE Sbl_1:  SBL 1               ! Shift B left instruction.
                 .
                 .
                 .
ISOURCE         LDA Pa              !
ISOURCE         CPA=0
ISOURCE         JMP Cant_use_zero
ISOURCE         ADA =-1             !
ISOURCE         IOR Sbl_1           ! Create the mask.
ISOURCE         LDB =1              !
ISOURCE         EXE A
ISOURCE         STB Mask            ! Store the mask.
                 .
                 .
                 .
```

When you want to signal a select code after others have already been signalled, a slightly different instruction sequence is required —

```
ISOURCE      LDB Isr_psw
ISOURCE      LDA =103B
ISOURCE      STA B,I
ISOURCE      ADB=3
ISOURCE      LDA Mask
ISOURCE      DIR
ISOURCE      IOR B,I      ! Ors in the select code
ISOURCE      STA B,I
ISOURCE      EIR
```

Mask is the same as above.

As a further example, suppose you want both to signal BASIC when a device sends a line-feed character to the computer, and to terminate the ISR's linkage. Then the ISR might appear as —

```
ISOURCE Lf:         EQU 10
                     .
                     .
                     .
ISOURCE Isr:        LDA R4
ISOURCE             CPA =Lf
ISOURCE             JMP Terminate
ISOURCE             STA R7
ISOURCE             RET 1
ISOURCE Terminate:JSM End_isr_high,I
ISOURCE             LDB Isr_psw            ! Signal BASIC
ISOURCE             LDA =103B
ISOURCE             STA B,I
ISOURCE             ADB =3
ISOURCE             LDA =1                 ! Signal "input"
ISOURCE             STA B,I
ISOURCE             RET 1                  ! Return to BASIC.
```

Prioritizing ON INT Branches

Since more than one interrupt may occur while a single BASIC statement is executing, it is possible that by the time the line finishes there may be a number of ON INT branches waiting to be executed. In such situations you may want to assure that some ON INT branches are taken before others, or that you finish one routine (caused by an ON INT GOSUB or ON INT CALL) before you start another. This can be achieved by using the {priority} option of the ON INT statement, thereby "prioritizing" the branching caused by interrupts.[1]

There is a "system priority" for ordering this interrupt branching. For an ON INT to be honored at the end of a BASIC line, its priority must be greater than the current system priority.

Initially, the system priority is set to 0. When a BASIC line finishes, and there is at least one ON INT branch pending which is greater than the system priority, then the system takes the branch associated with the ON INT with the greatest {priority}. The values assigned to {priority} may be any integer numeric expression from 1 to 15. If {priority} is omitted, 1 is assumed.

If the ON INT branch to be executed is a GOTO, then the system priority level is unchanged. But if the branch to be executed is a GOSUB or a CALL, then the system priority level is changed to the priority level of the ON INT. Whenever the subroutine or subprogram is finished executing, then the previous system priority level is restored.

1 This "prioritizing" also holds between the various types of end-of-line branch statements that have the priority parameter. Thus an ON KEY with high priority is executed before an ON INT with low priority.

Thus, with the GOSUB and CALL versions, there are two effects involving priorities —

- The subroutine or subprogram is not allowed to execute until its priority is the highest one pending.
- Whenever the subroutine or subprogram is executing, it locks out any other interrupting branches unless they have a higher priority.

With the GOTO version there are also two effects, slightly differing —

- The branch is not taken until it has the highest priority of all pending branches.
- The execution of the branch does not lock out any other branches, so that at the end of the line to which it branches, if there are other pending branches, the highest one of those is executed.

For example, suppose there are these four statements in effect —

```
ON INT #4, 1 GOTO Routine_4
ON INT #5, 9 GOSUB Routine_5
ON INT #6, 5 GOTO 1000
ON INT #7, 15 GOSUB Routine_7
```

and also suppose that at the end of some BASIC line in the program, an interrupt had been received from all four of the interfaces involved. Then the process of dealing with them proceeds like this —

EVENT	NEXT ACTION	SYSTEM PRIORITY
Reaches end of current BASIC line	GOSUB Routine_7	Changes from 0 to 15
Finishes Routine_7	GOSUB Routine_5	Changes from 15 to 9

Suppose at this point another interrupt is received from select code 7.

EVENT	NEXT ACTION	SYSTEM PRIORITY
Reaches end of current BASIC line in Routine_5	GOSUB Routine_7	Changes from 9 to 15
Finishes Routine_7	Returns to interrupted point in Routine_5	Changes from 15 to 9
Finishes Routine_5	GOTO 1000	Changes from 9 to 0
Finishes with line 1000	GOTO Routine_4	Stays at 0

Environmental Considerations

Changes in program environment (i.e., calling a subprogram or returning from one) can affect whether an ON INT is in effect or not.

Once executed, the CALL version of an ON INT is **always** in effect, if it is in the main program, until it is redefined by another ON INT or is specifically disabled (see below).

In the GOSUB or GOTO versions, the statement is in effect **only** in the same program environment. This is to say that if you have executed an ON INT statement in your main program, then it is effective only while your program is executing part of the main program. The instant the program goes into a subprogram (through a CALL statement), the statement is no longer effective until the execution returns to the main program. Similarly, if you define an ON INT in a subprogram, it is effective only while the program is executing that subprogram.

A side-effect occurs here when you use the CALL version of an ON INT. By calling the subprogram with an ON INT, you have the effect of locking out the other interrupts, except those which are executed in the subprogram itself and other CALL versions. This is regardless of priority. In the priority example in the previous section, if the ON INT#5 had been a CALL instead of a GOSUB, then the second interrupt from select code 7 would not have been acknowledged until the subprogram had finished.

Since recursive calls of subprograms are possible, it is also possible that many calls to the same subprogram may be stacked up because an interrupt from a different select code with a CALL version of an ON INT in effect may be received while processing the CALL caused by a previous interrupt.

Disabling ON INT Branching

The branching enabled by an ON INT statement can be disabled using an OFF INT statement for the same select code. It is effective for the ON INT statement within the same program environment (main program or subprogram) or for the CALL versions of the ON INT within any environment.

The statement has the form —

```
OFF INT #{select code}
```

where {select code} is a numeric expression for any valid interface select code between 1 and 13, inclusive.

The effect of the OFF INT statement is to disable the ON INT for that select code within the current environment. If there is no ON INT statement currently in effect for the select code, then the OFF INT has no effect.

The DISABLE and ENABLE statements work the same way for the ON INT statements as they do for the ON KEY statements. They should not be confused with the DIR and EIR machine instructions, which disable and enable the interrupt system.

Mass Storage Activities

For devices meeting the operating system's criteria for mass storage peripherals, utilities are provided for the reading and writing of records. The relationship between physical, logical, and defined records is discussed later in this chapter.

If a device has been specified in a MASS STORAGE IS statement in BASIC, as in —

```
MASS STORAGE IS ":F"
```

or is capable of being so specified, then it is possible to use utilities to access it. Note that the Mass Storage ROM is necessary to access any device other than the internal tape drive(s).

NOTE

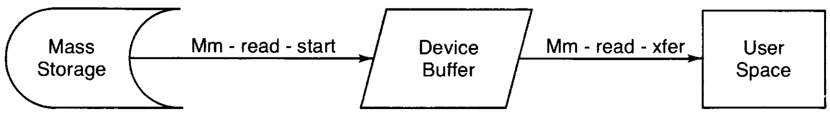
BUFFER# must not be used with files which are accessed using these utilities.

There are two utilities involved in reading from a mass storage device — Mm_read_start and Mm_read_xfer — and there are two utilities involved in writing to a mass storage device — Mm_write_start and Mm_write_test. The reading utilities are always used together. So, too, are the writing utilities.

Reading from Mass Storage

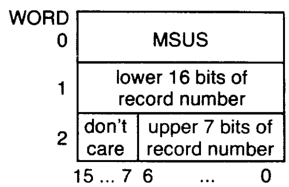
The flow of data to and from a mass storage device is buffered. For each device there is a "device buffer" in memory which holds data corresponding to a physical record (256 bytes). Device buffers are dynamically allocated by the operating system and their actual locations at any given time are of no concern.

To get information from a mass storage device into its device buffer, use the Mm_read_start utility. Then to get the information out of the buffer and into your user space, use the Mm_read_xfer utility. The transfer of data, therefore, looks something like this —

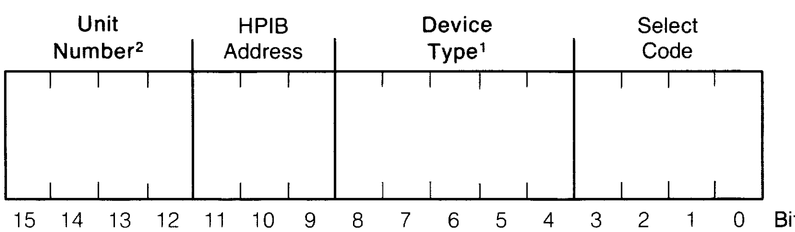


The utilities accomplish their purposes with the help of two locations containing vital information for their use. The first is the Mass Storage Descriptor (MSD) and the second is the Mass Storage Transfer Identifier (MSTID).

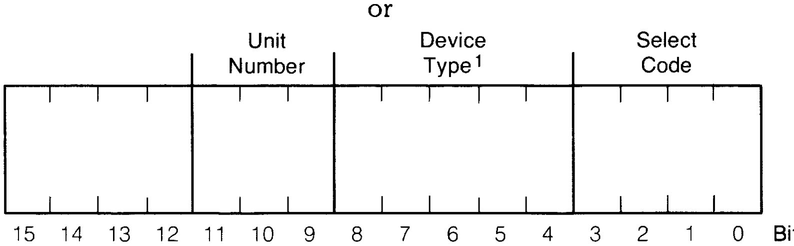
The MSD is three words in the ICOM region which contains the following information —



This information must be provided by your program. You must determine this information in advance of attempting the reading operation. The msus is given in one of two forms —



or



for the 9885MS Flexible Disk Drive.

If the MSUS word contains a −1, the mass storage device indicated by the MASS STORAGE IS statement is used. The instructions —

```
LDA = -1     ! -1 in the A register.
STA Msd      ! Store in the first word of the MSD.
```

1 The device type is the ASCII code for the type minus 100B.

2 For tape operations, bits 9-15 are zeroes.

specify the default mass storage device.

The MSTID is a single word. The information in it is returned by the Mm_read_start utility and used by the Mm_read_xfer utility.

The usual procedure in reading a record from mass storage (which is all that can be read at one time) is to call the Mm_read_start utility and then, if all goes well with that, to call the Mm_read_xfer utility. Because the latter utility may have to wait on the operating system or the device, it is possible the utility may return without having completed the transfer. In that case, it is your option either to loop back and keep trying, or to do something else and try again later.

UTILITY: Mm_read_start

General Procedure: The record number is determined, then the transfer of the record's contents is made from the device to the device buffer. If the buffer allocation causes a memory overflow, there is an error.

Special Requirements: The record number and msus must be loaded into the MSD in advance of the call. There must be a stable location (not changed by other activities) for the MSTID to be held.

Calling Procedure:

1. Store the msus and record number into the MSD area.
2. Load register A with the address of the MSD area.
3. Call the utility.

Exit Conditions:

RET 1 Occurs if there is a memory overflow during execution of the utility.

RET 2 Occurs if all went normally. Register A contains the MSTID. This should be immediately stored in the location reserved for it.

UTILITY: Mm_read_xfer

General Procedure: The MSTID is used to retrieve the record from the device buffer. The record is stored into a location set aside for this purpose.

Special Requirements: The MSTID must be available from a previous call to Mm_read_start. A location of 128 consecutive words must be set aside to hold the contents of the record when they are returned by the utility.

Calling Procedure:

1. Load register A with the contents of the MSTID.
2. Load register B with the address of the storage location for the data.
3. Call the utility. The transfer may not be completed on the first or subsequent calls (see exit conditions). In that case, to successfully complete the transfer, all three steps must be repeated.

Exit Conditions:

RET 1 Occurs when the transfer is not completed. It is up to your routine at this point to decide whether another attempt should be made immediately, or whether something else should be executed (and to come back later).

RET 2 Occurs when the transfer is complete. The location specified contains the data. If register A contains a non-zero value, an error occurred and A contains the error number. In addition to mass storage errors (80 through 99), error 19 is returned if the MSTID parameter is invalid.

CAUTION

PRESSING RESET (CONTROL STOP) DURING EXECUTION OF EITHER OF THE ABOVE UTILITIES MAY CAUSE A SCRATCH A TO OCCUR.

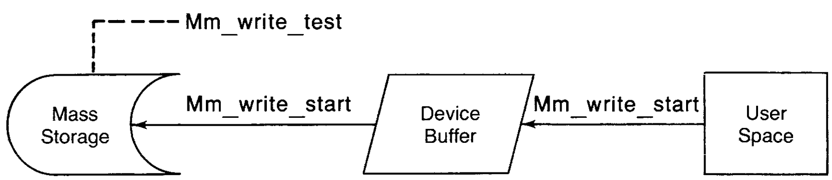
The following is an example of a typical call to these utilities to read a record from mass storage —

```
ISOURCE Number:  BSS 2
ISOURCE Msd:     BSS 3
ISOURCE Mstid:   BSS 1
ISOURCE Record:  BSS 128
                  .
                  .
                  .
ISOURCE          LDA =(('T-100B)*16+14) ! MSUS for ":T14"
ISOURCE          STA Msd            ! Create the MSD
ISOURCE          LDA Number         ! Store low-order bits of record number
ISOURCE          STA Msd+1
ISOURCE          LDA Number+1       ! Store high-order bits of record number
ISOURCE          STA Msd+2
ISOURCE          LDA =Msd
ISOURCE          JSM Mm_read_start! From device to buffer
ISOURCE          JMP Memory_overflow
```

```
ISOURCE         STA Mstid          ! Keep the MSTID
ISOURCE Fetch:  LDA Mstid
ISOURCE         LDB =Record
ISOURCE         JSM Mm_read_xfer   ! Transfer record to ICOM buffer
ISOURCE         JMP Fetch          ! Not completed (RET 1)
ISOURCE         SZA *+2            ! Check for errors (RET 2)
ISOURCE         JSM Error_exit
```

Writing to Mass Storage

Writing to mass storage is very much like reading from it. The flow of data is buffered. To get the data from the user space into the device buffer, and then to transfer the data from the buffer to the mass storage device, the Mm_write_start utility is used. Then a test can be made to determine when the transfer is complete by using the Mm_write_test utility. Thus, the transfer looks like —



As with the reading utilities, these utilities accomplish their purposes with the help of the same two locations — MSD and MSTID. They contain the same information as they do in the reading utilities and are used in a similar fashion.

UTILITY: Mm_write_start

General Procedure: The record number is determined, then the transfer of the data is made from the ICOM region to the device buffer. If the buffer allocation causes a memory overflow, there is an error.

Special Requirements: The record number and msus must be loaded into the MSD in advance of the call. There must be a stable location (not changed by other activities) for the MSTID to be held. The data to be transferred must be ready (256 bytes — 128 consecutive words).

Calling Procedure:

1. Store the data to be transferred in its location. Store the msus and record number into the MSD area.
2. Load register A with the address of the MSD area.
3. Load register B with the address of the data location.
4. Call the utility.

Exit Conditions:

RET 1 Occurs if there is a memory overflow during execution of the utility.

RET 2 Occurs if all went normally. Register A contains the MSTID. This should be immediately stored in the location reserved for it.

UTILITY: Mm_write_test

General Procedure: The MSTID is used to check to see if the data from the buffer has been transferred to the mass storage device.

Special Requirements: The MSTID must be available from a previous call to Mm_write_start.

Calling Procedure:

1. Load register A with the contents of the MSTID.
2. Call the utility. The transfer may not be completed on the first or subsequent calls (see exit conditions). In that case, to successfully test for a completed transfer, both steps in the calling procedure must be repeated.

Exit Conditions:

RET 1 Occurs when the transfer from the device buffer to the device is not completed. It is up to your routine at this point to decide whether another test should be made immediately, or whether something else should be executed (and to come back later).

RET 2 Occurs when the transfer is complete. If register A contains a non-zero value, an error occurred and A contains the error number. In addition to mass storage errors (80 through 99), error 19 is returned if the MSTID parameter is invalid.

CAUTION

PRESSING RESET (CONTROL STOP) DURING EXECUTION OF EITHER OF THE ABOVE UTILITIES MAY CAUSE A SCRATCH A TO OCCUR.

The following is an example of a typical call to these utilities to write a record to mass storage —

```
ISOURCE Number:  BSS 2
ISOURCE Msd:     BSS 3
ISOURCE Mstid:   BSS 1
ISOURCE Record:  BSS 128
                  .
                  .
                  .
ISOURCE          LDA =((`T-100B)*16+14 ! MSUS for ":T14"
ISOURCE          STA Msd           ! Create the MSD
ISOURCE          LDA Number        ! Store low_order bits of record number
ISOURCE          STA Msd+1
ISOURCE          LDA Number+1      ! Store high-order bits of record number
ISOURCE          STA Msd+2
ISOURCE          LDA =Msd
ISOURCE          LDB =Record
ISOURCE          JSM Mm_write_start ! Put record in buffer
ISOURCE          JMP Memory_overflow
ISOURCE          STA Mstid         ! Keep the MSTID
ISOURCE Test:    LDA Mstid
ISOURCE          JSM Mm_write_test ! Is transfer of data complete?
ISOURCE          JMP Test          ! Not completed
ISOURCE          SZA *+2           ! Check for errors
ISOURCE          JSM Error exit
```

System File Information

As an ASSIGN statement is executed in BASIC, a file-descriptor is created for that assignment in the operating system's files table. The ASSIGN statement essentially has two parameters — the file number and the file name (including the BASIC language mass storage unit specifier).

The file number is, for all practical purposes, an offset into the files table. The file name and the BASIC language mass storage unit specifier are translated and the critical information associated with them comprise an entry in the files table (i.e., the "file descriptor").

The file descriptor consists of 10 words containing the following information —

Word	Description
0	Lower 16 bits of the address of the first physical record in the file
1	Number of defined records in the file
2	BASIC's Current defined record number (i.e., an offset from the file's beginning).
3	BASIC's offset to current word within current defined record
4	Size of the defined record (in words)
5	Mass storage unit specifier (msus)
6	BUFFER# flag (0=no BUFFER# active)[1]
7	Check read status (0 = off, 1 = on)
8	Highest 7 bits of the first physical record in the file
9	(Reserved by the operating system)

Note that words 5, 0 and 8 contain the information necessary to create an MSD. You may access a file descriptor through two utilities — Get_file_info to obtain the information, and Put_file_info to change the information.

NOTE

A files table is created for each BASIC "environment" (i.e., main program and subprograms). When access is made through utilities to the files table, the table accessed is the one associated with the BASIC environment which called the assembly language program.

UTILITY: Get_file_info

General Procedure: The utility is given the file number and the location of a place to store the file descriptor. It retrieves the designated descriptor and stores it, provided the file has been assigned.

1 If this flag is non-zero, it indicates that a BUFFER# is active for this file. Therefore, Mass Storage utilities should no be used. Executing another ASSIGN statement for this file clears the BUFFER# flag.

Special Requirements: There must be a ten-word area available for the utility to store the information from the descriptor.

Call Procedure:

1. Load register A with the address of the ten-word area where you desire the information to be stored.
2. Load register B with the file number (an integer from 1 to 10).
3. Call the utility.

Exit Conditions:

RET 1 Occurs if the file is not currently assigned by a BASIC ASSIGN statement.

RET 2 Occurs if all went normally.

Here is an example of a routine which has a file number passed to it, and then gets the file descriptor —

```
                                .
                                .
                                .
ISOURCE File_descriptor:  BSS 10
ISOURCE File:             BSS 1
                                .
                                .
                                .
ISOURCE                   SUB
ISOURCE Parameter:        FIL
ISOURCE Routine:          LDA =File            ! Get file number
ISOURCE                   LDB =Parameter
ISOURCE                   JSM Get_value
ISOURCE                   JMP =File_descriptor ! Get file descriptor
ISOURCE                   LDB File
ISOURCE                   JSM Get_file_info
ISOURCE                   JMP No_file_error    ! File not assigned
                                .
                                .
                                .
```

UTILITY: Put_file_info

General Procedure: The utility is given the file number and the location of the area containing the new file descriptor information. It stores that information into the files table as indicated by the file number, provided that the file has been assigned.

Special Requirements: The new pointer information must be stored in the designated area before calling the utility. This information must be in the correct form and location or file difficulties may ensue. Most of the information is normally returned by the "Get_file_info" utility and only a couple of words are changed to change the pointer in the file (e.g., the current record and word numbers). Only words 2, 3, and 7 should be changed in the descriptor.

Calling Procedure:

1. Load register A with the address of the ten-word area where the information is stored.
2. Load register B with the file number (an integer from 1 to 10).
3. Call the utility.

Exit Conditions:

RET 1 Occurs if the file has not been assigned by a BASIC ASSIGN statement.

RET 2 Occurs if all went normally.

Here is an example where the next defined record in a file is specified —

```
File:            BSS 1   ! File number
File_descriptor: BSS 10  ! File information
                  .
                  .
                  .
                 ISZ File_descriptor+2  ! Increment record number
                 LDA =0
                 STA File_descriptor+3  ! Set word to 0
                 LDA =File_descriptor
                 LDB File
                 JSM Put_file_info
                 JMP No_file_error    ! File not assigned
```

Communication with BASIC Data Files

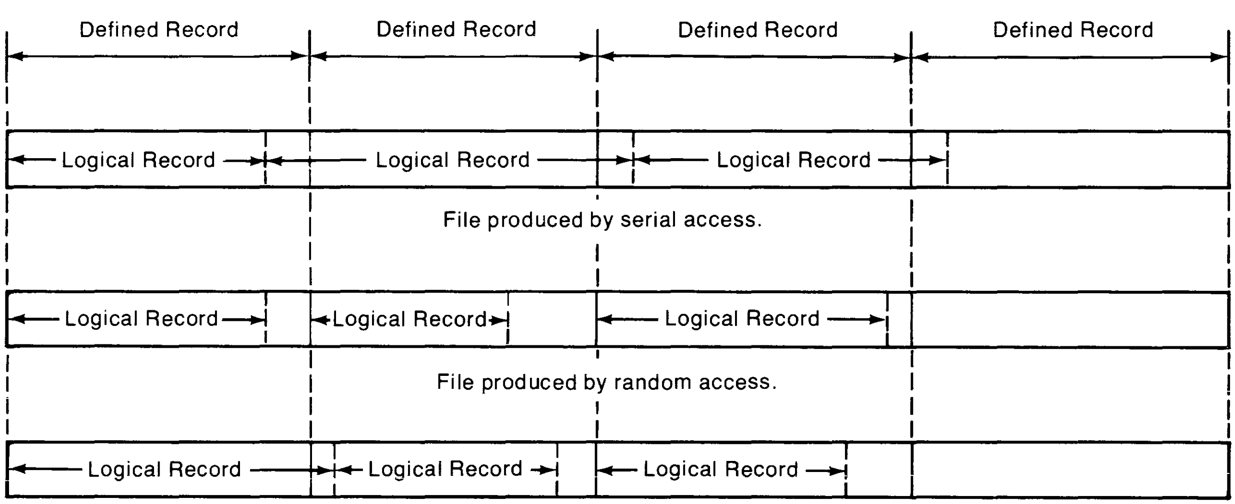
It is perfectly acceptable and practical for assembly language programs to write data patterns to data files and read them back. This has the advantages of simplicity and efficiency. However, such files cannot be properly read by the BASIC READ# statement or written for assembly routine use by the BASIC PRINT# statement. Therefore, if it is necessary for an assembly language program to read or write data which is compatible with READ# and PRINT#, the assembly language program must recognize and conform to the conventions used by these two BASIC statements. This section discusses these conventions.

Interrelation of Record Types

Recall from the System 45 Operating and Programming manual that there are three types of records used with the System 45 as follows:

- **Physical record** – 256-byte, fixed units which are established when a mass storage medium is initialized. Every file starts at the beginning of a physical record.
- **Defined record** – established using the CREATE statement. Defined records can be specified to contain any number of bytes in the range 4 to 32 767 (rounded up to an even number). The first defined record of a file starts at the beginning of a physical record.
- **Logical record** – a collection of data items that are grouped together conceptually. Different logical records may have different lengths within the same file. If a logical record is not immediately followed by another logical record and does not end on a defined record boundary, it is followed by either an EOR (end of record) or EOF (end of file) mark.

In order to locate logical records within a file, it is necessary to know the relationship between logical and defined records. This relationship depends on the method of file access used to write the information into the file. When a file is written using strictly serial file access, the first logical record starts at the beginning of the first physical record, the second logical record starts immediately after the first logical record and so on. Logical records may cross defined record boundaries. When a file is written using strictly random file access, each logical record starts at the beginning of a defined record and is contained entirely within the defined record. A hybrid method is also possible. With this method, logical records are written starting at the beginning of defined records other than the first one, and the logical records may cross defined record boundaries. Logical records may start immediately after other logical records, as well as at the beginning of defined records. Illustrations representing files produced by each of the three methods described above are presented here —



File produced by hybrid access.

The READ# and PRINT# statements read and write logical records which may be optionally positioned on defined record boundaries. Physical records are essentially invisible to the BASIC user. On the other hand, the assembly language mass memory utilities deal with physical records. To keep the relationship between defined and physical records simple, it is recommended that data files be created with 256 bytes per defined record (this is the default byte per record number used by the CREATE statement when the record length argument is not supplied). When 256 byte records are used, physical and defined records are identical. If you choose not to use 256 bytes per defined record, the relationship between physical and defined records is also fairly simple if the number of bytes per defined record is a power of 2 (e.g., 64) or is an integer multiple of 256 (e.g., 768).

Crossing Record Boundaries

The subject of what happens when a logical record crosses a physical and defined record boundary is now considered. The sequence of data words is not affected as the **physical record boundary** is crossed. For example, suppose there are three words remaining in a physical record and the next data item to be written is a real number (which requires four words). The first three words are written at the end of the current physical record and the last word is written at the beginning of the next physical record.

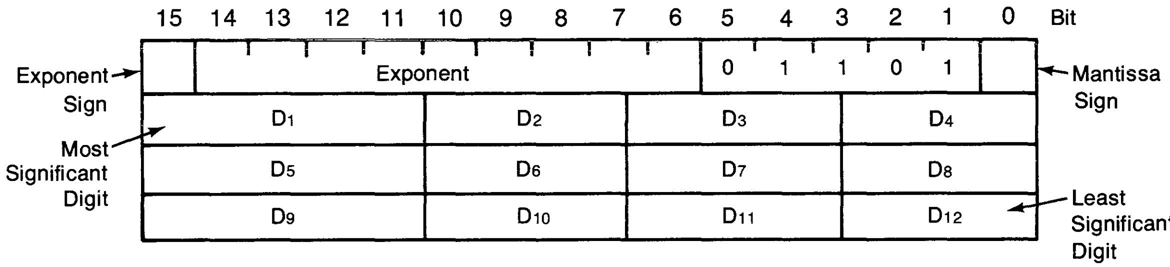
However, the same is not true when a sequence of data words crosses a **defined record boundary**. Numeric data items are not allowed to cross defined record boundaries. When writing a data item, the follow three cases exist:

- If there are enough words left in the current defined record to contain the item, the item is written in that record.
- If there are no words left in the current defined record, the item is written at the beginning of the next record.
- If there are one or more words left in the current record but not enough to hold the data item, an end of record mark is written immediately after the previous data item in the current record and the new data item is written at the beginning of the next record.

Of course, these cases apply when physical record boundaries coincide with defined record boundaries. A fourth case exits and involves an attempt to write a full-precision number into a file with 4 or 6 byte defined records or to write a string into a file with 4 byte defined records. If either operation is attempted, ERROR 61 results.

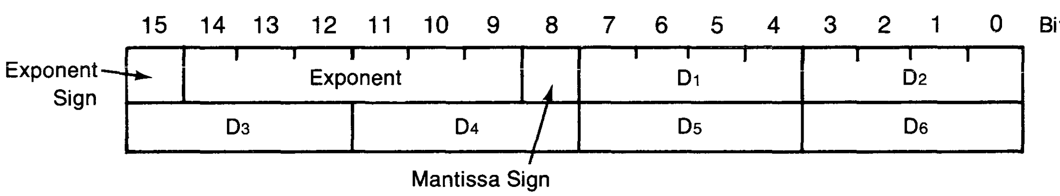
Strings may cross defined record boundaries but special rules apply in this case. These rules are described later when string data types are discussed.

A **full-precision number** exits in a data file as four words in a form tha looks like this —



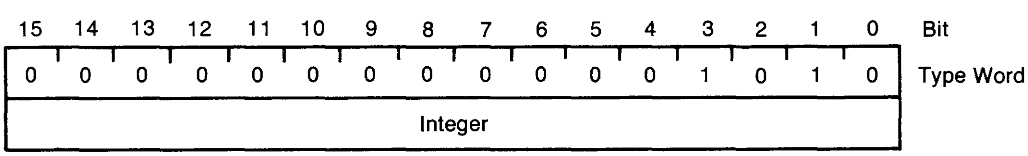
This is the same format as that shown in Chapter 3, except for the type bits which are used to identify the number as full precision. A full-precision number **must** have the type bits set to the pattern 01101 when written to a mass storage device, otherwise READ# will not interpret the data correctly. A full-precision number **must** have its type bits cleared before it is used with the math utilities or sent back to BASIC. Erroneous results occur if the type bits are not cleared. A full-precision number must not cross a defined record boundary.

A **short-precision number** exists in a data file as two words in the following form:



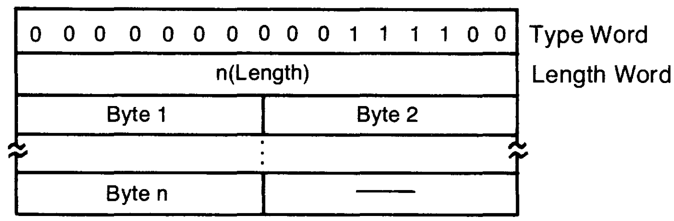
This is exactly the same as the usual short-precision format. READ# identifies short-precision numbers by the fact that D_1 and D_2 are valid BCD digits. A short-precision number must not cross a defined record boundary.

An **integer precision number** exits in a data file as two words in the following form —

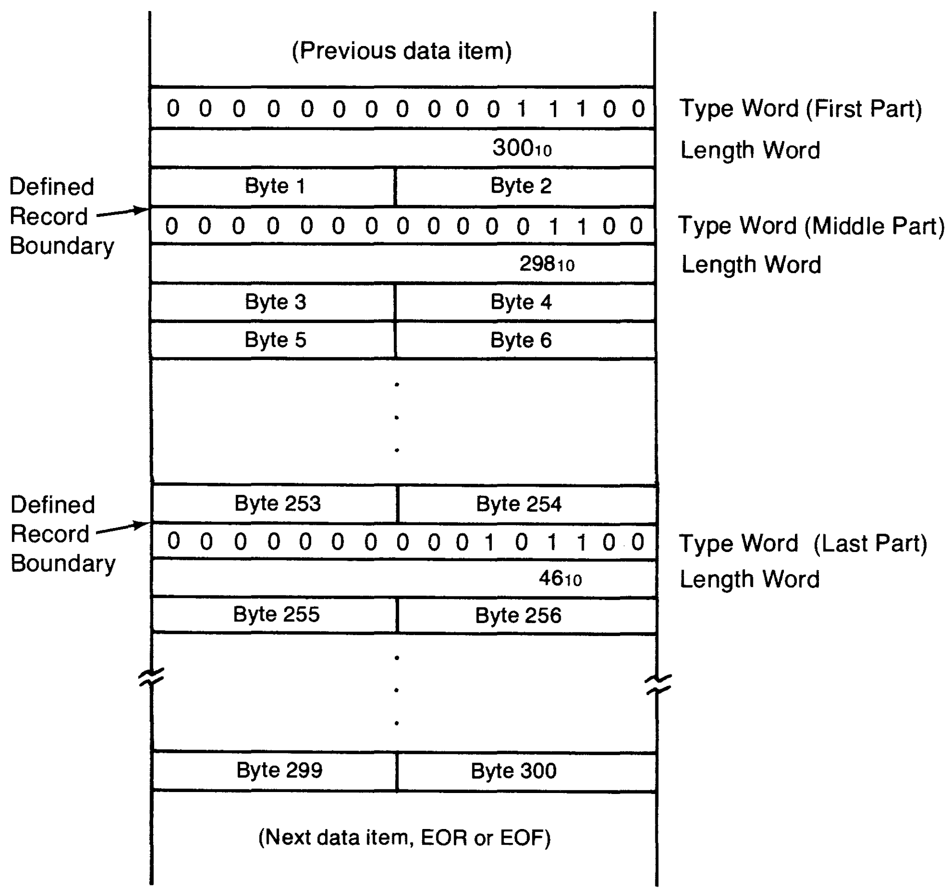


The first word is a type word which allows READ# to identify the data as an integer. The second word is the integer value in the usual two's complement form. An integer precision number must not cross a defined record boundary.

Strings are stored in data files in various forms, depending on how many defined record boundaries are crossed. The simplest case occurs when the string fits entirely within the current defined record. The fundamental format is illustrated here —



When the string does not fit entirely within on record, it is stored as a "first part", zero or more "middle parts" and a "last part". The following illustration represents a 300-byte string which has been written into 256-byte records starting at the third-to-last word of the record.



Note the different type words for the various parts. Also note that the length words contain the total number of bytes remaining in the string.

Strings are written according to the following rules:

1. If defined records are only 4 bytes long, then ERROR 61 results.
2. If the string fits entirely within the current record, the entire string is written into that record. (Null strings fall under this rule if there are at least 2 words available).
3. If there are 1 or 2 words left in the current record, an end-of-record mark is written after the previous data item. If there are 0, 1 or 2 words left in the current defined record (before an EOR was written), then the data file pointer is moved to the beginning of the next defined record and the string is then written starting in the new current record as in Step 2 above.
4. Otherwise, as much of the string as will fit in the current record is written as a first part string. Zero or more middle parts are written, one per defined record, and then the last part string is written.

File Marks

End-of-record (EOR) and end-of-file (EOF) marks exists as single word markers as shown below.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	

End-of-record Mark

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	

End-of-file Mark

An EOR indicates that there is no more valid data in the current defined record. If a serial READ# tries to read more data when the file pointer is positioned at an end-of-record mark or positioned past the end of the defined record, the READ# skips to the beginning of the next defined record and tries to read data there.

An EOF indicates an end of data. If a READ# tries to read more data when the file pointer is positioned at an end-of-file mark or past the end of the last defined record, then an ERROR 59 results unless there is an ON END# condition active for that file pointer.

For best results when writing data files, write EOR and EOF marks according to the following rules:

1. Write EOR marks as indicated in the discussion of string data files, given in the previous section and according to the rules outlined in the section "Crossing Record Boundaries". If these rules are not followed, the BASIC READ# statement will attempt to interpret the unused words at the end of defined records and will probably give ERROR 65, incorrect data type.
2. In a serial access file, write an EOF immediately after the last logical record. If there is no room in that record for the EOF mark, write the EOF at the beginning of the next defined record. If this is not done, you may not know where your data ends when you try to read it later. If another logical record is to be appended to the end of the previous data, the first word of the new data must overwrite the previous EOF. If there is no space in that record for an actual data item, the EOF must at least be replaced with an EOR.
3. If random access is used to find the end of data in a serial file, be sure that there is an EOF at the beginning of all unused defined records.
4. A defined record in a random access file can be made empty by writing an EOF at the beginning of that record.
5. The nature of programs that use random access is such that they usually do not try to read more data than was written. But for safety sake, it is a good idea to write an EOF or EOR after each logical record in a random access file, if there is room in each defined record.

Determining Data Types

The type of data item in a data file can be determined by ANDing the first word of the data item with 76B. The result (the type bits) can be used in conjunction with the following table to determine the data type:

Type Bits[1]	Data Type
12B	Integer number
32B	Full-precision number
14B	Middle part of a string
34B	First part of a string
54B	Last part of a string
74B	Total string
36B	EOR
76B	EOF
Other	If the right byte consists of two valid BCD digits the data type is a short-precision number.

1 The remaining codes have not yet been assigned but are reserved.

Printing

Three utilities are provided to enable you to gain access to the standard system printer: Printer_select, Print_string and Print_no_lf. An additional utility, To_system, allows you to expedite the printing process.

UTILITY: Printer_select

Background Information: Printer_select allows you to set the standard system printer to a select code of your choosing.

General Procedure: The utility is given the select code to be assigned as the standard system printer and the desired printing width. The utility makes the assignment and returns with the previous values of both the select code and printer width.

Special Requirements: The select code value must be in the range of 0 through 18 for the utility to work properly. The select code and associated device for committed printer select codes are as follows:

0 – internal printer
16 – CRT alpha raster
17 – display line of the CRT (as used for the DISP instruction)
18 – system message line of the CRT (as used for system error messages)

HP-IB devices are not allowed for use with the Printer_select utility.

Calling Procedure:

1. Load register A with the desired select code.
2. Load register B with the desired printer width.
3. Call the utility.

Exit Conditions: There are no error exits from the utility, so it always returns to the instruction following the call. Register A contains the value of the previous select, and register B contains the value of the previous printer width.

The utility can feasibly be used just to interrogate the current value of the printer's select code. However, a second call to the utility is needed in such cases to assure that the select is not changed by the first call. So, for example —

```
ISOURCE LDA =16
ISOURCE LDB =80
ISOURCE JSM Printer_select
ISOURCE STA Select_code
ISOURCE STB Printer_width
ISOURCE JSM Printer_select
```

This results in an unchanged printer specification and the values for the select code and width being stored in the ICOM area for future use.

Because of the possibility that a RESET (CONTROL STOP), or similar interruption, may occur between the first and second calls to the utility, it is recommended that the first call have a definite valid value for the select code in A (as above). In that way, should there indeed be an interruption, a valid select code for the printer can be assured.

UTILITY: Print_string

Background Information: Print_string allows you to print a string to the standard system printer. A carriage-return line-feed sequence is sent following the string.

General Procedure: The utility is given the address of a string, and it prints that string to the standard system printer.

Special Requirements: The string to be printed must be in standard string format (see "Data Structures" in Chapter 3). The string must be no longer than 506 characters.

Calling Procedure:

1. Load register A with the address of the string to be printed.
2. Call the utility.

Exit Conditions:

RET 1 If a memory overflow occurs during execution of the utility.

RET 2 If the STOP key is pressed during execution of the utility.

RET 3 If all goes normally.

For example —

```
ISOURCE String:      DAT 13,"ERROR IN CALL"
                      .
                      .
                      .
ISOURCE              LDA =String
ISOURCE              JSM Print_string
ISOURCE              JMP Overflowerror    ! Overflow condition.
ISOURCE              JMP Stop_routine     ! Stop key pressed.
ISOURCE              NOP                  ! Normal exit.
```

or

```
ISOURCE              LDA ==13,"ERROR IN CALL"
ISOURCE              JSM Print_string     ! No DAT statement!
ISOURCE              JMP Overflowerror    ! Overflow condition.
ISOURCE              JMP Stop_routine     ! Stop key pressed.
ISOURCE              NOP                  ! Normal exit.
```

The DAT statement and the location counter (*) can be used to calculate string length so that strings can be modified without having to constantly specify length. The following example illustrates this useful feature:

```
ISOURCE              NAM Stringlength     ! Module name.
ISOURCE              EXT Print_string     ! Declare externals.
ISOURCE String1:     DAT (Length1-*-1)*2  ! Length information.
ISOURCE              DAT "STRING #1"      ! Modifiable String1.
ISOURCE Length1:     *                    ! Location counter.
ISOURCE String2:     DAT (Length2-*-1)*2  ! Length information.
ISOURCE              DAT "STRING #2"      ! Modifiable String2.
ISOURCE Length2:     *                    ! Location counter.
ISOURCE              SUB
ISOURCE Entrypoint:   !                   ! Routine entry point.
ISOURCE               !
ISOURCE Print:       LDA =String1
ISOURCE              JSM Print_string     ! Print String#1.
ISOURCE              JMP Overflow         ! Overflow routine.
ISOURCE              JMP Stopkey          ! Stop key pressed.
ISOURCE              LDA =String2
ISOURCE              JSM Print_string     ! Print String#2.
ISOURCE              JMP Overflow         ! Overflow routine.
ISOURCE              JMP Stopkey          ! Stop key pressed.
ISOURCE               !
ISOURCE               !
ISOURCE              RET 1                ! Return to BASIC.
ISOURCE              END Stringlength
```

The strings in this example can be modified to any length less than 507 characters. The number of characters need not be placed in the DAT statement as this is taken care of in lines 30 through 50 and 60 through 80.

CAUTION

PRESSING RESET (CONTROL)(STOP)) DURING EXECUTION OF THE PRINT_STRING UTILITY OR THE PRINT_NO_LF MAY CAUSE A SCRATCH A TO OCCUR.

UTILITY: Print_no_lf

Background Information: Print_no_lf operates in an identical fashion to the Print_string utility except that no carriage-return line-feed sequence is appended to the end of the string. This is analogous to using PRINT (<print list>;) in BASIC.

General Procedure: The utility is given the address of a string, and it prints that string to the standard system printer.

Special Requirements: The string to be printed must be in standard string format (see "Data Structures" in Chapter 3). The string must be no longer than 506 characters.

Calling Procedure:

1. Load register A with the address of the string to be printed.
2. Call the utility.

Exit Conditions:

RET 1 If a memory overflow occurs during execution of the utility.

RET 2 If the (STOP) key is pressed during execution of the utility.

RET 3 If all goes normally.

For example —

```
                    .
                    .
                    .
ISOURCE             LDA ==27,"MESSAGE #1 IS CONCATENATED "
ISOURCE             JSM Print_no_lf
ISOURCE             JMP Overflow      ! Overflow routine.
ISOURCE             JMP Stopkey       ! Stop key pressed.
                    .
                    .
                    .
ISOURCE             LDA ==14,"TO MESSAGE #2."
```

```
ISOURCE          JSM Print_string
ISOURCE          JMP Overflow       ! Overflow routine.
ISOURCE          JMP Stopkey        ! Stop key pressed.
ISOURCE          NOP                ! Normal exit.
                  .
                  .
                  .
```

The result that is sent to the standard printer is —

MESSAGE# 1 IS CONCATENATED TO MESSAGE #2.

The Beep Signal

An audible tone (beep) can be produced from assembly language programs by storing 100000B into R7 while Pa=0. This procedure can be used in interrupt service routines as well as in background programs. Here is an example —

```
                  .
                  .
ISOURCE          LDB Pa          ! Save old Pa.
ISOURCE          LDA =0
ISOURCE          STA Pa          ! Clear Pa.
ISOURCE          LDA =100000B
ISOURCE          STA R7          ! Beep!
ISOURCE          STB Pa          ! Restore Pa.
                  .
                  .
```

Expediting I/O

The design of the System 45 operating system is such that an assembly language routine can be executing while there is one or more I/O operations pending or "queued up" by the system. This condition may arise when BASIC statements such as PRINT, OUTPUT, ENTER, PLOT, IASSEMBLE and others are executed in OVERLAP mode before an ICALL statement or when utilities such as Print_string or Print_no_lf initiate I/O from within the assembly language module itself. The operating system doesn't get a chance to move these I/O operations toward completion as long as the assembly routine is executing.

This fact is typically of little concern since the operating system resumes its attempt to complete the I/O operation as soon as the ICALL completes. However, there are three specific cases in which expedition of an I/O operation is useful or even necessary. These three cases follow:

1. when the assembly routine is waiting for a busy variable to become not busy.
2. when the assembly routine takes a long time to execute and the programmer wishes to continue working on queued up I/O.

3. when the assembly routine needs to guarantee that I/O to a particular select code has completed.

Case 1 has been discussed in Chapter 6. Case 2 can be taken care of by including an occassional JSM To_system in a long assembly routine. The third case might arise in situations where the routine must make sure that a message is printed on the CRT before starting a long computation process. This situation might also arise when the assembly routine must communicate with an I/O interface card which may be involved in an OVERLAPPED I/O operation. Consider the following example:

```
PRINTER IS 6
PRINT LIN(3),"HI THERE"
ICALL Mine
  .
  .
ISOURCE              SUB
ISOURCE Mine:        LDA =6
ISOURCE              STA Pa
ISOURCE              LDA =1
ISOURCE              STA R5
  .
  .
```

If this segment of code is executed in SERIAL, the ICALL would not begin until the PRINT is completed and there is no problem. If, however, the segment is executed in OVERLAP, the ICALL is allowed to begin, even though the operating system has not yet completed the PRINT. The results of this kind of situation are unpredictable.

A technique called "flushing" is used to ensure that all I/O operations on a particular select code have completed. The process of flushing involves interrogating a special table within the operating system to determine if an I/O operation is pending on a particular select code. The following routine flushes all I/O from the select code passed in the A register.

```
ISOURCE Flush_poiunter:  BSS 1
ISOURCE Flush_io:        SAL 1            ! Compute offset into table.
ISOURCE                  ADA =177000B     ! Compute pointer into table.
ISOURCE                  STA Flush_pointer
ISOURCE Flush_loop:      LDA Flush_pointer,I  ! Is select code busy?
ISOURCE                  SZA Flush_done       ! Yes.
ISOURCE                  JSM To_system
ISOURCE                  JMP Flush_loop
ISOURCE Flush_done:      RET 1
```

The flushing technique should not be used in the following two cases:

1. Mass memory devices: Use the mass memory utilities to communicate with mass memory devices.
2. The Isr_access utility: It automatically flushes the select code of all activity.

Chapter 8
Debugging

Summary: This chapter describes techniques for isolating and correcting logic problems in assembly programs. Included in the discussion are techniques for stepping through programs, getting dumps, patching, and using the keyboard.

The assembly system has provided you with a number of BASIC language tools to help you debug your assembly language programs during their development stages.

These tools are for run-time debugging, so your source code must have been assembled into object code and stored in the ICOM region before attempting to use any of the debugging features detailed in this chapter.

There are three classes into which these tools fall: stepping through programs, dumps, and value checking. There is also an additional capability provided for the correction of some errors — patching.

The BASIC statements available for debugging are —

```
IBREAK
IBREAK ALL
IBREAK DATA
ICHANGE
IDUMP
INORMAL
IPAUSE OFF
IPAUSE ON
```

and the following BASIC functions are available —

```
DECIMAL
IADR
IMEM
OCTAL
```

Symbolic Debugging

Many statements allow symbolic addressing. The general rules are —

An {address} or {assembled location} can have two forms —

{symbol} [, {numeric expression}]
{expression} [, {numeric expression}]

where,

{symbol} is an assembly location. It may be either a label for a particular machine instruction, an assembler-defined symbol or a symbol defined by an EQU instruction.

{expression} may be a numeric or string expression. Variables in expressions are assumed to be BASIC variables. If numeric, a decimal calculation is done and the result is interpreted as an octal value; an error results if the result is not an octal representation of an integer. If a string expression is used, the string must be interpretable as either an octal integer constant or a known assembly symbol.

{numeric expression} serves as a **decimal** offset from the given label or constant. Variables in these expressions are assumed to be BASIC variables. An undefined BASIC variable is always given the value 0.

Stepping Through Programs

"Logic" difficulties are some of the hardest problems to solve in debugging programs. In batch environments, the usual solution is to print the contents of variables at critical points in the program or to print dumps. The capabilities for both of these methods are provided. However, advantage has been taken of the interactive, "hands-on" nature of the 9845 and a feature has been added which allows you to execute the assembly statements individually. This permits you to examine the flow of the program as it executes rather than having to decipher a dump or trying to print the contents of specific variables at what you guess is the critical point.

If you wish to look only at particular points in the program, or at particular variables, there is also the ability to establish "break points" for these items, so that your debugging routines can be invoked only when certain conditions arise. You can also establish different routines for different break points, adding to the flexibility.

Individual Instruction Execution

Normally, all BASIC lines, including the ICALL statement, act as a **unit**. That is to say, whenever you press the PAUSE key, the line which is currently executing is allowed to finish before the program is actually interrupted. Thus, if you press PAUSE during execution of the line —

```
100  LET A=1+1
```

the line finishes and the variable A contains the value 2. Then the PAUSE takes effect. The same is true of a line containing an ICALL statement.

For example, if you press PAUSE during the execution of —

```
120  ICALL Sort(A(*))
```

then the assembly routine completes before the PAUSE is honored. This is not always desirable, especially during debugging of the assembly routine. This technique does not allow you to look at the execution of the routine to help you determine what may be going wrong.

The same problem occurs with the STEP key. Pressing STEP causes an entire BASIC line to be executed. Thus, if you stepped through line 120 as above, the entire routine Sort would be executed, and you would not be able to observe its execution on an instruction-by-instruction basis.

To permit you to analyze the execution of assembly language routines, an executable BASIC statement has been provided —

```
IPAUSE ON
```

Now, should you have the sequence in your program —

```
110  IPAUSE ON
120  ICALL Sort(A(*))
```

then pressing PAUSE during the execution of line 120 would cause program execution to be interrupted after completion of whatever machine instruction is being executed at the time. Further, the assembly language source line associated with the following instruction is displayed according to certain rules.

If the source lines are still in memory when you press PAUSE (e.g., you just assembled the object code which you are running), then the source line is displayed. If the source is no longer in memory (e.g., the object code was obtained through an ILOAD), then the instruction displayed is the result of a "reverse assembly". If there is an operand with an instruction which is reverse assembled, then the octal value of that operand is displayed (this is because the reverse assembly process has no way of knowing what symbols you might have used to assemble the instruction originally).

After pressing PAUSE, all you have to do to resume normal execution is press CONT.

After pressing PAUSE, you may want to observe the flow of execution of your assembly routine. This can be done by successively pressing the STEP key. Each time the key is pressed, another machine instruction is executed and the assembly source line associated with the next machine instruction is displayed. You may continue this way for as long as you like — until you press CONT to allow processing to proceed uninterrupted until the end of the routine.

Of course, the STEP key can be used to step through the BASIC program as you are used to doing. That feature is unchanged. It is possible, therefore, to "step into" the assembly language routine from the BASIC (i.e., you need only STEP into line 120 above) and not have to use the PAUSE key at all.

In summary, IPAUSE ON allows two unique features —

- The PAUSE key can be used to halt execution within an assembled routine.
- The STEP key can be used to execute individual assembly language instructions.

Some key things to remember in using the IPAUSE ON facility —

- This is an execution-time debugging tool. You must be executing your previously-assembled object code with an ICALL statement.
- If the source code is available for display, it will be displayed, otherwise the line is "reverse assembled".
- Utilities are not stepped instruction-by-instruction, but rather as a unit.
- The STEP key performs in BASIC just as before.
- Keeping the STEP key and the REPEAT key depressed causes repeated execution of the stepping function, the same as in BASIC.

By way of example, suppose you had the following source code —

```
10   DIM A$[10]
20   ICOM 100
30   IPAUSE ON
40   IASSEMBLE Extract
50 Loop:  LINPUT A$
60   PAUSE
70   ICALL Extract(A$)
80   PRINT "<";A$;">"
90   GOTO Loop
100  ISOURCE            NAM Extract   ! Extracts part of a
110  ISOURCE                          ! string preceding comma
120  ISOURCE            EXT Get_value,Put_value
130  ISOURCE String:    BSS 6
140  ISOURCE            SUB
150  ISOURCE Parameter:STR
160  ISOURCE Extract:   LDA =String       ! Retrieve string
170  ISOURCE            LDB =Parameter
180  ISOURCE            JSM Get_value
190  ISOURCE            LDB String        ! Initialize counter
200  ISOURCE            LDA =String       ! Initialize stack pointer
210  ISOURCE            SAL 1
220  ISOURCE            ADA String
230  ISOURCE            ADA =1
240  ISOURCE            STA C
250  ISOURCE            CBL
260  ISOURCE Loop:      WBC A             ! Retrieve next character
270  ISOURCE            CPA =',           ! Is it a comma?
280  ISOURCE            JMP Yes
290  ISOURCE            DSZ B             ! Decrement.  Done?
300  ISOURCE            JMP Loop
310  ISOURCE            RET 1             ! No commee, no extractee
320  ISOURCE Yes:       ADB =-1           ! Found comma, extract
330  ISOURCE            STB String        !    by changing length
340  ISOURCE            LDA =String       !    then extracting
350  ISOURCE            LDB =Parameter
360  ISOURCE            JSM Put_value
370  ISOURCE            RET 1
380  ISOURCE            END Extract
```

Then the following would be the display lines you would see as you executed this program using the STEP key —

```
10   DIM A$[10]
20   ICOM 100
30   IPAUSE ON
40   IASSEMBLE Extract
50 Loop:  LINPUT A$
?
12345,6789
160  00053 002026 Extract:LDA =String  ! Retrieve string
170  00054 006026     LDB =Parameter
180  00055 142026     JSM Get_value
190  00056 007763     LDB String       ! Initialize counter
200  00057 002022     LDA =String      ! Initialize stack pointer
210  00060 170600     SAL 1
220  00061 023760     ADA String
230  00062 022022     ADA =1
240  00063 030016     STA C
250  00064 070510     CBL
260  00065 074760 Loop:WBC A           ! Retrieve next character
270  00066 012017     CPA =',          ! Is it a comma?
290  00070 054001     DSZ B            ! Decrement.  Done?
300  00071 067774     JMP Loop
260  00065 074760 Loop:WBC A           ! Retrieve next character
270  00066 012017     CPA =',          ! Is it a comma?
290  00070 054001     DSZ B            ! Decrement.  Done?
300  00071 067774     JMP Loop
260  00065 074760 Loop:WBC A           ! Retrieve next character
270  00066 012017     CPA =',          ! Is it a comma?
290  00070 054001     DSZ B            ! Decrement.  Done?
300  00071 067774     JMP Loop
260  00065 074760 Loop:WBC A           ! Retrieve next character
270  00066 012017     CPA =',          ! Is it a comma?
290  00070 054001     DSZ B            ! Decrement.  Done?
300  00071 067774     JMP Loop
260  00065 074760 Loop:WBC A           ! Retrieve next character
270  00066 012017     CPA =',          ! Is it a comma?
280  00067 066004     JMP Yes
320  00073 026013     ADB =-1          ! Found comma, extract
330  00074 037745     STB String       !    by changing length
340  00075 002004     LDA =String      !    then extracting
350  00076 006004     LDB =Parameter
360  00077 142010     JMP Put_value
370  00100 170201     RET 1
80   PRINT "<";A$;">"
<12345>
90   GOTO Loop
50 Loop:  LINPUT A$
```

Note that the address of the instruction, as well as the octal value of the instruction, is displayed along with the source line.

This stepping facility can also be used, quite effectively, with the IBREAK statement (discussed below).

Should the IPAUSE ON facility be no longer desired, it can be turned off with —

```
IPAUSE OFF
```

The two statements can appear repeatedly in a program, allowing the stepping facility to be used in testing some programs but skipping over already proven programs. For example, suppose you had two programs — Sorta and Sortn — but the first was already tested and the second was not. Then this sequence might appear in your program —

```
110  IPAUSE OFF
120  ICALL Sorta(A$(*))
130  IPAUSE ON
140  ICALL Sortn(A(*))
```

Stepping through this sequence results in lines 110, 120, and 130 executing without interruption, but line 140's call to Sortn would be executed instruction-by-instruction.

Executing IPAUSE ON when the facility is already in effect causes no change. Similarly, executing IPAUSE OFF when the facility is already off causes no change.

Both IPAUSE ON and IPAUSE OFF can be executed from the keyboard.

Setting Break Points

It is possible to define points in an assembly language routine where the execution should pause should it ever reach that point. These are called "break points". They can be used to pause execution — allowing you to utilize the stepping activity described above in IPAUSE ON or to investigate the contents of variables, etc. They can also be used to allow branching to some BASIC routine, giving you the power of BASIC in doing some of your debugging.

Simple Pausing

To simply pause at a break point, you need to execute the following statement in advance of reaching that point (either in the program or from the keyboard) —

`IBREAK` {address}

where {address} is the assembled location[1] for the break point desired.[2] Following execution of this statement, anytime the program execution reaches this address, it pauses. You may do any keyboard operations necessary at this point, or you may start stepping the program, (if IPAUSE ON has been executed), or you may resume execution using the CONT key. The address must have been assembled before the IBREAK is executed.

1 See "Symbolic Debugging" in this chapter for the definition of "assembled location".

2 The use of IBREAK significantly slows execution of assembly programs.

If you were to execute —

```
IBREAK Hook,4
```

then every time the fourth word past assembly label "Hook" is reached during execution, the program execution pauses. If you were to execute —

```
IBREAK Hook+4
```

then Hook is assumed to be a BASIC variable, and the result of the expression is assumed to be an absolute address using whatever the value of Hook is when the statement is executed.

You can also specify the number of occurrences of reaching a break point before pausing should come into effect. This is done by executing —

IBREAK {address} ; {counter}

where {counter} is a numeric expression; any variables within {counter} are BASIC variables. A pause occurs when {address} has been reached {counter} number of times. {counter} is reset after each pause.

When a break point is reached and a pause is to be taken, the pause takes place **before** execution of the contents of that address.

After execution of the IBREAK statement, the contents of the assembled location for the break point are changed by the operating system; however, this does not affect the execution of the instruction contained therein.

If an ICALL statement is executed from the keyboard and an IBREAK is active for a location within the ICALLed routine, program execution is returned to BASIC when the breakpoint is reached. Stepping of the assembly language routine is halted and the CRT is cleared.

Transfers

Instead of just pausing at a break point, it is possible to branch to a BASIC routine. The intent of this facility is to give you access to BASIC's capabilities, particularly the printing and variable-testing facilities, during your debugging efforts.

The branch can be any of the three standard forms of BASIC branching —

```
IBREAK {address} [; {counter} ] CALL {subprogram}
IBREAK {address} [; {counter} ] GOSUB {line identifier}
IBREAK {address} [; {counter} ] GOTO {line identifier}
```

When either CALL or GOSUB has been designated, execution of the assembly language routine is suspended when {address} is reached. Then the designated subprogram or subroutine is executed. When that subprogram or subroutine is completed, then execution of the assembly language routine resumes with {address}.

When GOTO is specified, an unconditional branch is taken when {address} is encountered and execution of the assembly language routine is terminated.

{counter} performs the same as in the simple pausing form.

In the GOSUB and GOTO forms, there is an "environmental" restriction. The {line identifier} must be in the same BASIC environment (i.e., main program or subprogram) as that in which the IBREAK statement is executed. More on this in "Environments" below.

You should avoid recursive use of the ICALL statement when using the IBREAK statement to branch to a BASIC subroutine or subprogram. The problem arises when an ICALL statement in the BASIC debug subroutine or subprogram calls the broken assembly routine. The IBREAK transfer occurs at the same assembly routine address each time it is encountered. This process results in non-productive looping.

Environments

The GOSUB and GOTO types of break points are related to the BASIC "environment" (i.e., main program or subprogram) in which they are executed. Whenever an IBREAK statement of either type is encountered, the resulting break point is effective only for the environment in which the statement is located. The CALL version of break points is in effect in all environments.

For example —

```
200  SUB Test
210  IBREAK Hook GOTO Check_hook
         .
         .
         .
```

the break point established for "Hook" is good only in the subprogram "Test". Leaving "Test" causes the break point to be cleared.

Executing an IBREAK statement from the keyboard is effective only for the environment executing at the time the statement is made. For example, if the following program lines had been executed —

```
200  SUB Test
210  PAUSE
```

and while the pause caused by line 210 is still in effect —

```
IBREAK Hook GOTO Check_hook
```

is executed, then the break point established for "Hook" is good only in the subprogram "Test". As with the above, leaving Test causes the break point to be cleared.

If no program is executing when an IBREAK is executed from the keyboard, then the main program is considered to be the environment for the break point. If the program is replaced, as with a GET or a LOAD, then the break point is cleared.

If a LINK command is used to replace all or part of a program, existing break points are still active. If the LINK eliminated the line label or subprogram referenced in the IBREAK, then ERROR 186 results when the break point is reached. If a GET command is used to replace all or part of a program, all GOTO/GOSUB breaks are cleared. IBREAK CALLs are still active. Again, if the line label or subprogram referenced by the IBREAK is eliminated, then ERROR 186 results. If the program is replaced with a LOAD, all break points are cleared. You must re-execute the IBREAK statements in the new program. Only ENT and SUB symbols are defined in this new program until an IASSEMBLE is executed.

Care should be taken when calling BASIC subroutines or subprograms after an IBREAK has been set and before an ICALL has been executed. A CALL to a subprogram clears break points of the IBREAK...GOTO and IBREAK...GOSUB varieties; however, IBREAK...CALL is not cleared. This is because CALL executes an INORMAL which clears all break points except IBREAK...CALL. (An INORMAL is also executed when the (RUN) key or (CONTROL)(STOP) keys are pressed). Here is an example of break points being cleared by a CALL —

```
10     IDELETE ALL                            ! Clear ICOM area.
20     ICOM 100                               ! Set aside 100 words.
30     IASSEMBLE ALL                          ! Assemble all modules.
40     IBREAK Middle GOSUB Breakfound         ! Break at location Middle.
50     CALL Callable                          ! CALL subprogram.
60     ICALL Entrypt                          ! Do assembly routine.
70     END
80 Breakfound: PRINT "Breakpoint found."      ! Break subroutine.
90     RETURN                                 ! Subroutine end.
```

```
100
110       ISOURCE              NAM Example    ! Module name.
120       ISOURCE               !
130       ISOURCE               !
140       ISOURCE              SUB
150       ISOURCE Entrypt:     LDA =Sc        ! Routine entry point.
160       ISOURCE               !
170       ISOURCE               !
180       ISOURCE Middle:      LDA =String1   ! Break location.
190       ISOURCE               !
200       ISOURCE               !
210       ISOURCE              RET 1          ! Return to BASIC.
220       ISOURCE              END Example    ! Module end.
230       SUB Callable                        ! CALLed subprogram.
240       PRINT "Subprogram."
250       SUBEND                              ! Subprogram end.
```

The break point is cleared after execution of line 50.

Keeping in mind that different BASIC environments exist for the main program, each subprogram and each multi-line function, IBREAK...GOTO and IBREAK...GOSUB remain in effect only within the BASIC environment in which they are declared. IBREAK...CALL remains in effect in all environments. A maximum of eight IBREAK...CALLs are allowed.

Data Locations

Break points can also be established for data locations. This is done with —

`IBREAK DATA` {address}

In this case, {address} is presumed to be a data location referenced by other instructions. Whenever it is referenced by execution of some instruction, the pause occurs.

If you were to say —

```
IBREAK DATA Renras
```

then whenever "Renras" is referenced, such as in —

```
LDA Renras
```

a pause would occur for that instruction.

A counter can also be specified with this form of break point —

```
IBREAK DATA {address} ; {counter}
```

{counter} is of the same form, and operates in an identical fashion, to the counter of the non-DATA form of break point.

Because the XFR machine instruction may access a particular location twice when it is executed, the break point on a data location may not operate correctly if the instruction referencing it is an XFR. The way to avoid this incorrect operation of the break point is to set {counter} to 2. (The only time this problem occurs is when the destination area for the XFR overlaps the origination area.)

Symmetry suggests that you should also be able to branch to BASIC routines with the DATA form of break point just as you can with the non-DATA form. And so you can —

```
IBREAK DATA {address} [ ; {counter} ] CALL {subprogram}
IBREAK DATA {address} [ ; {counter} ] GOSUB {line identifier}
IBREAK DATA {address} [ ; {counter} ] GOTO {line identifier}
```

They operate in an identical fashion to transfers of the non-DATA type and are under the same "environmental" restrictions.

In order to determine whether an address is being referenced, each instruction is "interpreted" (that is, analyzed for its components). Resultantly, a program runs much slower while an IBREAK DATA statement is in effect.

In addition to the pausing capability, using IBREAK DATA also allows trapping on "protected memory" violations (see "Stepping vs. Running" section of this chapter).

IBREAK Everywhere

You may have a total of eight (8) break points (regardless of type) in effect at a given time, except for one extreme case. It may be desirable to establish a break point at every location in the ICOM region. This can be accomplished with —

```
IBREAK ALL
```

This statement overrides all other IBREAK statements and causes a pause before execution of every instruction in the ICOM region. There are also branching forms —

```
IBREAK ALL CALL {subprogram}
IBREAK ALL GOSUB {line identifier}
IBREAK ALL GOTO {line identifier}
```

Note, however, that there is no {counter} in any of these forms.

Number of Break Points

As was mentioned above, there can be no more than eight (8) IBREAK statements in effect at one time, that is to say within the same environment. And only one IBREAK ALL can be in effect at a given time.

In addition, there can only be one IBREAK or IBREAK DATA each in effect for a given {address}. Executing an IBREAK or IBREAK DATA with the same {address} as specified in an already effective IBREAK or IBREAK DATA statement causes the newly-executed statement to override the previous one. While there may be an IBREAK and IBREAK DATA both for the same {address}, the capability is not a useful one.

Clearing Break Points

There are a number of ways that break points can be cleared. One way as has already been mentioned, is leaving the BASIC environment, which clears any GOSUB or GOTO type of break points. Another way is to reassemble the module containing the break points. A third way is to execute an INORMAL statement. This statement has the form —

```
INORMAL {address}
```

After execution of the statement, whatever form of break point is established for the address (except IBREAK ALL) is cleared.

If {address} is omitted in this statement —

```
INORMAL
```

then all break points are cleared. This is the only way to clear an IBREAK ALL which may be in effect.

Interrogating Processor Bits

During execution of a break point, the values of three processor flags are stored in specified registers so that you can interrogate them. They are —

Decimal Carry	stored as least significant bit in location 36B
Extend	stored as most significant bit in location 37B
Overflow	stored as least significant bit in location 37B

Dumps

A common tool of debugging is the memory "dump". This is a print-out (or display) of the contents of selected locations in the memory. A typical use is to dump areas of the ICOM containing data so that the actual contents at some point during execution can be compared with the expected contents. All of this is in the hope that the comparison yields differences which give a clue as to the source of the difficulties being encountered.

This tool is provided through the IDUMP statement which has the form —

IDUMP {location} [; {location} [;...]]

This statement can be placed in a program to be executed (perhaps as the result of a branching IBREAK statement) or it can be executed from the keyboard (perhaps during a pause caused by stepping or IBREAK).

Any number of {location}s can be specified. They can take a number of forms. The simplest is —

{address}

Thus, IDUMP {address} prints the contents of {address} to the current system printer. The contents are printed in their octal representation. For an explanation of {address}, see the "Symbolic Debugging" section of this chapter.

{location} can specify a whole range of addresses by using the form —

{address} TO {address}

With this form, the IDUMP statement prints the contents of all addresses starting with the first and ending the last specified {address}. If the second address is numerically smaller than the first, then a "wrap-around" through the end of memory into the top of memory is taken. For example, if you execute —

```
IDUMP 177776 TO 1
```

then the contents of four addresses would be printed — those for 177776, 177777, 0, and 1, in that order. Again, the contents are printed in their octal (base-8) representation.

Addresses are always specified in their octal representation, or symbolically (such as "Hook" or "Loop"). This is the same as for an assembled location, which is what {address} happens to be.

Care must be used with symbolic addressing. In the statement —

```
IDUMP Hook TO Hook + 4
```

the first "Hook" is interpreted as an assembled location. Since the second "Hook" appears in an expression, it is interpreted as a BASIC variable. If it is undefined, this expression is evaluated as 4. To dump the fourth word past the assembled location "Hook", use the statement —

```
IDUMP Hook TO Hook,4
```

The output of the IDUMP statement is always printed to the current system printer. It is in octal form, unless otherwise specified. This specification is accomplished by preceding {address} with {mode selection}, which is one of the following —

`ASC` for ASCII character representation
`BIN` for binary representation (base-2)
`DEC` for decimal representation (base-10)
`HEX` for hexadecimal representation (base-16)
`OCT` for octal representation (base-8)

Thus, the general form of {location} is —

[{mode selection}] {address} [`TO` {address}]

As an example of all this, take the example program at the beginning of the chapter. If a couple of statements are added so that the main BASIC program reads —

```
10   DIM A$[10]
20   ICOM 100
30   IASSEMBLE Extract
40   IBREAK Loop GOSUB Dump
50   IDUMP 41 TO 104       ! Dump of ICOM region
60   PRINT
70 Loop:  LINPUT A$
80   ICALL Extract(A$)
90   PRINT "<";A$;">"
100  GOTO Loop
110 !
120 ! Dump A,B registers in octal form,
130 ! string length in decimal form, and
140 ! and the string in character form
150 !
160 Dump:  IDUMP A TO B;DEC String;ASC String,1 TO String,5
170  PRINT
180  RETURN
```

then running it results in the following print-out —

```
000041: 000005 030462 031464 032454 033067 034071 022265 100003 022607 000012
000053: 021335 000001 100207 000000 000205 002025 006025 142025 007756 002021
000065: 170600 023753 022021 030016 070530 141714 012016 066004 054001 067774
000077: 170201 026012 037740 002003 006003 166007

000000: 000115 000012
000041: +00010
000042: 12345,6789$5%

000000: 000071 000011
000041: +00010
000042: 12345,6789$5%

000000: 000070 000010
000041: +00010
000042: 12345,6789$5%

000000: 000067 000007
000041: +00010
000042: 12345,6789$5%

000000: 000066 000006
000041: +00010
000042: 12345,6789$5%

<12345>
```

Value Checking

Value checking is a method of tracing the value of variables in your assembly language program using the interactive capabilities of the 9845. You already have been introduced to break points and dumps in earlier sections. The capability of value checking serves as a useful adjunct to these procedures.

The value checking of assembly "variables" is similar to the monitoring of variables in BASIC during a debugging phase. Just as you would use a live-keyboard operation or judiciously placed PRINT statements to trace the execution of a program or the change in value of a variable in a BASIC program, so too can you use the monitoring tools for assembly programs.

Functions

Four additional functions are provided as extensions to BASIC which can be useful in the monitoring of values in an assembly language program. The four are —

```
DECIMAL
IADR
IMEM
OCTAL
```

They can be used as other than monitoring tools, but their descriptions here are primarily in that context. As functions, these items can be easily adapted for use in the special function keys.

DECIMAL

This function has the form —

DECIMAL({octal value})

The function converts an octal integer value between −177 777 and +177 777, inclusive, into its decimal representation. If the argument given is not octal, then an error (number 184) results.

This can be used as a quick, simple way of converting octal numbers into the more familiar decimal value. Being a function, it can be used anywhere any other BASIC numeric function can be used. Often you will find it useful in PRINT statements which are a part of subroutines called by break points.

OCTAL

NOTE

The values resulting from the OCTAL function must be treated with care. Though the result of the function is an octal representation, the value is still base-10. This difference is unimportant unless you are going to do arithmetic with the value resulting from the function.

This function is the converse of the DECIMAL function. Its role is to convert decimal values between −65 535 and +65 535, inclusive, into their octal (base-8) representation. The function has the form —

OCTAL({decimal value})

This can be used as a quick, convenient method of converting decimal numbers into their frequently used octal representations (a form which is useful because of its ready conversion into binary representation, and vice-versa).

As an example of this, suppose the decimal value 15 is to be converted into octal. The method is —

```
OCTAL(15)
```

and the resultant value is 17, the octal representation of 15. Now, if the result has 1 added to it, as with the expression —

```
OCTAL(15)+1
```

the ultimate result is 18. This can be a surprise since the usual octal arithmetic suggests that the result of 17B + 1 be 20B. To get the proper octal result, the procedure is —

The correct result can also be obtained with —

```
OCTAL(DECIMAL(17)+1)
```

or

```
OCTAL(DECIMAL(17) + DECIMAL(1) )
```

The preceding are examples of octal addition. Suppose you wanted the result of 17B + 14B. The expression used to obtain the correct result in octal representation is —

```
OCTAL(DECIMAL(17) + DECIMAL(14) )
```

The correct result is 33B.

IADR

This function yields the numeric value in octal representation of the address of an assembled location. The form is —

IADR({assembled location})[1]

As an example, take the case of the example program at the beginning of this chapter. The result of —

```
IADR(Loop,4)
```

is 76.

This function can be viewed as a convenient method of determining the address of a symbol, or of an offset from a symbol.

IMEM

This function is a quick, convenient way to look at the contents of a specific location in memory. The result is a numeric value, in octal representation, for the contents of a specified address. The form is —

IMEM ({assembled location})[1]

1 For an explanation of {assembled location}, see the "Symbolic Debugging" section of this chapter.

The function is similar in many respects to the IDUMP statement. It is easiest, perhaps, to list the differences —

- IMEM is a function, where IDUMP is a statement.
- IMEM deals only with a single address, where IDUMP can deal with many.
- IMEM represents the value only in octal, where IDUMP can use many different representations.
- IMEM can be displayed and stored, where IDUMP can only be printed.

An obvious use for this function is in a routine called by an IBREAK statement. By using the function in such a manner, perhaps in a PRINT statement, you can ease the burden of checking variables from the keyboard. You can even use the value returned as a comparison against some set of limits so that you print only when the value exceeds those limits. There are many other possibilities for its use.

Interrogating Registers and Flags

Interrogating the processor register A, B, P, R, Pa, Cb, Db, Dmapa, Dmama, Dmac, C, D, Ar2, Se, and Ar1 yields meaningful results only when execution of an assembly language subprogram has been suspended due to detection of a break point, or due to the use of the STEP or PAUSE keys (see Stepping Through Programs).

Further, the values of cetain processor flags are stored in specific memory locations when a subprogram is suspended as described above. The flags are then available for interrogation as follows:

Decimal Carry	least significant bit of location 30B
Overflow	least significant bit of location 31B
Extend	most significant bit of location 31B

It is important to note that interrogating an I/O register (R4, R5, R6, or R7) causes an input I/O bus cycle, using the current Pa register contents as the interface address. See Chapter 7 for details on the effects of such an action.

Patching

Patching is the practice of changing the contents of memory locations without re-assembling.

Patching as a standard procedure does not come highly recommended in the programming world. Nonetheless, there are circumstances which arise that occasionally suggest patching as the most profitable course of action.

To change a particular location in memory in the 9845 is not difficult. The statement to use is —

ICHANGE {assembled location} TO {octal expression}

After execution of the statement, the specified {assembled location} contains the specified octal value.

Changing the contents of a register is a common use of this facility. However, it should be remembered that attempting to change the contents of the I/O registers (R4, R5, R6, or R7) causes an output I/O bus cycle to occur, using the Pa register for the interface address. See Chapter 7 for details on the effects of such an action.

Some precautions should be taken in attempting to change the DMA registers. The contents of Dmapa are set by the Isr_access utility and should not be changed while stepping. The contents of Cb and Db (contained in register 13 along with Dmapa) can be changed at any time. The contents of Dmac and Dmama can be changed but be sure that your DMA routine has DMA access at the time of the change. Changing the contents of these registers at a time when another routine has DMA access can have disastrous results.

Stepping vs. Running

You should be made aware at this point of some conditions that exist during stepping that do not exist during a free run of a program. During stepping with the STEP key or when an IBREAK DATA statement is in effect, an assembly language program is not allowed to access (jump to or write into) certain portions of memory. These portions of memory are known as "protected memory" and error 187 results if an attempt is made to access them.

All memory is protected except —

- The ICOM region.
- BASIC's "value" area (the region where BASIC variables are stored).
- BASIC's common area (the region where BASIC common variables are stored).
- The processor registers.
- The temporary values stored in the base page (pre-defined symbol "Base_page").
- The utilities.

Protected memory exists only when you are stepping a program, when an IBREAK DATA statement is in effect, or when you are using the ICHANGE statement. This feature reduces the danger of inadvertent destruction of data or nonsensical execution of data by the processor. Keep in mind that this feature does not exist when the program is free running.

Since the contents of the processor registers are stored in read/write memory, a full 16 bits is used to represent the contents of each register, regardless of whether the register is a four-bit register (Pa,Dmapa,Se) or not. Only the least significant four bits are of interest when an IDUMP statement is used to interrogate the four-bit registers.

The second major difference between stepping and free running is that the processor registers displayed by an IDUMP statement are, in actuality, read/write memory locations. These memory locations are updated only when the program is stopped. Therefore, running a program that changes the contents of the processor registers does not appear to have changed them when the IDUMP statement is used.

In addition, a breakpoint cannot be set for a location within an interrupt service routine. An interrupt service routine cannot be stepped. Attempts to perform either function will lock up the computer.

Chapter 9
Errors and Error Processing

Summary: This chapter contains a discussion of Assembly Language ROM and other related errors, and what causes them. Included are methods for trapping errors and possible methods for correcting them.

Whether you are writing or accessing an assembly language routine, it is possible to encounter an error resulting from your actions. The intent of this chapter is to give some guidance as to how certain errors can be handled. It is not intended as a definitive checklist of what can go wrong, nor is it an exhaustive treatment of the means to correct the difficulties which are listed. Rather, it is meant as a reference for some of the things which can go wrong, what might cause them, and how to deal with them. Each programmer has a unique method of approaching the problem of error processing and there is no way to anticipate all of them. Even so, the following should offer some assistance in identifying the source of an error.

Not every machine error is covered here — only those directly related to writing or accessing assembly language routines. A complete listing of error messages (though not in the same detail as in this chapter) can be found in Appendix J.

Error numbers 900 through 999 are reserved for your own use (with the Error_exit utility).

Types of Errors

There are three types of errors associated with assembly language routines: those which occur during the writing (or entering) of the source code (called "syntax-time" errors); those which occur while assembling the source code (called "assembly-time" errors); and those which occur during the execution of an assembly language routine (called "run-time" errors). Some of these errors can be anticipated and trapped, others cannot.

Syntax-Time and Assembly-Time Errors

Syntax errors are caught when entering source code, usually with the message —

```
IMPROPER ISOURCE STATEMENT
```

The error can then be immediately corrected and the statement reentered. A side-effect of this entry-time check of the syntax is that the time required for assembly is greatly shortened over what it would be if syntax-checking were deferred until assembly.

Errors encountered during the assembly process are indicated by the assembler in three ways:

- The message —

  `ERROR 192 IN LINE` **nn**

  is displayed. **nn** is the line number of the IASSEMBLE statement. This is a fatal BASIC error, unless otherwise trapped.

- Each line in the source code containing an assembly error is printed on the current system printer. Included is the message —

  ```
  **ERROR**
  ```

  followed by the error type.

- The message —

  ```
  ERRORS IN ASSEMBLY
  ```

  follows the listing of the individual errors. The total number of errors is also printed.

An explanation of the individual assembly-time errors can be found at the end of this chapter.

Run-Time Errors

Run-time errors can sometimes be anticipated. They come at two distinct times, and your error processing is different depending upon which of those times are of concern. The times are "program development" and "production run".

During program development, errors normally are handled using the debugging techniques detailed in Chapter 8. Care should be taken in recognizing errors during development. Not all of them are obvious or indicated by an error message — many simply lock up the machine.

During the running of production (debugged) routines, errors can be caused by the users of the routines. For instance, the user may inadvertently assign an argument a value of zero when that argument is to be used as a divisor within the assembly language routine. You should try to anticipate these usage errors and program procedures to trap them.

There are many alternatives for actions to take when your routine encounters and traps a usage error. For example, you may wish to assign a value to a particular return variable, or you may want to print a warning message, or, perhaps, to correct the value and proceed with the routine. Another method is to notify the user by issuing a BASIC error message. Such messages can be issued through the Error_exit utility discussed below.

Of course, you need to tell the users (in the documentation of the routine) what kind of errors can occur, when they can occur, and what to do about them.

UTILITY: Error_exit

The Error_exit utility provides you with the capability of aborting an assembly language routine by "creating" a BASIC error. Two types of BASIC errors can be created — "recoverable", which can be trapped by a BASIC ON ERROR statement; and "non-recoverable" (or "fatal"), which cannot be trapped.

General Procedure: The utility is given the number of the error to be created. Then the utility is called with the JSM instruction, but no return is made to the original assembly language routine from the utility. Instead, the utility uses the information placed on the return stack to help create the error. The return stack is appropriately "cleaned up" and control is returned either to the BASIC driver (if the error is non-fatal) or to the operating system (if the error is fatal).

Special Requirements: Error numbers are passed to the utility in the A register. The value of the error number is placed in bits 0-14. Bit 15 is set if the error is to be non-recoverable. If bit 15 is not set, the error will be recoverable. Error numbers 32 762 through 32 767, with bit 15 set, are reserved by the operating system and should not be used.

If you are setting bit 15 to specify a non-recoverable error, the use of negative numbers should be avoided. For example, loading the A register with −8 does not result in non-recoverable error 8. This is because the error number in bits 0-14 is not 8. A suggested method of setting bit 15 is —

```
LDA = 8+100000B
```

In addition, it is suggested that you limit your error numbers to three digits. The block of error numbers 900 to 999 are reserved for your use in assembly language routines and will not be used in future Hewlett-Packard products.

Calling Procedure:

1. Load the error number into the A register.
2. Call the utility using the JSM instruction.

Exit Conditions: The utility returns control to the BASIC driver which called the routine, appropriately setting conditions so that ERRL, ERRM$, and ERRN work as expected. Also triggers ON ERROR, if applicable.

The utility can be used anywhere in your assembly language, wherever you would like to abort the execution of the current assembly language routine and where you would like to indicate to BASIC what reason (error) caused the abortion.

For example, suppose somewhere in one of your assembly routines you wanted to abort the routine if a certain variable (Flag) is non-zero at a certain point. Suppose also that the variable, when non-zero, contained the error number, then your program could look like —

```
ISOURCE  LDA Flag
ISOURCE  SZA *+2
ISOURCE  JSM Error_exit
```

Similarly, there are some utilities which, when an error is encountered, return an error number in register A. In these cases, a quick two-instruction sequence can give you an error-related abort. For example, the Rel_math utility is such a utility —

```
ISOURCE  JSM Rel_math
ISOURCE  SZA *+2
ISOURCE  JSM Error_exit
```

As an example of a fatal error, suppose the error desired is 8. The error sequence could be —

```
ISOURCE  LDA =100010B
ISOURCE  JSM Error_exit
```

Run-Time Messages

The following is a list of the system error messages you, or the users of your routines, may receive should something go wrong retrieving, using, or storing assembly language routines. A possible corrective action, or actions, is included in the discussion of the error.

ERROR 1 — ROM missing, or configuration error. To operate the 9845, all system ROMs must be in place. In addition, to write assembly programs, the Assembly Execution and the Development ROM must also be installed. Perform the system test if the problem persists.

ERROR 2 — Memory overflow. You may have specified an ICOM which is too large for your current available space. Some things to try: select a smaller ICOM size; execute SCRATCH C (if no important data remain in common), delete modules and reduce the ICOM size; segment your BASIC programs; segment your assembly programs. The error may also be caused by trying to load modules which are too large for the current ICOM region (either collectively or individually) or by placing a COM statement before an ICOM statement.

ERROR 9 — The number of arguments passed by an ICALL statement exceeds the number of parameter declarations in the subroutine entry section. This error is not given if the number of arguments is equal to or fewer than the parameter declarations. The actual number passed is stored in the word reserved by the SUB pseudo-instruction.

ERROR 184 — Improper argument in DECIMAL or OCTAL function. The OCTAL function has a range from − 65535 to + 65535. The DECIMAL function has a range for its arguments of − 177777B to + 177777B. Reference made to an absolute address greater than 177777B or 65 535_{10}.

ERROR 185 — Break Table overflow. A maximum of eight breaks can be established with the IBREAK statements and be in effect at one time. If eight breaks are in effect, then to allow other breaks to be established it is necessary to clear previous breaks using the INORMAL statement.

ERROR 186 — Undefined BASIC label or subprogram name used in IBREAK statement. When the IBREAK statement is executed, an undefined label or name is allowed, but when the break actually occurs, the label or name must exist.

ERROR 187 Attempt to write into protected memory; or, an attempt to execute an instruction not in the ICOM region. This is the result of an attempt to branch outside of permissible areas or to change the contents of memory outside of the permissible areas. There is probably a difficulty in the logic of the program which needs to be corrected. This error occurs when the STEP key is being used, an IBREAK DATA statement is in effect, when using the ICHANGE function or when the IBREAK statement is used to break at a location in a non-existent module or at a location beyond the current ICOM region.

ERROR 188 Label used in an assembled location not found. Symbolic addressing requires that all assembly symbols be resolved by execution time. This error probably results from a misspelling of a label or forgetting to assemble the module containing the label.

ERROR 189 Doubly-defined entry point or routine. A module being assembled (with an IASSEMBLE statement) or loaded from mass storage (with an ILOAD statement) contains a SUB or ENT entry point with the same label as a SUB or ENT entry point within a module already resident within the ICOM region. Check the other routines for the duplicate occurrences.

ERROR 190 Missing ICOM statement. You must include an ICOM statement to create your ICOM region before assembling or loading modules. Program an ICOM statement of adequate size and re-run the program

ERROR 191 Module not found. The module indicated in an ISTORE or IASSEMBLE statement is not currently resident in the ICOM region. Check the module names used in your ISTORE statement to find the one which is missing from memory.

ERROR 192 Errors in assembly. At least one error was encountered while assembling one of the modules in your IASSEMBLE statement.

ERROR 193 Attempt to move or delete module containing an active interrupt service routine. This is the result of trying to reduce the size of the ICOM region (or to eliminate it), or trying to delete a module, when one of the affected modules contains an active interrupt service routine (ISR). The only ways to allow the action to take place are to SCRATCH A (which affects a number of other things) or to inactivate the ISR. To inactivate the ISR, consult the routine's documentation, or press Reset (CONTROL STOP).

ERROR 194 IDUMP specification too large. The resulting dump would be more than 32 768 elements.

ERROR 195 Routine specified in ICALL not found. You are specifying the wrong routine name or you are failing to load the correct module. Double check the documentation indicating the location and name of the routine.

ERROR 196 Unsatisfied externals. Symbolic addressing requires that all references to symbols outside the current module be resolved at the time any routine within the current module is executed. This may possibly be a missing ENT instruction within another module.

ERROR 197 Missing COM statement. The routine you are calling is expecting to find or place some of its data in common, but you are not providing the COM statement required. Add the appropriate COM statement in the BASIC program and re-run it.

ERROR 198 BASIC'S common area does not correspond to assembly module requirements. The routine you have called is expecting to find or place some of its data in common, but your COM statement does not match up with the assembly COM declarations in either type or size. Check both the COM statement in the BASIC program and the COM declarations in the assembly routine.

ERROR 199 Insufficient number of BASIC COM items. The routine you are calling is expecting to find or place some of its data in common, but your BASIC COM statement does not provide enough variables to satisfy the routine's needs. Check both the COM statement in the BASIC program and the COM declarations in the assembly routine.

Assembly-Time Messages

The following is a list of the assembler error messages you may receive while assembling a module. All of these errors cause a "fatal" error, which means that the assembly produced no object code. After the error has been corrected, it is necessary to re-assemble the module containing the error. A possible corrective action, or actions, is included in the discussion of the error.

DD — Doubly-defined label. A label can only be defined once in a module. In addition, any label used in an EXT instruction is restricted from being used again as a label in the module. Check all spellings; change a label name to something else, if necessary. Mixing SET and EQU on the same variable may also cause this error to occur.

EN — END statement missing; or module name does not match. The END statement (in an ISOURCE statement) must be included to signify the end of a module. The name in the END statement must match the name used in the immediately preceding NAM statement. Particular ones to look out for: assembling more than one module at a time, but leaving out the END instruction between modules; or, the END statement is not in the same BASIC environment as the NAM statement.

EX — Expression evaluation error. This is a result of a mismatch of element types in the operand of an instruction. The particular prohibited forms are: relocatable + relocatable; external ± external; using the relocatable or external forms with the * or / operators. Check the spelling and type of your symbols in the expression.

LT — Literal pools full or out of range. You may have exhausted the storage given in your literal pool (LIT) declarations. In this case you should add more LIT declarations or increase the size of the ones you have. Another cause of the error can be using a literal in an instruction and there is no literal pool within 512 words of the instruction. Additionally, for some instructions, the assembler attempts to create an indirect reference automatically and requires a literal pool within 512 words of the instruction. In either case, add another literal pool (using a LIT instruction) within range.

MO ICOM region memory overflow. The current module being assembled has caused object code generation which exceeds the current memory allowance for the ICOM region. Either you must re-run the current **main BASIC program** with a new ICOM statement increasing the ICOM size, or you must rearrange your assembly so that the module fits. This latter course can include deleting other modules or rewriting the abortive module so that it requires less memory.

RN Operand out of range. Some instructions using indirection require a relocatable expression to evaluate to an address within 512 words of the current address. Skips must be no more than 32 words in either direction. The EXE instruction requires a register (0 to 31) and the instructions in the Stack Group require registers in the range of 0 to 7. Check to see that the operand used is within the range appropriate for the instruction. Also, check the spelling on all symbols to see that the right symbol was used.

SQ Parameter declaration pseudo-instruction out of sequence. The ANY, FIL, INT, REL, SHO, and STR pseudo-instructions must follow a SUB or COM pseudo-instruction, or be a part of a group of such pseudo-instructions which follow a SUB or COM pseudo-instruction. Any other appearance of these can cause this error. It can also be caused if a SUB sequence does not terminate with a machine instruction with a label. Check to see that you have not inadvertently omitted the SUB or COM, or have placed another instruction in between the pseudo-instruction and its SUB or COM.

TP Incorrect type of operand used. Each instruction requires that its operand be of a certain type — relocatable or absolute. Check the type of all symbols used in the expression in the operand and see that they correspond to the type required by the instruction. If you are using a constant, check to see that a constant is allowed by the instruction.

UN Undefined symbol. By the end of the assembly, all symbols must have been defined, either by use as a label on an instruction or as a symbol associated with a value through an EQU, EXT, or SET pseudo-instruction. A symbol not so defined (except those pre-defined by the assembler) and used in the assembly, causes this error. Check the spelling of all undefined symbols to make sure that you did not intend something else. The symbol otherwise has to be defined, either by label or EQU, EXT, or SET.

Chapter 10 Graphics

Summary

The graphics topics described in this chapter include displaying the graphics raster by setting individual pixels, reading and writing full words, the cursor operations, and line drawing.

Introduction

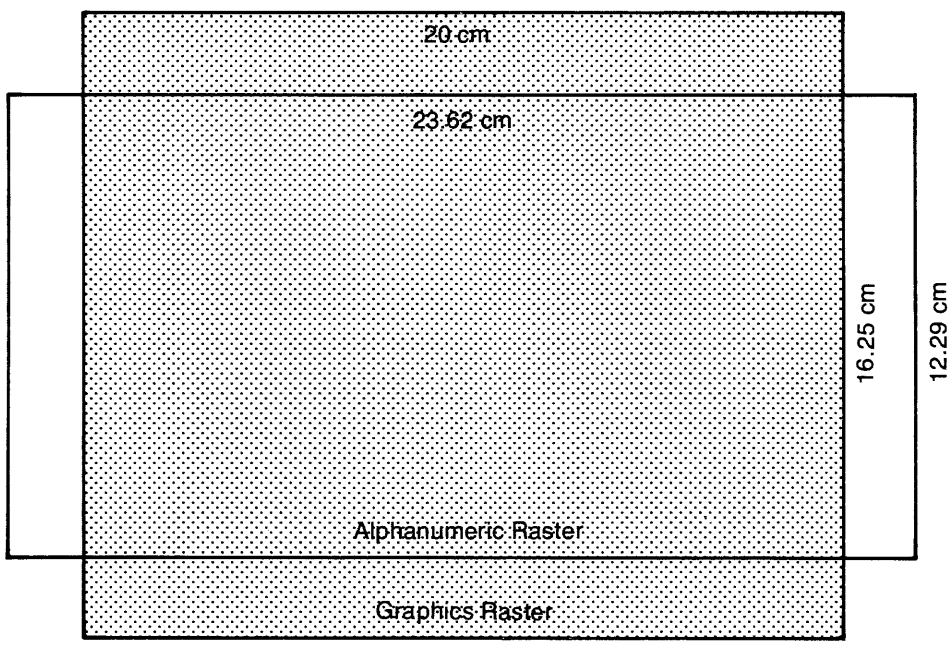
Computer graphics is the computer-aided creation and manipulation of images. These images typically appear on the screen of a CRT or are drawn by a plotter. This chapter explains the fundamental commands and techniques used to create images on the CRT of the System 45 using assembly language. Of course, your System 45 must have the graphics option installed in order for graphics to be implemented.

The advantage of using assembly language rather than BASIC to create and manipulate images on the CRT is one of speed. Graphical data can be manipulated, and input information can be plotted in real time using assembly language in many cases where BASIC could not be used.

The CRT graphics is thought of as being a peripheral on select code 13. Displaying graphics images from assembly language is essentially an I/O operation to that select code.

The Graphics Raster

The CRT of the System 45 computer is capable of displaying two independent rasters (display areas). These are the alphanumeric raster and the graphics raster (when the graphics hardware is installed). When the computer is turned on, the alphanumeric raster is displayed. This is the raster used to display alphanumeric characters when entering programs, displaying program results, etc. With a single command (GRAPHICS) from BASIC or a short sequence of instructions from assembly, the graphics raster is displayed. Both rasters cannot be displayed simultaneously. The alphanumeric and graphics rasters are illustrated below –



Displaying the Graphics Raster

The graphics raster is displayed from your programs by one of two methods. The first involves executing the GRAPHICS command from BASIC. The graphics mode is exited and the alphanumeric raster is displayed with the EXIT GRAPHICS command.

The second method involves executing a short sequence of assembly language instructions. The sequences used to enter and exit graphics from assembly are —

```
10      ISOURCE Graph_on:   LDB 35B
20      ISOURCE             LDA =1       ! This routine turns GRAPHICS on.
30      ISOURCE             STA 35B
40      ISOURCE             LDA (=70000B),I
50      ISOURCE             SAP *+1,C    ! If bit 15 set, clear it.
60      ISOURCE             STA (=70000B),I
70      ISOURCE             STB 35B
80      ISOURCE             RET 1
90      ISOURCE              !
100     ISOURCE Alpha_on:   LDB 35B
110     ISOURCE             LDA =1       ! This routine turns GRAPHICS off.
120     ISOURCE             STA 35B
130     ISOURCE             LDA (=70000B),I
140     ISOURCE             SAP *+1,S    ! If bit 15 is clear, set it.
150     ISOURCE             STA (=70000B),I
160     ISOURCE             STB 35B
170     ISOURCE             RET 1
```

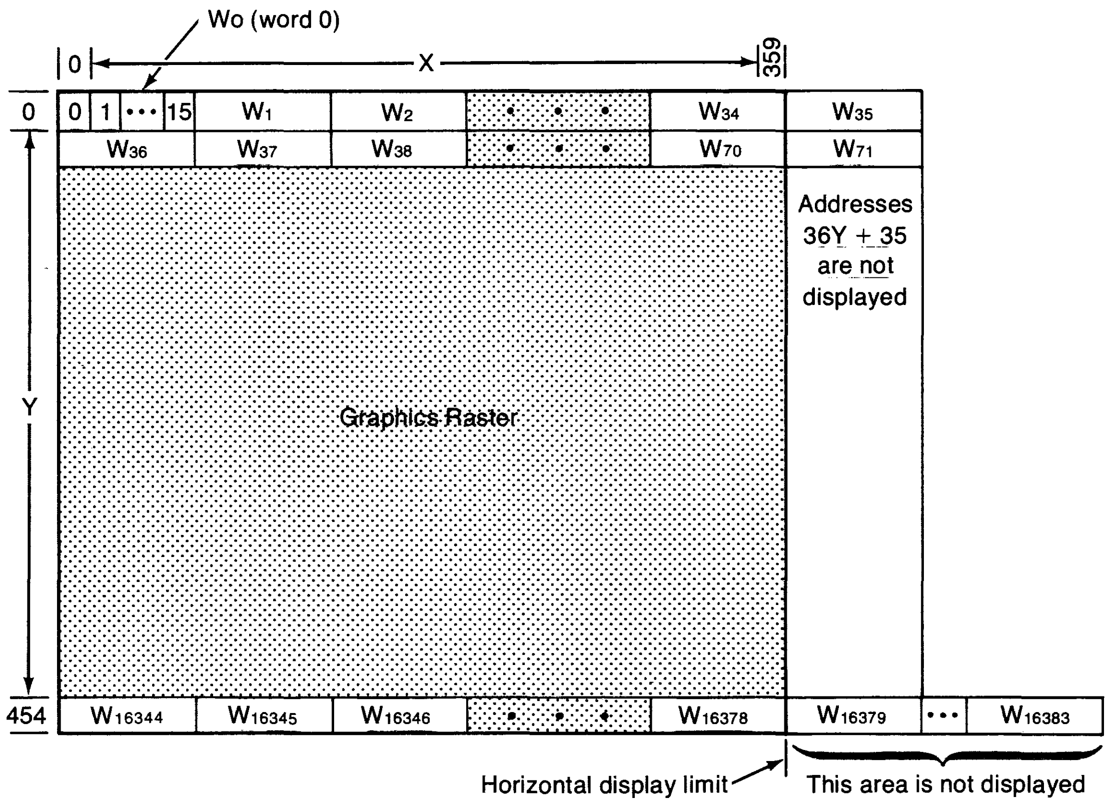
Note that clearing bit 15 of word 70000B causes the graphics raster to be displayed and setting this bit displays the alphanumeric raster. When the computer is turned on, bit 15 is automatically set. It is imperative that the instructions referencing register 35 appear in the raster control program segment. Failure to include these instructions will lock up the computer.

The Graphics Memory

The graphics raster is subdivided into 254 800 individually addressable dots or pixels. The raster is 560 pixels wide and 455 pixels high. Pixels are specified by their X (horizontal, 0-559 and Y (vertical, 0-454) coordinates. Each pixel can be turned on or off, producing the graphics image. This on/off information for each pixel is stored (one bit per pixel) in a separate memory known as the graphics memory.

The graphics memory consists of 16 384 16-bit words of read/write memory. Each bit of the graphics memory determines the on/off status of an individual pixel. This memory contains information even when the graphics raster is not displayed.

The graphics memory is mapped to the graphics raster in the manner represented by the following illustration:



Graphics Memory Map

Each pixel has a word address and a bit address associated with it for communication purposes. For example, word 0, bit 0 holds the on/off information for the pixel in the upper left corner of the raster and word 16 378, bit 15, is mapped to the pixel in the lower right corner. As the illustration indicates, word addresses represented by 36Y + 35, and 16 379 through 16 383 are not displayed.

The X and Y coordinates of an individual pixel are translated into word and bit addresses with the following formulas:

word address = (36*Y)+INT(X/16)
bit address = X MOD 16

The origin, point (0,0), can be moved to the lower left corner of the raster by simply subtracting the Y (vertical) coordinate from 454. This is done in some of the examples for consistency with BASIC commands involving X,Y coordinates.

Graphics Operations

Checking for Graphics Hardware

To test that the graphics hardware is present, execute the following statements —

```
LDA =13
STA Pa
LDA R5
```

The graphics hardware is **not** present if R5 = 0.

Overview

There are several different operations which the graphics hardware can perform. However, each operation is accomplished by issuing a command and then transferring data to or from select code 13. This section discusses the general procedures used to carry out these operations. Details necessary for each operation (such as command and data encoding) are discussed in later sections.

The following graphics operations are available:

- Writing individual pixels
- Writing full words
- Clearing full words
- Reading full words
- Cursor operations

Each graphics operation has a unique control code associated with it that is stored in register 5 with a STA R5 instruction. The control register is represented here —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
unused								INT	DMA	RST	AH	opcode			

where:

INT = interrupt enable bit

DMA = DMA enable bit

RST = reset bit (always sent with a new control code)

AH = auto-handshake bit(for DMA operations)

Since each graphics operation can be carried out by handshake, interrupt or DMA, there are many combinations of control codes.

The general algorithm for each operation includes the following steps —

1. Verify that the graphics hardware is present and operational.
2. If interrupts or DMA are to be used, call Isr_access to obtain the necessary access.
3. Wait for the graphics hardware to become ready.
4. Store the control code identifying the operation to be performed and any interrupt or DMA enable information into R5 of select code 13.
5. The data necessary for the operation is sent to or received from select code 13.
6. If another operation is to be performed, continue with Step 3.
7. If interrupts or DMA are used, access must be released.

In general, the data transfer (Step 5) can be made using programmed I/O or DMA methods. However, interrupt is not recommended where speed is a consideration, and for some operations, only programmed I/O is recommended. When choosing between programmed I/O and DMA, keep the following in mind —

- Programmed I/O is easier to implement but may or may not generate the faster throughput.
- There is only one DMA channel. The rules of access to the DMA channel prevent attempts by two I/O tasks which need the DMA channel (your graphics task and a disc or I/O ROM operation, for example) from occurring simultaneously. In addition, DMA activity cannot occur at the same time as a synchronous I/O task (such as writing to or reading from a tape cartridge).
- The maximum data transfer rate to or from the graphics hardware using DMA is twice that of programmed I/O.
- When using DMA, the Isr_access utility must be called before using the DMA channel. In addition, all data to be transferred to the graphics memory must be in contiguous memory locations within the ICOM region (i.e. a buffer area). Thus the overhead encountered in starting a DMA transfer is higher than that involved in starting a programmed I/O transfer.
- Several transfers may be initiated as a result of a single ICALL. In this case, the Isr_access utility would be called only once and the resulting overhead distributed over all the transfers.

Generally speaking, then, if ease of implementation is a major concern or if the data transfers are short and not numerous, then programmed I/O is the preferred technique. If there are many transfers or they are long, the additional overhead of using DMA will be overcome by the faster transfer rate, resulting in higher throughput.

Operation: Writing Individual Pixels

Individual bits within the graphics memory can be set or cleared using the "write pixels" command. This capability might be used, for example, within a line drawing subroutine to turn on a sequence of pixels.

General Procedure:

- A "write pixels" command is stored in R5.
- A data transfer is started to send word address, bit address, and new value for each bit to be changed.

Special Considerations:

- The control code for the "write pixel" command is as follows —

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">—</td><td>INT</td><td>DMA</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
</table>

where:

INT = interrupt enabled bit
DMA = DMA enabled bit
— = don't care

- The data must be in a special format consisting of two words per bit to be changed. This is represented in the following illustration.

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>First Word:</td><td>—</td><td>—</td><td colspan="14">CWA</td></tr>
<tr><td>Second word:</td><td>D</td><td colspan="11">—</td><td colspan="4">BA</td></tr>
</table>

where:

CWA = complemented word address
BA = bit address
D = data value (1=ON, 0=OFF)
— = don't care

Thus for each pixel to be set or cleared, two words must be transferred to select code 13.

- Either DMA or programmed I/O can be used.

Writing Pixels Using Programmed I/O

```
10 ! GRAPHICS, WRITING INDIVIDUAL PIXELS USING PROGRAMMED I/O.
20    ICOM 200
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    INTEGER X,Y,On,Off
70    IASSEMBLE Pixel_on_off
80    !
90    On=1
100   Off=0
110   !
120   FOR X=100 TO 200
130      Y=X
140      ICALL Write_pixel_pio(X,Y,On)
150   NEXT X
160   !
170   FOR X=100 TO 200
180      Y=X
190      ICALL Write_pixel_pio(X,Y,Off)
200   NEXT X
201   !
210   GOTO 120
220   !
230      ISOURCE                 NAM Pixel_on_off     ! Module name
240      ISOURCE                 EXT Get_value        ! Declare externals
250      ISOURCE X_coord:        BSS 1                ! Storage for X
260      ISOURCE Y_coord:        BSS 1                ! Storage for Y
270      ISOURCE Bit:            BSS 1                ! Storage for BIT status
280      ISOURCE                 SUB
290      ISOURCE X_parm:         INT
300      ISOURCE Y_parm:         INT
310      ISOURCE Bit_parm:       INT
320      ISOURCE Write_pixel_pio: LDA =X_coord        ! Get X coordinate
330      ISOURCE                 LDB =X_parm
340      ISOURCE                 JSM Get_value
350      ISOURCE                 LDA =Y_coord         ! Get Y coordinate
360      ISOURCE                 LDB =Y_parm
370      ISOURCE                 JSM Get_value
380      ISOURCE                 LDA =Bit             ! Get BIT status
390      ISOURCE                 LDB =Bit_parm
400      ISOURCE                 JSM Get_value
410      ISOURCE                 LDA =13              ! Put select code in Pa
420      ISOURCE                 STA Pa
430      ISOURCE                 LDA R5               ! Check for GRAPHICS hardware
440      ISOURCE                 SZA No_graphics
450      ISOURCE                 LDA =51B            ! Send WRITE PIXEL control code
460      ISOURCE                 SFC *
470      ISOURCE                 STA R5
480      ISOURCE                 LDB Y_coord          ! Calculate word address
490      ISOURCE                 LDA =36              ! (36*Y) + INT(X/16)
500      ISOURCE                 MPY
```

```
510     ISOURCE              LDB X_coord
520     ISOURCE              SBR 4
530     ISOURCE              ADA B
540     ISOURCE              CMA                ! Complement address
550     ISOURCE              SFC *              ! Wait for flag
560     ISOURCE              STA R4             ! Send word address
570     ISOURCE              STA R7             ! Trigger output
580     ISOURCE              LDA X_coord        ! Calculate bit address
590     ISOURCE              AND =17B           ! X MOD 16
600     ISOURCE              LDB Bit            ! Get BIT status
610     ISOURCE              SBL 15             ! Shift to bit 15
620     ISOURCE              IOR B              ! Combine bit status & address
630     ISOURCE              SFC *              ! Wait for flag
640     ISOURCE              STA R4             ! Send bit status and address
650     ISOURCE              STA R7             ! Trigger output
660     ISOURCE              RET 1              ! Return to BASIC
670     ISOURCE              !
680     ISOURCE No_graphics:RET 1
690     ISOURCE              END Pixel_on_off   ! Module end
```

Writing Individual Pixels Using DMA

```
10 ! GRAPHICS, WRITING INDIVIDUAL PIXELS USING DMA.
20    ICOM 200
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    IASSEMBLE Write_pixel_dma
70    !
80    ICALL Write_pixel_dma
90    !
100   STOP
110   !
120       ISOURCE              NAM Write_pixel_dma ! Module name
130       ISOURCE              EXT Get_value       ! Declare externals
140       ISOURCE              EXT Isr_access
150       ISOURCE Buffer:      DAT 177003B,100001B ! Data word pairs
160       ISOURCE              DAT 177003B,100002B
170       ISOURCE              DAT 177003B,100003B
180       ISOURCE              DAT 177003B,100004B
190       ISOURCE              DAT 177003B,100005B
200       ISOURCE              DAT 177003B,100006B
210       ISOURCE              DAT 177003B,100007B
220       ISOURCE              DAT 177047B,100004B
230       ISOURCE              DAT 177113B,100004B
240       ISOURCE              DAT 177157B,100004B
250       ISOURCE              DAT 176737B,100004B
260       ISOURCE              DAT 176673B,100004B
270       ISOURCE              DAT 176627B,100004B
280       ISOURCE Count:       DAT *-Buffer-1
290       ISOURCE              SUB
300       ISOURCE Write_pixel_dma: LDA =13         ! Put select code in Pa
310       ISOURCE              STA Pa
320       ISOURCE              LDA R5              ! Check for GRAPHICS hardware
330       ISOURCE              SZA No_graphics
340       ISOURCE Try_again:   LDA =Isr            ! Get DMA Resource
350       ISOURCE              LDB =(64*256)+(2*16)+13
360       ISOURCE              JSM Isr_access
370       ISOURCE              JMP Try_again
380       ISOURCE              LDA Count           ! Set Dmac to Count
390       ISOURCE              STA Dmac
400       ISOURCE              LDA =Buffer         ! Set Dmama to BUFFER address
410       ISOURCE              STA Dmama
420       ISOURCE              SDO                 ! Set DMA direction to OUT
430       ISOURCE              DMA                 ! Notify DMA hardware
440       ISOURCE              LDA =51B+300B       ! Send DMA WRITE pixel Command
450       ISOURCE                                  ! NOTE: Bit 5 is ONE
460       ISOURCE              SFC *               ! Wait for flag
470       ISOURCE              STA R5
480       ISOURCE              RET 1               ! Return to BASIC
490       ISOURCE              !
500       ISOURCE No_graphics:RET 1
510       ISOURCE              !
520       ISOURCE Isr:         LDA =0              ! End of transfer interrupt
530       ISOURCE              STA R5              ! Clear control register
540       ISOURCE              JMP End_isr_high,I  ! Release DMA access
550       ISOURCE              !
560       ISOURCE              END Write_pixel_dma ! Module end
```

Operation: Writing Full Words

The "write words" command is recommended when all bits within a graphics memory word are to be changed, and especially when several contiguous words in the memory are to be changed.

General Procedure:

- A "write words" command is stored in R5.
- A data transfer is started to send data to the graphics hardware. The first word sent indicates the starting address within the graphics memory and subsequent words are stored into the graphics memory at sequentially increasing addresses.

Special Considerations:

- The control code for the "write words" command is as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—	—	—	—	—	—	—	—	INT	DMA	1	0	1	0	0	0

where:

INT = interrupt enabled bit
DMA = DMA enabled bit
— = don't care

- The data sent to the graphics hardware must be in the format illustrated here —

CWA
data
data
data
•
•
•

where:

CWA = complemented word address
data = the data to be written into the graphics memory
(Note that the most significant bit of each data word represents the leftmost bit (bit 0) within the graphics memory.)

- Recall that while there are only 35 words of graphics memory data displayed in each row of the CRT raster, there are actually 36 words in the memory for each row. (One word is never displayed.) When using the "write words" command to write data into the last words of one row and the first words of the next row, you must remember to supply data for the "extra" word.
- Either DMA or programmed I/O can be used.

Writing Full Words Using Programmed I/O

```
10 ! GRAPHICS, WRITING FULL WORDS USING PROGRAMMED I/O.
20   ICOM 200
30   IDELETE ALL
40   GCLEAR
50   GRAPHICS
60   INTEGER X,Y,Data
70   IASSEMBLE Write_word_pio
80   !
90   Data=255
100  !
110  FOR X=100 TO 200
120  Y=X
130  ICALL Write_word(X,Y,Data,9)
140  NEXT X
150  !
160  STOP
170  !
180      ISOURCE              NAM Write_word_pio   ! Module name
190      ISOURCE              EXT Get_value        ! Declare externals
200      ISOURCE X_coord:     BSS 1                ! Storage for X
210      ISOURCE Y_coord:     BSS 1                ! Storage for Y
220      ISOURCE Data:        BSS 1                ! Storage for DATA WORD
230      ISOURCE Parm_number:SUB
240      ISOURCE X_parm:      INT
250      ISOURCE Y_parm:      INT
260      ISOURCE Data1_parm:  INT
270      ISOURCE Data2_parm:  INT
280      ISOURCE Write_word:  LDA =X_coord         ! Get X coordinate
290      ISOURCE              LDB =X_parm
300      ISOURCE              JSM Get_value
310      ISOURCE              LDA =Y_coord         ! Get Y coordinate
320      ISOURCE              LDB =Y_parm
330      ISOURCE              JSM Get_value
340      ISOURCE              LDA =Data            ! Get first DATA WORD
350      ISOURCE              LDB =Data1_parm
360      ISOURCE              JSM Get_value
370      ISOURCE              LDA =13              ! Put select code in Pa
380      ISOURCE              STA Pa
381      ISOURCE              LDA R5               ! Check for GRAPHICS hardware
382      ISOURCE              SZA No_graphics
390      ISOURCE              LDA =50B             ! Send WRITE WORD control code
400      ISOURCE              SFC *                ! Wait for flag
410      ISOURCE              STA R5
420      ISOURCE              LDB Y_coord          ! Calculate word address
430      ISOURCE              LDA =36              ! (36*Y) + INT(X/16)
440      ISOURCE              MPY
450      ISOURCE              LDB X_coord
460      ISOURCE              SBR 4
```

```
470      ISOURCE              ADA B
480      ISOURCE              CMA                  ! Complement address
490      ISOURCE              SFC *                ! Wait for flag
500      ISOURCE              STA R4               ! Send word address
510      ISOURCE              STA R7               ! Trigger output
520      ISOURCE              LDA Data             ! Get DATA WORD
530      ISOURCE              SFC *                ! Wait for flag
540      ISOURCE              STA R4               ! Send DATA WORD
550      ISOURCE              STA R7               ! Trigger output
560      ISOURCE              LDA Parm_number      ! Check for second WORD
570      ISOURCE              ADA =-4
580      ISOURCE              SAP Send_second
590      ISOURCE              RET 1                ! Return to BASIC
591      ISOURCE              !
592      ISOURCE No_graphics:RET 1
593      ISOURCE              !
600      ISOURCE Send_second:LDA =Data             ! Get second DATA WORD
610      ISOURCE              LDB =Data2_parm
620      ISOURCE              JSM Get_value
630      ISOURCE              LDA Data
640      ISOURCE              SFC *                ! Wait for flag
650      ISOURCE              STA R4               ! Send DATA WORD
660      ISOURCE              STA R7               ! Trigger output
670      ISOURCE              RET 1                ! Return to BASIC
680      ISOURCE              END Write_word_pio   ! Module end
```

Writing Full Words Using DMA

```
10 ! GRAPHICS, WRITING FULL WORDS USING DMA.
20    ICOM 20000
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    SCALE 0,559,0,454
70    FRAME
80    MOVE 100,454-100
90    DRAW 450,454-450
100   INTEGER X,Y,Count
110   IASSEMBLE Write_word_dma
120   !
130   Count=500
140   FOR X=100 TO 400 STEP 50
150      Y=X
160      ICALL Write_word(X,Y,Count)
170   NEXT X
180   !
190   STOP
200   !
210      ISOURCE                NAM Write_word_dma  ! Module name
220      ISOURCE                EXT Get_value       ! Declare externals
230      ISOURCE                EXT Isr_access
240      ISOURCE X_coord:       BSS 1               ! Storage for X
250      ISOURCE Y_coord:       BSS 1               ! Storage for Y
260      ISOURCE Count:         BSS 1               ! Storage for WORD COUNT
261      ISOURCE Cwa:           BSS 1               ! Complemented WORD ADDRESS
270      ISOURCE Buffer:        BSS 16384           ! FOLLOWED BY Storage for data
280      ISOURCE                SUB
290      ISOURCE X_parm:        INT
300      ISOURCE Y_parm:        INT
310      ISOURCE Count_parm:    INT
320      ISOURCE Write_word:    LDA =X_coord        ! Get X coordinate
330      ISOURCE                LDB =X_parm
340      ISOURCE                JSM Get_value
350      ISOURCE                LDA =Y_coord        ! Get Y coordinate
360      ISOURCE                LDB =Y_parm
370      ISOURCE                JSM Get_value
380      ISOURCE                LDA =Count          ! Get WORD count
390      ISOURCE                LDB =Count_parm
400      ISOURCE                JSM Get_value
410      ISOURCE                LDA =Buffer         ! Initilize BUFFER
420      ISOURCE                LDB =52525B
430      ISOURCE Again:         STB A,I
440      ISOURCE                ADA =1
450      ISOURCE                CPA =Buffer+16384
460      ISOURCE                JMP *+2
470      ISOURCE                JMP Again
480      ISOURCE                LDA =13             ! Put select code in Pa
490      ISOURCE                STA Pa
500      ISOURCE                LDA R5              ! Check for GRAPHICS hardware
510      ISOURCE                RZA Graphics_here
511      ISOURCE                !
512      ISOURCE                RET 1               ! NO GRAPHICS EXIT
513      ISOURCE                !
550      ISOURCE Graphics_here:LDB Y_coord          ! Calculate word address
560      ISOURCE                LDA =36             ! (36*Y) + INT(X/16)
570      ISOURCE                MPY
580      ISOURCE                LDB X_coord
590      ISOURCE                SBR 4
600      ISOURCE                ADA B
```

```
610  ISOURCE              CMA                  ! Complement address
630  ISOURCE              STA Cwa              ! Save word address
650  ISOURCE              LDA Count            ! Check for max WORD count
660  ISOURCE              ADA =-16384
670  ISOURCE              SAM Count_ok
680  ISOURCE              LDA =16384           ! Set count to max allowed
690  ISOURCE              STA Count
700  ISOURCE Try_again:                        !
710  ISOURCE Count_ok:    LDA =Isr             ! Get DMA Resource
720  ISOURCE              LDB =(64*256)+(2*16)+13
730  ISOURCE              JSM Isr_access
740  ISOURCE              JMP Try_again
750  ISOURCE              LDA Count            ! Set Dmac to COUNT
770  ISOURCE              STA Dmac
780  ISOURCE              LDA =Cwa             ! Set Dmama to Complemented
781  ISOURCE                                   ! Address
790  ISOURCE              STA Dmama
800  ISOURCE              SDO                  ! Set DMA direction to OUT
820  ISOURCE              DMA                  ! Notify DMA hardware
830  ISOURCE              LDA =50B+300B        ! Send DMA WRITE WORD Command
840  ISOURCE                                   ! NOTE: Bit 5 is ONE
850  ISOURCE              SFC *                ! Wait for flag
860  ISOURCE              STA R5
870  ISOURCE              RET 1                ! Return to BASIC
880  ISOURCE              !
890  ISOURCE Isr:         LDA =0               ! End of transfer interrupt
900  ISOURCE              STA R5               ! Clear control register
920  ISOURCE              JMP End_isr_high,I   ! Release DMA access
940  ISOURCE              !
950  ISOURCE              END Write_word_dma   ! Module end
```

Operation: Clearing Full Words

Clearing words within the graphics memory can be accomplished using the "write pixels" or the "write words" commands discussed previously. However, if many sequential words are to be cleared, the most efficient way is to use the "clear words" command with DMA. This operation is identical to the "write words" command including the data transfer, except that the data is ignored by the graphics hardware and zeroes are written into the graphics memory.

General Procedure:

- A "clear words" command is stored in R5.

- A data transfer is started to send data to the graphics hardware. The first word sent indicates the starting address within the graphics memory. Each subsequent word transferred causes one word of graphics memory to be cleared.

Special Considerations:

- The control code for the "clear words" command is as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—	—	—	—	—	—	—	—	INT	DMA	1	0	1	0	1	0

INT = interrupt enabled bit
DMA = DMA enabled bit
– = don't care
where:

- The data sent to the graphics hardware must be in the format illustrated here —

CWA
data
data
data
•
•
•

where:
CWA = complemented word address
data = data is ignored

- Recall that while there are only 35 words of graphics memory data displayed in each row of the CRT raster, there are actually 36 words in the memory for each row. (One word is never displayed.) When using the "clear words" command to clear the last words of one row and the first words of the next row, you must remember to allow for the "extra" word.

Clearing Full Words Using DMA

```
10 ! GRAPHICS, CLEARING FULL WORDS USING DMA.
20    ICOM 200
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    SCALE 0,559,0,454
70    FRAME
80    MOVE 100,454-100
90    DRAW 450,454-450
100   INTEGER X,Y,Count
110   IASSEMBLE Clear_word_dma
120   !
130   Count=500
140   FOR X=100 TO 400 STEP 50
150   Y=X
160   ICALL Clear_word(X,Y,Count)
170   NEXT X
180   !
190   STOP
200   !
210       ISOURCE              NAM Clear_word_dma   ! Module name
220       ISOURCE              EXT Get_value        ! Declare externals
230       ISOURCE              EXT Isr_access
240       ISOURCE X_coord:     BSS 1                ! Storage for X
250       ISOURCE Y_coord:     BSS 1                ! Storage for Y
260       ISOURCE Count:       BSS 1                ! Storage for WORD COUNT
270       ISOURCE Cwa:         BSS 1                ! Storage for Complemented
280       ISOURCE                                   ! WORD ADDRESS
290       ISOURCE              SUB
300       ISOURCE X_parm:      INT
310       ISOURCE Y_parm:      INT
320       ISOURCE Count_parm:  INT
330       ISOURCE Clear_word:  LDA =X_coord         ! Get X coordinate
340       ISOURCE              LDB =X_parm
350       ISOURCE              JSM Get_value
360       ISOURCE              LDA =Y_coord         ! Get Y coordinate
370       ISOURCE              LDB =Y_parm
380       ISOURCE              JSM Get_value
390       ISOURCE              LDA =Count           ! Get WORD count
400       ISOURCE              LDB =Count_parm
410       ISOURCE              JSM Get_value
420       ISOURCE              LDA =13              ! Put select code in Pa
430       ISOURCE              STA Pa
440       ISOURCE              LDA R5               ! Check for GRAPHICS hardware
450       ISOURCE              RZA Graphics_here
460       ISOURCE              !
470       ISOURCE              RET 1                ! NO GRAPHICS EXIT
480       ISOURCE              !
490       ISOURCE Graphics_here:LDB Y_coord         ! Calculate word address
500       ISOURCE              LDA =36              ! (36*Y) + INT(X/16)
510       ISOURCE              MPY
520       ISOURCE              LDB X_coord
530       ISOURCE              SBR 4
540       ISOURCE              ADA B
550       ISOURCE              CMA                  ! Complement address
560       ISOURCE              STA Cwa              ! Store starting word address
570       ISOURCE              LDA Count            ! Check for max WORD count
580       ISOURCE              ADA =-16384
590       ISOURCE              SAM Count_ok
600       ISOURCE              LDA =16384           ! Set count to max allowed
610       ISOURCE              STA Count
```

```
620     ISOURCE Try_again:                          !
630     ISOURCE Count_ok:    LDA =Isr               ! Get DMA Resource
640     ISOURCE              LDB =(64*256)+(2*16)+13
650     ISOURCE              JSM Isr_access
660     ISOURCE              JMP Try_again
670     ISOURCE              LDA Count              ! Set Dmac to COUNT-1
680     ISOURCE              ADA =-1
690     ISOURCE              STA Dmac
700     ISOURCE              LDA =Cwa               ! Set Dmama to start address
710     ISOURCE              STA Dmama
720     ISOURCE              SDO                    ! Set DMA direction to OUT
730     ISOURCE              DMA                    ! Notify DMA hardware
740     ISOURCE              LDA =52B+300B          ! Send DMA CLEAR WORD Command
750     ISOURCE                                     ! NOTE: Bit 5 is one
760     ISOURCE              SFC *                  ! Wait for flag
770     ISOURCE              STA R5
780     ISOURCE              RET 1                  ! Return to BASIC
790     ISOURCE              !
800     ISOURCE Isr:         LDA =0                 ! End of transfer interrupt
810     ISOURCE              STA R5                 ! Clear control register
820     ISOURCE              JMP End_isr_high,I     ! Release DMA access
830     ISOURCE              !
840     ISOURCE              END Clear_word_dma     ! Module end
```

Operation: Reading Full Words

The data in the graphics memory can be retrieved using the "read words" command. This is the only way data can be retrieved since there is no "read pixels" command. This capability might be used to store graphic images on mass memory or to update the graphic image using a read-modify-write algorithm.

General Procedure:

- A "read words" command is stored in R5.
- A single word is sent to the graphics hardware to indicate the starting address within the graphics memory.
- An input data transfer retrieves consecutive words from the graphics memory starting at the specified address.

Special Considerations:

- The control code for the "read words" command is as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—	—	—	—	—	—	—	—	INT	DMA	1	0	1	1	0	0

where:

INT = interrupt enabled bit

DMA = DMA enabled bit

– = don't care

- The data sent to the graphics hardware must be in the format illustrated here —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CWA															

where:

CWA = complemented word address

- Recall that while there are only 35 words of graphics memory data displayed in each row of the CRT raster, there are actually 36 words in the memory for each row. (One word is never displayed.) When using the "read words" command to read data from the last words of one row and the first words of the next row, you must remember to allow for the "extra" word.

- Either DMA or programmed I/O can be used.

Reading Full Words Using Programmed I/O

```
10 ! GRAPHICS, READING FULL WORDS USING PROGRAMMED I/O.
20    ICOM 200
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    SCALE 0,559,0,454
70    MOVE 100,455-100
80    DRAW 200,455-200
90    INTEGER X,Y,Data
100   IASSEMBLE Read_word_pio
110   !
120   WAIT 1000
130   EXIT GRAPHICS
140   !
150   FOR X=100 TO 200
160   Y=X
170   ICALL Read_word(X,Y,Data)
180   PRINT Data,OCTAL(Data)
190   NEXT X
200   !
210   STOP
220   !
230       ISOURCE               NAM Read_word_pio   ! Module name
240       ISOURCE               EXT Get_value       ! Declare externals
250       ISOURCE               EXT Put_value
260       ISOURCE X_coord:      BSS 1               ! Storage for X
270       ISOURCE Y_coord:      BSS 1               ! Storage for Y
280       ISOURCE Data:         BSS 1               ! Storage for DATA WORD
290       ISOURCE               SUB
300       ISOURCE X_parm:       INT
310       ISOURCE Y_parm:       INT
320       ISOURCE Data_parm:    INT
330       ISOURCE Read_word:    LDA =X_coord        ! Get X coordinate
340       ISOURCE               LDB =X_parm
350       ISOURCE               JSM Get_value
360       ISOURCE               LDA =Y_coord        ! Get Y coordinate
370       ISOURCE               LDB =Y_parm
```

```
380       ISOURCE               JSM Get_value
390       ISOURCE               LDA =13              ! Put select code in Pa
400       ISOURCE               STA Pa
410       ISOURCE               LDA R5               ! Check for GRAPHICS hardware
420       ISOURCE               SZA No_graphics
430       ISOURCE               LDA =54B             ! Send READ WORD control code
440       ISOURCE               SFC *                ! Wait for flag
450       ISOURCE               STA R5
460       ISOURCE               LDB Y_coord          ! Calculate word address
470       ISOURCE               LDA =36              ! (36*Y) + INT(X/16)
480       ISOURCE               MPY
490       ISOURCE               LDB X_coord
500       ISOURCE               SBR 4
510       ISOURCE               ADA B
520       ISOURCE               CMA                  ! Complement address
530       ISOURCE               SFC *                ! Wait for flag
540       ISOURCE               STA R4               ! Send word address
550       ISOURCE               STA R7               ! Trigger output
560       ISOURCE               SFC *                ! Wait for flag
570       ISOURCE               LDA R4               ! Get DATA WORD
580       ISOURCE               STA Data
590       ISOURCE               LDA =Data            ! Send DATA WORD to BASIC
600       ISOURCE               LDB =Data_parm
610       ISOURCE               JSM Put_value
620       ISOURCE               RET 1                ! Return to BASIC
630       ISOURCE               !
640       ISOURCE No_graphics:RET 1
650       ISOURCE               !
660       ISOURCE               END Read_word_pic    ! Module end
```

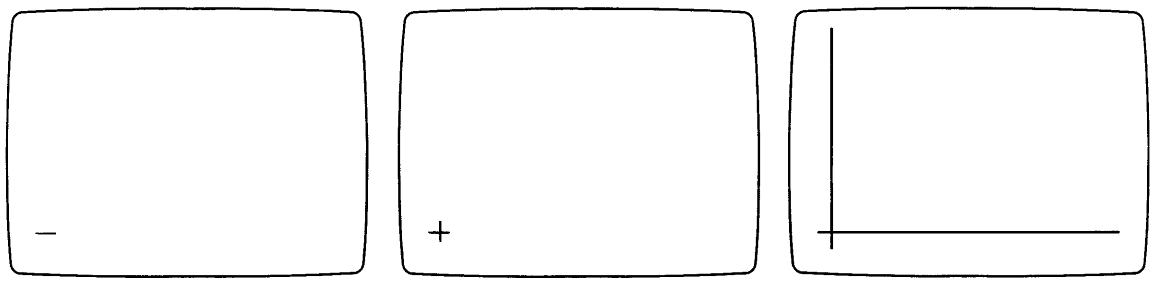
Reading Full Words Using DMA

```
10 ! GRAPHICS, READING FULL WORDS USING DMA.
20    ICOM 20000
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    SCALE 0,559,454,0   ! (0,0) IS AT UPPER LEFT & (559,454) IS AT LOWER RIGHT
70    FRAME
80    MOVE 100,100
90    DRAW 110,110
100   INTEGER X,Y,Count
110   IASSEMBLE Read_word_dma
120   !
130   WAIT 1000
140   EXIT GRAPHICS
150   Count=100
160   FOR X=100 TO 110
170      Y=X
180      ICALL Read_word(X,Y,Count)
190      IDUMP Buffer TO Buffer,39
200      PRINT
210   NEXT X
220   !
230   STOP
240   !
250       ISOURCE               NAM Read_word_dma    ! Module name
260       ISOURCE               EXT Get_value        ! Declare externals
270       ISOURCE               EXT Isr_access
280       ISOURCE X_coord:      BSS 1                ! Storage for X
```

```
290     ISOURCE Y_coord:     BSS 1                ! Storage for Y
300     ISOURCE Count:       BSS 1                ! Storage for WORD COUNT
310     ISOURCE Buffer:      BSS 16384            ! Storage for data
320     ISOURCE              SUB
330     ISOURCE X_parm:      INT
340     ISOURCE Y_parm:      INT
350     ISOURCE Count_parm:  INT
360     ISOURCE Read_word:   LDA =X_coord         ! Get X coordinate
370     ISOURCE              LDB =X_parm
380     ISOURCE              JSM Get_value
390     ISOURCE              LDA =Y_coord         ! Get Y coordinate
400     ISOURCE              LDB =Y_parm
410     ISOURCE              JSM Get_value
420     ISOURCE              LDA =Count           ! Get WORD count
430     ISOURCE              LDB =Count_parm
440     ISOURCE              JSM Get_value
450     ISOURCE              LDA =13              ! Put select code in Pa
460     ISOURCE              STA Pa
470     ISOURCE              LDA R5               ! Check for GRAPHICS hardware
480     ISOURCE              RZA Graphics_here
490     ISOURCE              !
500     ISOURCE              RET 1                ! NO GRAPHICS EXIT
510     ISOURCE              !
520     ISOURCE Graphics_here:LDA =54B            ! Send READ WORD control code
530     ISOURCE              SFC *                ! Wait for flag
540     ISOURCE              STA R5
550     ISOURCE              LDB Y_coord          ! Calculate word address
560     ISOURCE              LDA =36              ! (36*Y) + INT(X/16)
570     ISOURCE              MPY
580     ISOURCE              LDB X_coord
590     ISOURCE              SBR 4
600     ISOURCE              ADA B
610     ISOURCE              CMA                  ! Complement address
620     ISOURCE              SFC *                ! Wait for flag
630     ISOURCE              STA R4               ! Send word address
640     ISOURCE              STA R7               ! Trigger output
650     ISOURCE              LDA Count            ! Check for max WORD count
660     ISOURCE              ADA =-16384
670     ISOURCE              SAM Count_ok
680     ISOURCE              LDA =16384           ! Set count to max allowed
690     ISOURCE              STA Count
700     ISOURCE Try_again:                        !
710     ISOURCE Count_ok:    LDA =Isr             ! Get DMA Resource
720     ISOURCE              LDB =(64*256)+(2*16)+13
730     ISOURCE              JSM Isr_access
740     ISOURCE              JMP Try_again
750     ISOURCE              LDA Count            ! Set Dmac to COUNT-1
760     ISOURCE              ADA =-1
770     ISOURCE              STA Dmac
780     ISOURCE              LDA =Buffer          ! Set Dmama to BUFFER address
790     ISOURCE              STA Dmama
800     ISOURCE              SDI                  ! Set DMA direction to IN
810     ISOURCE              DMA                  ! Notify DMA hardware
820     ISOURCE              LDA =14B+300B        ! Send DMA Read Word Command
830     ISOURCE                                   ! NOTE: Bit 5 is ZERO
840     ISOURCE              SFC *                ! Wait for flag
850     ISOURCE              STA R5
860     ISOURCE              RET 1                ! Return to BASIC
870     ISOURCE              !
880     ISOURCE Isr:         LDA =0               ! End of transfer interrupt
890     ISOURCE              STA R5               ! Clear control register
900     ISOURCE              JMP End_isr_high,I   ! Release DMA access
910     ISOURCE              !
920     ISOURCE              END Read_word_dma    ! Module end
```

Operation: Cursor Operations

Three graphics cursors are provided for your use with the graphics hardware. These are a non-blinking, full-screen, cross-line cursor, a small (9 pixels by 9 pixels), blinking, cross-line cursor, and a horizontal underline, blinking cursor. The three cursors are illustrated here —



horizontal cursor **small blinking cursor** **full-screen cursor**

General Procedure:

- An "X cursor position" command is stored in R5.
- A value indicating the X (or horizontal) position of the cursor is sent to the hardware.
- A "Y cursor position" command is stored in R5 (the command also identifies which cursor appears).
- A value indicating the Y (or vertical) position of the cursor is sent to the hardware.

Special Considerations:

- For most applications, only programmed I/O is used for cursor control. Thus the values stored in R5 should be selected from the following table —

Cursor Type	**Octal Control Code (to R5)**
X cursor position	44
Y position (small blinking)	40
Y position (full-screen)	41
Y position (small horizontal)	42

- The data for the X coordinate must be in a special format as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMX1										–	–	–	–	–	–

where:

CMX1 = one's complement of (X coordinate + 63)

– = don't care

- The data for the Y coordinate must be in a special format as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMY1										–	–	–	–	–	–

where:

CMY1 = one's complement of (Y coordinate + 44)

– = don't care

Setting the Cursor Using Programmed I/O

The following program demonstrates the algorithm for controlling the cursor.

```
10 ! GRAPHICS, SETTING CURSOR USING PROGRAMMED I/O.
20    ICOM 200
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    FRAME
70    INTEGER X,Y,Type
80    IASSEMBLE Cursor_pio
90    !
110   FOR Type=0 TO 2  ! 0 = SMALL, 1 = LARGE, 2 = HORIZONTAL
120      FOR X=0 TO 454
130         Y=X
140         ICALL Cursor(X,Y,Type)
150      NEXT X
160   NEXT Type
170   !
180   STOP
190   !
200       ISOURCE              NAM Cursor_pio       ! Module name
210       ISOURCE              EXT Get_value        ! Declare externals
220       ISOURCE X_coord:     BSS 1                ! Storage for X
230       ISOURCE Y_coord:     BSS 1                ! Storage for Y
240       ISOURCE Type:        BSS 1                ! Storage for TYPE word
250       ISOURCE              SUB
260       ISOURCE X_parm:      INT
270       ISOURCE Y_parm:      INT
280       ISOURCE Type_parm:   INT
290       ISOURCE Cursor:      LDA =X_coord         ! Get X coordinate
300       ISOURCE              LDB =X_parm
310       ISOURCE              JSM Get_value
320       ISOURCE              LDA =Y_coord         ! Get Y coordinate
330       ISOURCE              LDB =Y_parm
340       ISOURCE              JSM Get_value
350       ISOURCE              LDA =13              ! Put select code in Pa
360       ISOURCE              STA Pa
361       ISOURCE              LDA R5               ! Check for GRAPHICS hardware
362       ISOURCE              RZA Graphics_here
363       ISOURCE              !
364       ISOURCE              RET 1                ! NO GRAPHICS EXIT
365       ISOURCE              !
370       ISOURCE Graphics_here:LDA =44B            ! Send CURSOR X LOAD control
380       ISOURCE              SFC *                ! Wait for flag
390       ISOURCE              STA R5
400       ISOURCE              LDA X_coord          ! Get X address
410       ISOURCE              ADA =63              ! Add offset
420       ISOURCE              CMA                  ! Complement and
430       ISOURCE              SAL 6                ! shift X coordinate
440       ISOURCE              SFC *                ! Wait for flag
450       ISOURCE              STA R4               ! Send X address
460       ISOURCE              STA R7               ! Trigger output
470       ISOURCE              LDA =Type            ! Get TYPE code
480       ISOURCE              LDB =Type_parm
490       ISOURCE              JSM Get_value
500       ISOURCE              LDA Type
510       ISOURCE              CPA =0               ! Is it small cursor?
520       ISOURCE               JMP Small           ! Yes
530       ISOURCE              CPA =1               ! Is it large cursor?
540       ISOURCE               JMP Large           ! Yes
```

```
550       ISOURCE              CPA =2              ! Is it horizontal cursor?
560       ISOURCE               JMP Horizontal     ! Yes
570       ISOURCE              RET 1               ! None, return to BASIC
580       ISOURCE Small:       LDA =40B            ! Send SMALL CURSOR Y LOAD
590       ISOURCE Y_load:      SFC *               ! Wait for flag
600       ISOURCE              STA R5
610       ISOURCE              LDA Y_coord         ! Get Y address
620       ISOURCE              ADA =44             ! Add offset
630       ISOURCE              CMA                 ! Complement and
640       ISOURCE              SAL 6               ! shift Y coordinate
650       ISOURCE              SFC *               ! Wait for flag
660       ISOURCE              STA R4              ! Send Y address
670       ISOURCE              STA R7              ! Trigger output
680       ISOURCE              RET 1               ! Return to BASIC
690       ISOURCE Large:       LDA =41B            ! Get LARGE CURSOR Y LOAD
700       ISOURCE              JMP Y_load
710       ISOURCE Horizontal:  LDA =42B            ! Get HORIZONTAL Y LOAD
720       ISOURCE              JMP Y_load
730       ISOURCE              !
740       ISOURCE              END Cursor_pio      ! Module end
```

Comprehensive Example

```
10 ! GRAPHICS, MOVING SYMBOL USING WRITE FULL WORDS WITH DMA.
20    ICOM 20000
30    IDELETE ALL
40    GCLEAR
50    GRAPHICS
60    SCALE 0,559,0,454
70    FRAME
80    MOVE 100,454-100
90    DRAW 450,454-450
100   INTEGER X,Y,Count
110   IASSEMBLE Write_dma
120   !
130   X=RND*559
140   Y=RND*454
150   ICALL Write_symbol(X,Y)
160   GOTO 130
161   !
170   X=100
180   FOR Y=100 TO 400
190      ICALL Write_symbol(X,Y)
200   NEXT Y
210   FOR Y=400 TO 100 STEP -1
220      ICALL Write_symbol(X,Y)
230   NEXT Y
240   GOTO 180
250   !
260   STOP
270   !
280       ISOURCE              NAM Write_dma       ! Module name
290       ISOURCE              EXT Get_value       ! Declare externals
300       ISOURCE              EXT Isr_access
310       ISOURCE X_coord:     BSS 1               ! Storage for X
320       ISOURCE Y_coord:     BSS 1               ! Storage for Y
330       ISOURCE Cwa:         BSS 1
340       ISOURCE Line_1:      DAT 1,0,0           ! FIRST LINE OF SYMBOL
350       ISOURCE Line_2:      DAT 2,0,0
360       ISOURCE Line_3:      DAT 3,100001B,100000B
370       ISOURCE Line_4:      DAT 4,100002B,40000B
```

```
380    ISOURCE Line_5:      DAT 5,100004B,20000B
390    ISOURCE Line_6:      DAT 6,100010B,10000B
400    ISOURCE Line_7:      DAT 7,100020B,4000B
410    ISOURCE Line_8:      DAT 8,100040B,2000B
420    ISOURCE Line_9:      DAT 9,100040B,2000B
430    ISOURCE Line_10:     DAT 10,100020B,4000B
440    ISOURCE Line_11:     DAT 11,100010B,10000B
450    ISOURCE Line_12:     DAT 12,100004B,20000B
460    ISOURCE Line_13:     DAT 13,100002B,40000B
470    ISOURCE Line_14:     DAT 14,100001B,100000B
480    ISOURCE Line_15:     DAT 15,0,0
490    ISOURCE Line_16:     DAT 16,0,0
500    ISOURCE Width:       DAT 2
510    ISOURCE              SUB
520    ISOURCE X_parm:      INT
530    ISOURCE Y_parm:      INT
540    ISOURCE Count_parm:  INT
550    ISOURCE Write_symbol:LDA =X_coord        ! Get X coordinate
560    ISOURCE              LDB =X_parm
570    ISOURCE              JSM Get_value
580    ISOURCE              LDA =Y_coord        ! Get Y coordinate
590    ISOURCE              LDB =Y_parm
600    ISOURCE              JSM Get_value
610    ISOURCE              LDA =13             ! Put select code in Pa
620    ISOURCE              STA Pa
630    ISOURCE              LDA R5              ! Check for GRAPHICS hardware
640    ISOURCE              RZA Graphics_here
650    ISOURCE              !
660    ISOURCE              RET 1               ! NO GRAPHICS EXIT
670    ISOURCE              !
680    ISOURCE Graphics_here:LDB Y_coord        ! Calculate word address
690    ISOURCE              LDA =36             ! (36*Y) + INT(X/16)
700    ISOURCE              MPY
710    ISOURCE              LDB X_coord
720    ISOURCE              SBR 4
730    ISOURCE              ADA B
740    ISOURCE              LDB =-8*36          ! Substract 8 LINES for symbol
750    ISOURCE              ADA B
760    ISOURCE              CMA                 ! Complement address
770    ISOURCE              LDB =Line_1
780    ISOURCE Loop:        STA B,I             ! Store address
790    ISOURCE              CPB =Line_16        ! All addresses stored?
800    ISOURCE               JMP Cont           ! yes
810    ISOURCE              ADB =3              ! Point to next address word
820    ISOURCE              ADA =-36            ! Calculate next address
830    ISOURCE              JMP Loop
840    ISOURCE Try_again:                       !
850    ISOURCE Cont:        LDA =Isr            ! Get DMA Resource
860    ISOURCE              LDB =(64*256)+(2*16)+13
870    ISOURCE              JSM Isr_access
880    ISOURCE              JMP Try_again
890    ISOURCE              LDA Width           ! Set Dmac to COUNT-1
900    ISOURCE              STA Dmac
910    ISOURCE              LDA =Line_1         ! Set Dmama to Complemented
920    ISOURCE                                  ! Address
930    ISOURCE              STA Dmama
940    ISOURCE              SDO                 ! Set DMA direction to OUT
950    ISOURCE              DMA                 ! Notify DMA hardware
960    ISOURCE              IOF
970    ISOURCE Send_cmd:    LDA =50B+300B       ! Send DMA WRITE WORD Command
980    ISOURCE                                  ! NOTE: Bit 5 is ONE
990    ISOURCE              SFC *               ! Wait for flag
1000   ISOURCE              STA R5
```

```
1010      ISOURCE             RET 1                 ! Return to BASIC
1020      ISOURCE             !
1030      ISOURCE Isr:        LDA Dmama             ! End of transfer interrupt
1040      ISOURCE             ADA =0
1050      ISOURCE             CPA =Line_16+3        ! At end of symbol yet
1060      ISOURCE              JMP End_isr_high,I    ! Release DMA access
1070      ISOURCE             STA Dmama
1080      ISOURCE             LDA Width
1090      ISOURCE             STA Dmac
1100      ISOURCE             JMP Send_cmd
1110      ISOURCE             ION
1120      ISOURCE             !
1130      ISOURCE             END Write_dma         ! Module end
```

Line Drawing

Lines drawn on the CRT must be drawn pixel-for-pixel between two points because the System 45 graphics is a raster scan graphics. Line drawing routines are typically implemented in software and called when needed. One such routine is provided for your use on the Demonstration Cartridge.

The Demo Cartridge line drawing routine is contained within a file called "BRAL". To use this routine, simple follow the prompts which are displayed.

A listing of the line drawing routine appears here —

```
10   PRINT "**********************************************************************"
20   PRINT "*       BRESENHAM ALGORITHM FOR LINE TO DOT CONVERSION           *"
30   PRINT "**********************************************************************"
40   ICOM 1000
50   ON KEY #6 GOTO Last
60   PRINT "Press KEY6 to exit"
70   INTEGER X1,Y1,X2,Y2,Lipat
80   IDELETE ALL
90   IASSEMBLE Mod1
100      GCLEAR
110 Begin:  PRINT
120        PRINT "enter the X,Y coordinates of the 2 points, maximum X value is"
130        PRINT "559 and maximum Y value is 454"
140        INPUT X1,Y1,X2,Y2      ! Get coordinates of 2 points line will join
150   PRINT "enter the line pattern type:  eraser= 0, solid= 1"
160   INPUT Lipat                  ! Get line type: solid or erase
170  PRINT "point coordinates: X1=";X1;"Y1=";Y1;"X2=";X2;"Y2=";Y2
180      GRAPHICS
190  ICALL Draw(X1,Y1,X2,Y2,Lipat)    ! Call assembly routine to draw line
200                                    ! between 2 points
210  PAUSE
220   EXIT GRAPHICS
230  GOTO Begin                        ! Repeat drawing lines
240 Last:  END
```

```
250    ISOURCE              NAM Mod1
260    ISOURCE              EXT Get_value
270    ISOURCE X1:          BSS 1      ! First X coordinate
280    ISOURCE Y1:          BSS 1      ! First Y coordinate
290    ISOURCE X2:          BSS 1      ! Second X coordinate
300    ISOURCE Y2:          BSS 1      ! Second Y coordinate
310    ISOURCE Lipat:       BSS 1      ! Line type: 1=solid, 0=erase
320    ISOURCE Da:          BSS 4      ! Da= X2-X1    ->Delta X
330    ISOURCE Dbb:         BSS 4      ! Dbb= Y2-Y1   ->Delta Y
340    ISOURCE Dxy:         BSS 4      ! Dxy= ABS(Delta X)- ABS(Delta Y)
350    ISOURCE I1:          BSS 1      ! Address of X or Y increment or
360                                    ! decrement routine
370    ISOURCE I2:          BSS 1      ! Address of X or Y increment or
380                                    ! decrement routine
390    ISOURCE Nxcnt:       BSS 1      ! Next count
400    ISOURCE Del:         BSS 1      ! Del= (-Da)
410    ISOURCE Gloadc:      DAT 51B    ! Graphics command
420    ISOURCE                         !
430    ISOURCE                         ! octant   Dx   Dy   Dxy
440    ISOURCE Octia:       DAT 1001B !   8      0    0     0
450    ISOURCE              DAT 402B  !   7      0    0     1
460    ISOURCE              DAT 2001B !   1      0    1     0
470    ISOURCE              DAT 404B  !   2      0    1     1
480    ISOURCE              DAT 1003B !   5      1    0     0
490    ISOURCE              DAT 1402B !   6      1    0     1
500    ISOURCE              DAT 2003B !   4      1    1     0
510    ISOURCE              DAT 1404B !   3      1    1     1
520    ISOURCE                        !
530    ISOURCE I12a:        *+1
540    ISOURCE              DAT Incx  ! Address of X increment routine
550    ISOURCE              DAT Incy  ! Address of Y increment routine
560    ISOURCE              DAT Decx  ! Address of X decrement routine
570    ISOURCE              DAT Decy  ! Address of Y decrement routine
580    ISOURCE              SUB
590    ISOURCE T1:          INT
600    ISOURCE T2:          INT
610    ISOURCE T3:          INT
620    ISOURCE T4:          INT
630    ISOURCE T5:          INT
640    ISOURCE Draw:        LDA =X1          ! Get first X coordinate into ICOM
650    ISOURCE              LDB =T1          !
660    ISOURCE              JSM Get_value    !
670    ISOURCE              LDA =Y1          ! Get first Y coordinate into ICOM
680    ISOURCE              LDB =T2          !
690    ISOURCE              JSM Get_value    !
700    ISOURCE              LDA =X2          ! Get second X coordinate into ICOM
710    ISOURCE              LDB =T3          !
720    ISOURCE              JSM Get_value    !
730    ISOURCE              LDA =Y2          ! Get second Y coordinate into ICOM
740    ISOURCE              LDB =T4          !
750    ISOURCE              JSM Get_value    !
760    ISOURCE              LDA =Lipat       ! Get line type into ICOM
770    ISOURCE              LDB =T5          !
780    ISOURCE              JSM Get_value    !
790    ISOURCE              LDA Y1           ! Offset origin to lower left
800    ISOURCE              TCA              ! corner by subracting Y from
810    ISOURCE              ADA =454         ! 454
820    ISOURCE              STA Y1           !
830    ISOURCE              LDA Y2           !
840    ISOURCE              TCA              !
850    ISOURCE              ADA =454         !
860    ISOURCE              STA Y2           !
```

```
870     ISOURCE Ocommand:   LDA =13       ! Send out graphics command
880     ISOURCE             STA Pa        ! to CRT at select code 13
890     ISOURCE             SFC *         !
900     ISOURCE             LDB Gloadc    !
910     ISOURCE             STB R5        !
920     ISOURCE Brham:      LDA X1        ! Calculate X2-X1 and store in Da
930     ISOURCE             TCA           !
940     ISOURCE             ADA X2        !
950     ISOURCE             STA Da        !
960     ISOURCE             SAP *+2
970     ISOURCE             TCA
980     ISOURCE             LDB Y1        ! Calculate Y2-Y1 and store in Dbb
990     ISOURCE             TCB           !
1000    ISOURCE             ADB Y2        !
1010    ISOURCE             STB Dbb       !
1020    ISOURCE             SBM *+2
1030    ISOURCE             TCB           ! B= -ABS(Delta Y)
1040    ISOURCE             ADA B         ! A= ABS(Delta X)- ABS(Delta Y)
1050    ISOURCE             STA Dxy       ! Store in Dxy
1060    ISOURCE             SAM Chs       ! If Dxy< 0 then point is in octant
1070    ISOURCE                           ! 2, 3, 6, or 7
1080    ISOURCE             LDA Da        ! Otherwise point is in octant
1090    ISOURCE                           ! 1, 4, 5, or 8
1100    ISOURCE             LDB Dbb
1110    ISOURCE             JMP Brhm1
1120    ISOURCE Chs:        LDA Da        ! If Dxy< 0 then switch Da and Dbb
1130    ISOURCE             LDB Dbb       !
1140    ISOURCE             STA Dbb       !
1150    ISOURCE             STB Da        !
1160    ISOURCE                           !
1170    ISOURCE Brhm1:      SAR 13        ! Calculate octant using sign:
1180    ISOURCE                           !       (+)= 0 and (-)= 1
1190    ISOURCE                           ! Octant  Dx   Dy   Dxy
1200    ISOURCE             AND =4        !   8     0    0     0
1210    ISOURCE             SBR 14        !   7     0    0     1
1220    ISOURCE             IOR B         !   1     0    1     0
1230    ISOURCE             AND =6        !   2     0    1     1
1240    ISOURCE             LDB Dxy       !   5     1    0     0
1250    ISOURCE             SBR 15        !   6     1    0     1
1260    ISOURCE             IOR B         !   4     1    1     0
1270    ISOURCE             ADA =Octla    !   3     1    1     1
1280    ISOURCE             LDB A,I
1290    ISOURCE             LDA B
1300    ISOURCE             AND =7
1310    ISOURCE             ADA =I12a     ! Calculate I1 and I2:
1320    ISOURCE             LDA A,I       ! address of routines for X and Y
1330    ISOURCE             STA I1        !   decrement or increment
1340    ISOURCE             LDA B         !
1350    ISOURCE             SAR 8         !
1360    ISOURCE             ADA =I12a     !
1370    ISOURCE             LDA A,I       !
1380    ISOURCE             STA I2        !
1390    ISOURCE             LDA Dbb       ! Calculate ABS(Dbb)
1400    ISOURCE             SAP Brhm2
1410    ISOURCE             TCA
1420    ISOURCE             STA Dbb
1430    ISOURCE Brhm2:      LDA Da
1440    ISOURCE             SAP Brhm3
1450    ISOURCE             TCA
1460    ISOURCE             STA Da
1470    ISOURCE Brhm3:      ADA =1
```

```
1480    ISOURCE              STA Nxcnt
1490    ISOURCE              LDA Da
1500    ISOURCE              TCA
1510    ISOURCE              STA Del        ! Del= -Da
1520    ISOURCE              ADA Dbb
1530    ISOURCE              SAL 1
1540    ISOURCE              STA Da         ! Da= 2(Dbb-Da)
1550    ISOURCE              LDA Dbb
1560    ISOURCE              SAL 1
1570    ISOURCE              STA Dbb        ! Dbb=2*Dbb
1580    ISOURCE              LDA Del
1590    ISOURCE Loopa:       ADA Dbb
1600    ISOURCE              STA Del        ! Del= -Da + Dbb
1610    ISOURCE Loopb:       LDB Y1         ! Get word address:
1620    ISOURCE              LDA =36        ! 36*Y1 + X1/16
1630    ISOURCE              MPY            ! complemented
1640    ISOURCE              LDB X1
1650    ISOURCE              SBR 4          !
1660    ISOURCE              ADA B          !
1670    ISOURCE              CMA            !
1680    ISOURCE              JSM Gdata      ! Send out word address
1690    ISOURCE              LDA X1         ! Get bit address:
1700    ISOURCE              AND =17B       ! Lower 4 bits are address and
1710    ISOURCE              LDB Lipat      ! most significant bit is whether
1720    ISOURCE              SBL 15         ! pixel is turned on or not
1730    ISOURCE              IOR B          !
1740    ISOURCE              JSM Gdata      ! Send out bit address
1750    ISOURCE Oldpt:       DSZ Nxcnt      ! Decrement Nxcnt
1760    ISOURCE              JMP *+2
1770    ISOURCE              RET 1          ! Exit if Nxcnt is 0
1780    ISOURCE              JSM I1,I       ! Update X and Y addresses
1790    ISOURCE              LDA Del        !
1800    ISOURCE              SAP *+2        !
1810    ISOURCE              JMP Loopa
1820    ISOURCE              JSM I2,I       !
1830    ISOURCE              ADA Da
1840    ISOURCE              STA Del
1850    ISOURCE              JMP Loopb
1860    ISOURCE Incx:        ISZ X1         ! X address increment routine
1870    ISOURCE              RET 1          !
1880    ISOURCE Incy:        ISZ Y1         ! Y address increment routine
1890    ISOURCE              RET 1          !
1900    ISOURCE Decx:        DSZ X1         ! X address decrement routine
1910    ISOURCE              RET 1          !
1920    ISOURCE              RET 1          !
1930    ISOURCE Decy:        DSZ Y1         ! Y address decrement routine
1940    ISOURCE              RET 1          !
1950    ISOURCE              RET 1          !
1960    ISOURCE Gdata:       SFC *          ! Routine to output data to
1970    ISOURCE              STA R4         ! the CRT using registers
1980    ISOURCE              STA R7         ! R4 and R7
1990    ISOURCE              RET 1          !
2000    ISOURCE              END Mod1
```

Appendix A
ASCII Character Set

ASCII Character Codes

ASCII Char.	EQUIVALENT FORMS			
	Binary	Oct	Hex	Dec
NULL	00000000	000	00	0
SOH	00000001	001	01	1
STX	00000010	002	02	2
ETX	00000011	003	03	3
EOT	00000100	004	04	4
ENQ	00000101	005	05	5
ACK	00000110	006	06	6
BELL	00000111	007	07	7
BS	00001000	010	08	8
HT	00001001	011	09	9
LF	00001010	012	0A	10
VT	00001011	013	0B	11
FF	00001100	014	0C	12
CR	00001101	015	0D	13
SO	00001110	016	0E	14
SI	00001111	017	0F	15
DLE	00010000	020	10	16
DC1	00010001	021	11	17
DC2	00010010	022	12	18
DC3	00010011	023	13	19
DC4	00010100	024	14	20
NAK	00010101	025	15	21
SYNC	00010110	026	16	22
ETB	00010111	027	17	23
CAN	00011000	030	18	24
EM	00011001	031	19	25
SUB	00011010	032	1A	26
ESC	00011011	033	1B	27
FS	00011100	034	1C	28
GS	00011101	035	1D	29
RS	00011110	036	1E	30
US	00011111	037	1F	31

ASCII Char.	EQUIVALENT FORMS			
	Binary	Oct	Hex	Dec
space	00100000	040	20	32
!	00100001	041	21	33
"	00100010	042	22	34
#	00100011	043	23	35
$	00100100	044	24	36
%	00100101	045	25	37
&	00100110	046	26	38
'	00100111	047	27	39
(	00101000	050	28	40
)	00101001	051	29	41
*	00101010	052	2A	42
+	00101011	053	2B	43
,	00101100	054	2C	44
−	00101101	055	2D	45
.	00101110	056	2E	46
/	00101111	057	2F	47
0	00110000	060	30	48
1	00110001	061	31	49
2	00110010	062	32	50
3	00110011	063	33	51
4	00110100	064	34	52
5	00110101	065	35	53
6	00110110	066	36	54
7	00110111	067	37	55
8	00111000	070	38	56
9	00111001	071	39	57
:	00111010	072	3A	58
;	00111011	073	3B	59
<	00111100	074	3C	60
=	00111101	075	3D	61
>	00111110	076	3E	62
?	00111111	077	3F	63

ASCII Char.	EQUIVALENT FORMS			
	Binary	Oct	Hex	Dec
@	01000000	100	40	64
A	01000001	101	41	65
B	01000010	102	42	66
C	01000011	103	43	67
D	01000100	104	44	68
E	01000101	105	45	69
F	01000110	106	46	70
G	01000111	107	47	71
H	01001000	110	48	72
I	01001001	111	49	73
J	01001010	112	4A	74
K	01001011	113	4B	75
L	01001100	114	4C	76
M	01001101	115	4D	77
N	01001110	116	4E	78
O	01001111	117	4F	79
P	01010000	120	50	80
Q	01010001	121	51	81
R	01010010	122	52	82
S	01010011	123	53	83
T	01010100	124	54	84
U	01010101	125	55	85
V	01010110	126	56	86
W	01010111	127	57	87
X	01011000	130	58	88
Y	01011001	131	59	89
Z	01011010	132	5A	90
[	01011011	133	5B	91
\	01011100	134	5C	92
]	01011101	135	5D	93
^	01011110	136	5E	94
_	01011111	137	5F	95

ASCII Char.	EQUIVALENT FORMS			
	Binary	Oct	Hex	Dec
`	01100000	140	60	96
a	01100001	141	61	97
b	01100010	142	62	98
c	01100011	143	63	99
d	01100100	144	64	100
e	01100101	145	65	101
f	01100110	146	66	102
g	01100111	147	67	103
h	01101000	150	68	104
i	01101001	151	69	105
j	01101010	152	6A	106
k	01101011	153	6B	107
l	01101100	154	6C	108
m	01101101	155	6D	109
n	01101110	156	6E	110
o	01101111	157	6F	111
p	01110000	160	70	112
q	01110001	161	71	113
r	01110010	162	72	114
s	01110011	163	73	115
t	01110100	164	74	116
u	01110101	165	75	117
v	01110110	166	76	118
w	01110111	167	77	119
x	01111000	170	78	120
y	01111001	171	79	121
z	01111010	172	7A	122
{	01111011	173	7B	123
ǀ	01111100	174	7C	124
}	01111101	175	7D	125
~	01111110	176	7E	126
DEL	01111111	177	7F	127

The following table gives the octal value for an ASCII character in the most significant byte ("First Character" column) and the least significant byte ("Second Character" column) of a word. The diagram illustrates the positions of the first and second character positions of a word.

First Character								Second Character							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent	ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
NUL	000000	000000	%	022400	000045
SOH	000400	000001	&	023000	000046
STX	001000	000002	'	023400	000047
ETX	001400	000003	(	024000	000050
EOT	002000	000004	)	024400	000051
ENQ	002400	000005	*	025000	000052
ACK	003000	000006	+	025400	000053
BEL	003400	000007	,	026000	000054
BS	004000	000010	-	026400	000055
HT	004400	000011	.	027000	000056
LF	005000	000012	/	027400	000057
VT	005400	000013	0	030000	000060
FF	006000	000014	1	030400	000061
CR	006400	000015	2	031000	000062
SO	007000	000016	3	031400	000063
SI	007400	000017	4	032000	000064
DLE	010000	000020	5	032400	000065
DC1	010400	000021	6	033000	000066
DC2	011000	000022	7	033400	000067
DC3	011400	000023	8	034000	000070
DC4	012000	000024	9	034400	000071
NAK	012400	000025	:	035000	000072
SYN	013000	000026	;	035400	000073
ETB	013400	000027	<	036000	000074
CAN	014000	000030	=	036400	000075
EM	014400	000031	>	037000	000076
SUB	015000	000032	?	037400	000077
ESC	015400	000033	@	040000	000100
FS	016000	000034	A	040400	000101
GS	016400	000035	B	041000	000102
RS	017000	000036	C	041400	000103
US	017400	000037	D	042000	000104
SP	020000	000040	E	042400	000105
!	020400	000041	F	043000	000106
"	021000	000042	G	043400	000107
#	021400	000043	H	044000	000110
$	022000	000044	I	044400	000111

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent	ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
J	045000	000112	e	062400	000145
K	045400	000113	f	063000	000146
L	046000	000114	g	063400	000147
M	046400	000115	h	064000	000150
N	047000	000116	i	064400	000151
O	047400	000117	j	065000	000152
P	050000	000120	k	065400	000153
Q	050400	000121	l	066000	000154
R	051000	000122	m	066400	000155
S	051400	000123	n	067000	000156
T	052000	000124	o	067400	000157
U	052400	000125	p	070000	000160
V	053000	000126	q	070400	000161
W	053400	000127	r	071000	000162
X	054000	000130	s	071400	000163
Y	054400	000131	t	072000	000164
Z	055000	000132	u	072400	000165
[	055400	000133	v	073000	000166
\	056000	000134	w	073400	000167
]	056400	000135	x	074000	000170
^	057000	000136	y	074400	000171
8	057400	000137	z	075000	000172
`	060000	000140	{	075400	000173
a	060400	000141	⊦	076000	000174
b	061000	000142	}	076400	000175
c	061400	000143	∿	077000	000176
d	062000	000144	DEL	077400	000177

Appendix B
Machine Instructions

Detailed List

Instruction	Form	Group	Description
AAR	AAR {n}	Shift/Rotate	Shifts the A register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)
ABR	ABR{n}	Shift/Rotate	Shifts the B register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)
ADA	ADA {loc} [, I]	Integer Math	Adds the contents of the specified location to the contents of register A. The result is in A. If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow is a carry from bit 15 or 14, but not both. Extend and Overflow are bits in the processor. Specifying register R4, R5, R6, or R7 as the location causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
ADB	ADB{loc} [, I]	Integer Math	Adds the contents of the specified location to the contents of register B. The result is in B. If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow is a carry from bit 15 or 14, but not both. Extend and Overflow are bits in the processor. Specifying register R4, R5, R6, or R7 as the location causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
AND	AND {loc} [, I]	Logical	Logical "and" operation. The contents of the A register are compared, bit by bit, with the contents of the specified location. For each bit comparison a 1 results if both bits are 1's, a 0 results otherwise. The 16-bit result is left in A. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.

Instruction	Form	Group	Description
CBL	CBL	Stack	Clears the Cb register. Specifies the lower block of memory for byte-referencing stack instructions.
CBU	CBU	Stack	Sets the Cb register. Specifies the upper block of memory for byte-referencing stack instructons.
CDC	CDC	BCD Math	Clears Decimal Carry explicitly.
CLA	CLA	Shift	Clears register A. This is exactly equivalent to SAR 16.
CLB	CLB	Shift	Clears register B. This is exactly equivalent to SBR 16.
CLR	CLR {n}	Load/Store	Clears the specified number of words, beginning at the location pointed at by the A register. A maximum of 16 words may be cleared.
CMA	CMA	Memory	Perform a one's complement of the A register (bit by bit inversion of all 16 bits).
CMB	CMB	Memory	Perform a one's complement of the B register (bit by bit inversion of all 16 bits).
CMX	CMX	BCD Math	Ten's complement of Ar1. The mantissa of Ar1 is replaced with its ten's complement and Decimal Carry is cleared.
CMY	CMY	BCD Math	Ten's complement of Ar2. The mantissa of Ar2 is replaced with its ten's complement and Decimal Carry is cleared.
CPA	CPA {loc} [, I]	Test/Branch	Compares the contents of register A with the contents of the specified location and skips if they are unequal. Indirect addressing may be specified. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. {loc} must be on base or current page.
CPB	CPB {loc} [, I]	Test/Branch	Compares the contents of register B with the contents of the specified location and skips if they are unequal. Indirect addressing may be specified. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. {loc} must be on base or current page. {loc} must be on base or current page.
DBL	DBL	Stack	Clears the Db register. Specifies the lower block of memory for byte-referencing stack instructions.
DBU	DBU	Stack	Sets the Db register. Specifies the upper block of memory for byte-referencing stack instructions.
DDR	DDR	I/O	Disables Data Request. Cancels the DMA instruction.
DIR	DIR	I/O	Disables the interrupt system. Cancels the EIR instruction.
DMA	DMA	I/O	Enables the DMA mode. Cancels the DDR instruction.

Instruction	Form	Group	Description
DRS	DRS	BCD Math	Mantissa right shift of Ar1 for one digit. The twelfth digit is shifted into bits 0-3 of the A register. The non-digit part of the A register is cleared (bits 4-15), and the Decimal Carry bit in the processor is cleared. The first digit in the mantissa is set to 0.
DSZ	DSZ {loc} [, I]	Test/Alter/Branch	Decrements the contents of the specified location and skips if the new contents are 0. Specifying register R4, R5, R6, or R7 causes an input (or an input and an output) bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
EIR	EIR	I/O	Enables the interrupt system. Cancels the DIR instruction.
EXE	EXE {reg} [, I]	Miscellaneous	Executes the contents of a register. {reg} is an integer in the range of 0 through 31, indicating the register to be used (see Memory Map for the correspondence between location and register). The register is left unchanged unless the instruction code causes it to be altered. The next instruction to be executed is the one following the EXE, unless the code in the executed register causes a branch. Indirect addressing may be specified.
FDV	FDV	BCD Math	Fast divide. The mantissas of Ar1 and Ar2 are added together, along with Decimal Carry, until the first decimal overflow occurs. The result accumulates into Ar2. The number of additions without overflow is placed into the lower 4 bits of the B register (0-3). The remainder of the B register is cleared, as is the Decimal Carry bit in the processor.
FMP	FMP	BCD Math	Fast Multiply. Performs the multiplication by repeated additions. The mantissa of Ar1 is added to Ar2 along with Decimal carry, a specified number of times. The number of times is specified in the lower 4 bits (0-3) of the B register. The result accumulates in Ar2. If intermediate overflows occur, the number of times they occur appears in the lower 4 bits of the A register after the operation is complete. The upper 12 bits of the A register are cleared along with Decimal Carry.
FXA	FXA	BCD Math	Fixed-point addition. The mantissas of Ar1 and Ar2 are added together and the result placed in Ar2. Decimal Carry is used as the twelfth digit. After the addition, Decimal Carry is set if an overflow occurred, otherwise Decimal Carry is cleared.

Instruction	Form	Group	Description
IOR	IOR {loc} [, I]	Logical	Logical "inclusive or" operation. The contents of the A register are compared, bit by bit, with the contents of the specified location. For each bit comparison, a 0 results if both bits are 0's, a 1 otherwise. The 16-bit result is left in A. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
ISZ	ISZ {loc} [, I]	Test/Alter/Branch	Increments the contents of the specified location and skips if the new contents are 0. Specifying register R4, R5, R6, or R7 causes an input (or an input followed by an output) bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
JMP	JMP {loc} [, I]	Branch	Unconditionally branches to the specified location. Indirect addressing may be specified. {loc} must be on base or current page.
JSM	JSM {loc} [, I]	Branch	Jumps to subroutine. The value of the R register is incremented by 1 and the value of the P register (i.e., the location of the JSM instruction itself) is stored in the address pointed to by the R register. Execution then proceeds to the specified location. Return from the subroutine is effected by the RET instruction. Indirect addressing may be specified. {loc} must be on base or current page.
LDA	LDA {loc} [, I]	Load/Store	Loads register A with the contents of the specified location. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
LDB	LDB {loc} [, I]	Load/Store	Loads register B with the contents of the specified location. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
MLY	MLY	BCD Math	Mantissa left shift on Ar2 for one digit. This is a circular shift, with the bits 0-3 of the A register forming a thirteenth digit. The non-digit part of the A register is cleared (bits 4-15), and the Decimal Carry bit in the processor is cleared.
MPY	MPY	Integer Math	Binary multiply. Uses Booth's Algorithm. The values of the A and B registers are multiplied together with the product placed into A and B. The A register contains the least significant bits and the B register contains the most significant bits and the sign. B may contain any integer within the range −32 767 to +32 767.

Instruction	Form	Group	Description
MRX	MRX	BCD Math	Mantissa right shift on Ar1. The number of digits to be shifted is specified in the lower 4 bits (0-3) of the B register. The shift is accomplished in three stages: 1) Bits 0-3 of the A register are right-shifted into D_1 of the mantissa, with the twelfth digit being lost. This is the first shift. This shift always takes place, even if B = 0. 2) The digits are then right-shifted for the remaining number of digits specified. The twelfth digit is lost on each shift (except for the last shift) and the vacated digits are zero-filled. 3) Finally, the last right-shifting takes place, with the twelfth digit shifting into the lower 4 bits (0-3) of the A register. The Decimal Carry bit in the processor is cleared and the non-digit part of the A register is cleared (bits 4-15).
MRY	MRY	BCD Math	Mantissa right shift on Ar2. The number of digits to be shifted is specified in the lower 4 bits (0-3) of the B register. The shift is accomplished in three stages: 1) Bits 0-3 of the A register are right-shifted into D_1 of the mantissa, with the twelfth digit being lost. This is the first shift. This shift always takes place, even if B = 0. 2) The digits are right-shifted for the remaining number of digits specified. The twelfth digit is lost on each shift (except for the last shift) and the vacated digits are zero-filled. 3) Finally, the last right-shifting takes place, with the twelfth digit shifting into the lower 4 bits (0-3) of the A register. The non-digit part of the A register is cleared (bits 4-15), and the Decimal Carry bit in the processor is cleared.
MWA	MWA	BCD Math	Mantissa word addition. The contents of the B register are added to the ninth through twelfth digits of the Ar2 register. Decimal Carry is added to the twelfth digit; if an overflow occurs, Decimal Carry is set, otherwise Decimal Carry is cleared.
NOP	NOP	Miscellaneous	Null operation. This is exactly equivalent to LDA A.
NRM	NRM	BCD Math	Normalizes the Ar2 mantissa. Up to twelve left-shifts of the mantissa are performed until the first digit of the mantissa is non-zero. If the original first digit is already non-zero, no shifts occur. The number of shifts required is stored in the first 4 bits (0-3) of the B register. If 12 shifts are required, the Decimal Carry bit in the processor is set; otherwise, the Decimal Carry bit is cleared. The exponent is not altered.

Instruction	Form	Group	Description
PBC	PBC {reg} [, I]	Stack	Pushes the lower byte (right half) of the specified register onto the stack pointed at by the Cb and C registers. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing of the C register can be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action takes place before pushing.
PBD	PBD {reg} , D PBD {reg} [, I]	Stack	Pushes the lower byte (right half) of the specified register onto the stack pointed at by the Db and D registers. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the D register can be specified. Incrementing is the default. {reg} must be in the range of 0 through 7.The incrementing or decrementing action takes place before pushing.
PWC	PWC {reg} , D PWC {reg} [, I]	Stack	Pushes entire register (full word) onto the stack pointed at by the C register. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the C register may be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action takes place before pushing.
PWD	PWD {reg} , D PWD {reg} [, I]	Stack	Pushes the entire register (full word) onto the stack pointed at by the D register. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the D register may be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action taken place before pushing.
RAL	RAL {n}	Shift/Rotate	Rotates the A register left the indicated number of bits. Bit 15 rotates into bit 0 (left circular). Maximum rotation of 16 bits.
RAR	RAR {n}	Shift/Rotate	Rotates the A register right the indicated number of bits. Bit 0 rotates into bit 15 (right circular). Maximum rotation of 16 bits.
RBL	RBL {n}	Shift/Rotate	Rotates the B register left the indicated number of bits. Bit 15 rotates into bit 0 (left circular). Maximum rotation of 16 bits rotated.

Instruction	Form	Group	Description
RBR	RBR {n}	Shift/Rotate	Rotates the B register right the indicated number of bits. Bit 0 rotates into bit 15 (right circular). Maximum rotation of 16 bits.
RET	RET {n}	Branch	Returns from subroutine. {n} is added to the contents of the address pointed to by the R register. The R register is decremented by 1. This is, in effect, a return from a JSM instruction (see above), to {n} instructions following the JSM itself. The "usual" return is RET 1. {n} must be in the range of – 32 through 31.
RIA	RIA {adrs}	Test/Branch	Skips to {adrs} if register A is not 0, then increments register A by 1. Extend and Overflow are not effected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within – 32 and + 31 of the current location.
RIB	RIB {adrs}	Test/Branch	Skips to {adrs} if register B is not 0, then increments register B by 1. Extend and Overflow are not affected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within – 32 and + 31 of the current location.
RLA	RLA {adrs} [, S] RLA {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the A register is not 0. Setting or clearing the bit after the test can be specified. {adrs} must be within – 32 and + 31 of the current location.
RLB	RLB {adrs} [, S] RLB {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the B register is not 0. Setting or clearing the bit after the test can be specified. {adrs} must be within – 32 and + 31 the current location.
RZA	RZA {adrs}	Test/Branch	Skips to {adrs} if register A is not 0. {adrs} must be within – 32 and + 31 of the current location.
RZB	RZB {adrs}	Test/Branch	Skips to {adrs} if register B is not 0. {adrs} must be within – 32 and + 31 of the current location.
SAL	SAL {n}	Shift/Rotate	Shifts the A register left the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.
SAM	SAM {adrs} [, S] SAM {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the A register is negative (bit 15 is 1). Setting or clearing the bit after the test can be specified. {adrs} must be within – 32 and + 31 of the current location.
SAP	SAP {adrs} [, S] SAP {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the A register is positive or zero (bit 15 is 0). Setting or clearing the bit after the test can be specified. {adrs} must be within – 32 and + 31 of the current location.

Instruction	Form	Group	Description
SAR	SAR {n}	Shift/Rotate	Shifts the A register right the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.
SBL	SBL {n}	Shift/Rotate	Shifts the B register left the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.
SBM	SBM {adrs} [, S] SBM {adrs} [, C]	Test/Alter/Branch Test/Alter/Branch	Skips to {adrs} if the B register is negative (bit 15 is 1). Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.
SBP	SBP {adrs} [, S] SBP {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the B register is positive (bit 15 is 0). Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.
SBR	SBR {n}	Shift/Rotate	Shifts the B register right the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.
SDC	SDC {adrs}	BCD Math	Skips to {adrs} if Decimal Carry is clear. Decimal carry is a single bit in the processor which may have been set by certain arithmetic operations. {adrs} must be within − 32 and + 31 of the current location.
SDI	SDI	I/O	Sets DMA inwards. Reads from peripheral, writes to memory.
SDO	SDO	I/O	Sets DMA outwards. Reads from memory, writes to peripheral.
SDS	SDS {adrs}	BCD Math	Skips to {adrs} if Decimal Carry is set. Decimal carry is a single bit in the processor which may have been set by certain arithmetic operations. {adrs} must be with − 32 and + 31 of the current location.
SEC	SEC {adrs} [, S] SEC {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if Extend is clear. Extend is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.
SES	SES {adrs} [, S] SES {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if Extend is set. Extend is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.

Instruction	Form	Group	Description
SFC	SFC {adrs}	I/O	Skips to {adrs} if the Flag line is false (clear). The Flag line is the one associated with a peripheral on the current select code (pointed to by the Pa register). {adrs} must be within − 32 and + 31 of the current location.
SFS	SFS {adrs}	I/O	Skips to {adrs} if the Flag line is true (set). The flag line is that associated with the peripheral on the current select code (pointed to by the Pa register). {adrs} must be within - 32 and + 31 of the current location.
SHC	SHC{{adrs}	Test/Branch	Skips to {address} if CRT is scanning its raster. {adress} must be within −32 and +31 of the current location.
SHS	SHS{adrs}	Test/Branch	Skips to {address} if CRT is doing vertical retrace. {address} must be within −32 and +31 of the current location.
SIA	SIA {adrs}	Test/Branch	Skips to {adrs} if register A is 0, then increments register A by 1. Extend and Overflow are not affected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within − 32 and + 31 of the current location.
SIB	SIB {adrs}	Test/Branch	Skips to {adrs} if register B is 0, then increment register B by 1. Extend and Overflow are not affected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within − 32 and + 31 of the current location.
SLA	SLA {adrs} [, S] SLA {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the A register is 0. Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.
SLB	SLB {adrs} [, C] SLB {adrs} [, S]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the B register is 0. Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.
SOC	SOC {adrs} [, S] SOC {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if Overflow is clear. Overflow is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.

Instruction	Form	Group	Description
SOS	SOS {adrs} [, S] SOS {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the Overflow is set. Overflow is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within − 32 and + 31 of the current location.
SSC	SSC {adrs}	I/O	Skips to {adrs} if Status line is false (clear). The status line is the one associated with the peripheral on the current select code (pointed to by the Pa register). {adrs} must be within − 32 and + 31 of the current location.
SSS	SSS {adrs}	I/O	Skips to {adrs} if the Status line is true (set). The status line is the one associated with the peripheral on the current select code (pointed to by the Pa register). {adrs} must be within − 32 and + 31 of the current location.
STA	STA {loc} [, I]	Load/Store	Stores the contents of the A register into the specified location. Specifying register R4, R5, R6, or R7 causes an output bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
STB	STB {loc} [, I]	Load/Store	Stores the contents of the B register into the specified location. Specifying register R4, R5, R6, or R7 causes an output bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.
SZA	SZA {adrs}	Test/Branch	Skips to {adrs} if register A is 0. {adrs} must be within − 32 and + 31 of the current location.
SZB	SZB {adrs}	Test/Branch	Skips to {adrs} if register B is 0. {adrs} must be within − 32 and + 31 of the current location.
TCA	TCA	Integer Math	Performs a two's complement of the A register (one's complement, incremented by 1). If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow occurs when complementing − 32 768. Extend and Overflow are bits in the processor.
TCB	TCB	Integer Math	Performs a two's complement of the B register (one's complement, incremented by 1). If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow occurs when complementing − 32 768. Extend and Overflow are bits in the processor.

Instruction	Form	Group	Description
WBC	WBC {reg} [, D] WBC {reg} , I	Stack	Withdraws a byte from the stack pointed at by the Cb and C registers and places it into the lower byte (right half) of the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the C register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing routine takes place after the withdrawal.
WBD	WBD {reg} [, D] WBD {reg} , I	Stack	Withdraws a byte from the stack pointed at by the Db and D registers and places it into the lower byte (right half) of the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the D register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing routine takes place after the withdrawal.
WWC	WWC {reg} [, D] WWC {reg} , I	Stack	Withdraws a full word from the stack pointed at by the C register and places it into the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing of the C register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing action takes place after the withdrawal.
WWD	WWD {reg} [, D] WWD {reg} , I	Stack	Withdraws a full word from the stack pointed at by the D register and places it into the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing of the D register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing action takes place after the withdrawal.
XFR	XFR {n}	Load/Store	Transfers the specified number of words, from the location starting at the address pointed at by the A register to the location starting at the address pointed at by the B register. A maximum of 16 words can be transferred.

Approximate Numerical List
Bit Patterns

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LD A/B	D/I	0	0	0	A/B	Address Field										
CP A/B	D/I	0	0	1	A/B											
AD A/B	D/I	0	1	0	A/B											
ST A/B	D/I	0	1	1	A/B											
JSM	D/I	1	0	0	0											
AND	D/I	1	0	1	0											
I/D SZ	D/I	1	0	I/D	1											
IOR	D/I	1	1	0	0											
JMP	D/I	1	1	0	1											
EXE	D/I	1	1	1	0	0	0	0	0	0	0	0	Register Address			
SD O/I	0	1	1	1	0	0	0	1	0	0	0	0	O/I	0	0	0
E/D IR	0	1	1	1	0	0	0	1	0	0	0	1	E/D	0	0	0
DMA	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0
DDR	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0
D/C B U/L	0	1	1	1	0	0	0	1	0	1	0	U/L	D/C	0	0	0
P/W W/B C/D	0	1	1	1	W/B	0	0	1	I/D	1	1	P/W	C/D	Register Address		
MWA	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
CM Y/X	0	1	1	1	0	0	1	0	0	Y/X	1	0	0	0	0	0
FXA	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0
XFR	0	1	1	1	0	0	1	1	0	0	0	0	N=# of words binary=(n−1)			
CLR	0	1	1	1	0	0	1	1	1	0	0	0				
NRM	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0
CDC	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0
FMP	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
FDV	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1
MRX	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0
DRS	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1
MRY	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0
MLY	0	1	1	1	1	0	1	1	0	1	1	0	0	0	0	1
MPY	0	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1
S F/D S/C	0	1	1	1	0	1	0	S/C	1	F/D	Skip Field: if bit 5 is 0, then skip to(P+n), n=bits 0-4; if bit 5=1, then skip to(P−n), n=two's complement of bits 0-4					
R/S Z/I A/B	0	1	1	1	A/B	1	0	R/S	0	Z/I						
S/R L A/B	0	1	1	1	A/B	1	1	S/R	H/$\overline{H}$	C/S						
SS S/C	0	1	1	1	1	1	0	S/C	1	0						
SH S/C	0	1	1	1	1	1	0	S/C	1	1						
S A/B P/M	1	1	1	1	A/B	1	0	P/M	H/$\overline{H}$	C/S						
S O/E C/S	1	1	1	1	O/E	1	1	C/S	H/$\overline{H}$	C/S						
RET	1	1	1	1	0	0	0	0	1	0	complemented skip field					
TC A/B	1	1	1	1	A/B	0	0	0	0	0	1	0	0	0	0	0
CM A/B	1	1	1	1	A/B	0	0	0	0	1	1	0	0	0	0	0
CL A/B	1	1	1	1	A/B	0	0	1	0	1	0	0	1	1	1	1
A A/B R	1	1	1	1	A/B	0	0	1	0	0	0	0	Shift Field in source, n=1-16 binary=(n−1)			
R/S A/B R	1	1	1	1	A/B	0	0	1	R/S	1	0	0				
S A/B L	1	1	1	1	A/B	0	0	1	1	0	0	0				
R A/B L	1	1	1	1	A/B	0	0	1	1	1	0	0	complemented shift			

Alphabetic List
Bit Patterns and Timings

Instruction	Bit Pattern																Timing	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Direct	Indirect
AAR n	1	1	1	1	0	0	0	1	0	0	0	0	←	n−1		→	n+9	
ABR n	1	1	1	1	1	0	0	1	0	0	0	0	←	n−1		→	n+9	
ADA	D/I	0	1	0	0	B/C	←				address					→	13	19
ADB	D/I	0	1	0	1	B/C	←				address					→	13	19
AND	D/I	1	0	1	0	B/C	←				address					→	13	19
CBL	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0	0	12	
CBU	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0	0	12	
CDC	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	11	
CLA	1	1	1	1	0	0	0	1	0	1	0	0	1	1	1	1	25	
CLB	1	1	1	1	1	0	0	1	0	1	0	0	1	1	1	1	25	
CLR n	0	1	1	1	0	0	1	1	1	0	0	0	←	n−1		→	6n+16	
CMA	1	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	9	
CMB	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	9	
CMX	0	1	1	1	0	0	1	0	0	1	1	0	0	0	0	0	59	
CMY	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0	23	
CPA	D/I	0	0	1	0	B/C	←				address					→	16	22
CPB	D/I	0	0	1	1	B/C	←				address					→	16	22
DBL	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	12	
DBU	0	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0	12	
DDR	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	12	
DIR	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0	12	
DMA	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0	12	
DRS	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1	56	
DSZ	D/I	1	0	1	1	B/C	←				address					→	19	25
EIR	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	12	
EXE	D/I	1	1	1	0	0	0	0	0	0	0	←		register		→	8	14
FDV	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1	37+13B	
FMP	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	42+13B	
FXA	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	40	
IOR	D/I	1	1	0	0	B/C	←				address					→	13	19
ISZ	D/I	1	0	0	1	B/C	←				address					→	19	25
JMP	D/I	1	1	0	1	B/C	←				address					→	8	14
JSM	D/I	1	0	0	0	B/C	←				address					→	17	23
LDA	D/I	0	0	0	0	B/C	←				address					→	13	19
LDB	D/I	0	0	0	1	B/C	←				address					→	13	19
MLY	0	1	1	1	1	0	1	1	0	1	1	0	0	0	0	1	32	
MPY	0	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1	65+2T	
MRX	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	62+4B	
MRY	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	33+4B	
MWA	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	28	
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	
NRM	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	23+Z	
PBC r	0	1	1	1	1	0	0	1	I/D	1	1	0	0	←	r	→	23	
PBD r	0	1	1	1	1	0	0	1	I/D	1	1	0	1	←	r	→	23	
PWC r	0	1	1	1	0	0	0	1	I/D	1	1	0	0	←	r	→	23	
PWD r	0	1	1	1	0	0	0	1	I/D	1	1	0	1	←	r	→	23	

Instruction	Bit Pattern																Timing	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Direct	Indirect
RAL n	1	1	1	1	0	0	0	1	1	1	0	0	← 15−n →				25−n	
RAR n	1	1	1	1	0	0	0	1	1	1	0	0	← n−1 →				n+9	
RBL n	1	1	1	1	1	0	0	1	1	1	0	0	← 15−n →				25−n	
RBR n	1	1	1	1	1	0	0	1	1	1	0	0	← n−1 →				n+9	
RET	1	1	1	1	0	0	0	0	1	0	← skip →						16	
RIA	0	1	1	1	0	1	0	0	0	1	← skip →						14	
RIB	0	1	1	1	1	1	0	0	0	1	← skip →						14	
RLA	0	1	1	1	0	1	1	1	H/$\bar{H}$	C/S	← skip →						14	
RLB	0	1	1	1	1	1	1	1	H/$\bar{H}$	C/S	← skip →						14	
RZA	0	1	1	1	0	1	0	0	0	0	← skip →						14	
RZB	0	1	1	1	1	1	0	0	0	0	← skip →						14	
SAL n	1	1	1	1	0	0	0	1	1	0	0	0	← n−1 →				n+9	
SAM	1	1	1	1	0	1	0	1	H/$\bar{H}$	C/S	← skip →						14	
SAP	1	1	1	1	0	1	0	0	H/$\bar{H}$	C/S	← skip →						14	
SAR n	1	1	1	1	0	0	0	1	0	1	0	0	← n−1 →				n+9	
SBL n	1	1	1	1	1	0	0	1	1	0	0	0	← n−1 →				n+9	
SBM	1	1	1	1	1	1	0	1	H $\bar{H}$	C S	← skip →						14	
SBP	1	1	1	1	1	1	0	0	H $\bar{H}$	C S	← skip →						14	
SBR n	1	1	1	1	1	0	0	1	0	1	0	0	← n−1 →				n+9	
SDC	0	1	1	1	0	1	0	1	1	1	← skip →						14	
SDI	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	0	12	
SDO	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	12	
SDS	0	1	1	1	0	1	0	0	1	1	← skip →						14	
SEC	1	1	1	1	1	1	1	0	H $\bar{H}$	C S	← skip →						14	
SES	1	1	1	1	1	1	1	1	H $\bar{H}$	C S	← skip →						14	
SFC	0	1	1	1	0	1	0	1	1	0	← skip →						14	
SFS	0	1	1	1	0	1	0	0	1	0	← skip →						14	
SHC	0	1	1	1	1	1	0	1	1	1	← skip →						14	
SHS	0	1	1	1	1	1	0	0	1	1	← skip →						14	
SIA	0	1	1	1	0	1	0	1	0	1	← skip →						14	
SIB	0	1	1	1	1	1	0	1	0	1	← skip →						14	
SLA	0	1	1	1	0	1	1	0	H $\bar{H}$	C S	← skip →						14	
SLB	0	1	1	1	1	1	1	0	H $\bar{H}$	C S	← skip →						14	
SOC	1	1	1	1	0	1	1	0	H $\bar{H}$	C S	← skip →						14	
SOS	1	1	1	1	0	1	1	1	H $\bar{H}$	C S	← skip →						14	
SSC	0	1	1	1	1	1	0	1	1	0	← skip →						14	
SSS	0	1	1	1	1	1	0	0	1	0	← skip →						14	
STA	D/I	0	1	1	0	B/C	← address →										13	19
STB	D/I	0	1	1	1	B/C	← address →										13	19
SZA	0	1	1	1	0	1	0	1	0	0	← skip →						14	
SZB	0	1	1	1	1	1	0	1	0	0	← skip →						14	
TCA	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	9	
TCB	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	9	
WBC r	0	1	1	1	1	0	0	1	I/D	1	1	1	0	← r →			23	
WBD r	0	1	1	1	1	0	0	1	I/D	1	1	1	1	← r →			23	
WWC r	0	1	1	1	0	0	0	1	I/D	1	1	1	0	← r →			23	
WWD r	0	1	1	1	0	0	0	1	I/D	1	1	1	1	← r →			23	
XFR n	0	1	1	1	0	0	1	1	0	0	0	0	← n−1 →				12n+21	

Bit Patterns and Timings

Notes on timings:

All timings are in clock cycles. One clock cycle = 175 nanoseconds. (The clock rate is 5.7 megahertz.)

The symbols used to represent timing information are as follows:

n – number of bit positions to be shifted or rotated.
N – the current value in bits 0-3 of the instruction word.
B – the current value in bits 0-3 of the instruction word.
T – the total number of 0→1 and 1→0 transitions in the A register using an imaginary 0 to the right of bit 0.
Z – the number of leading zeroes in the mantissa of Ar2. If Z = 12, then the total timing is 69 clock cycles.

Other factors that affect timing are as follows:

- Up to 4.3% of the total processor execution time is dedicated to dynamic memory refresh.
- The total execution time dedicated to CRT refresh is –

Minimum	Typical	Maximum
6 clock cycles every 1/60 sec. (GRAPHICS mode)	5%	30% (Full screen of alternating blinking, underlined or inverse-video characters)

- Interrupt response depends upon certain hardware and software considerations. The processor must be enabled with an EIR instruction. The operating system is allowed to disable interrupts for up to 100 μs during various operations. A fast handshake transfer locks out interrupts until the transfer is complete. The processor must complete the currently executing instruction before acknowledging an interrupt.

- Add two clock cycles to the instruction execution time if an interrupt is pending. Software overhead involved in getting to a user interrupt service routine consitutes a delay of approximately 50 μs to get to the service routine and 50 μs to return from the service routine. These delays can be lengthened by the effects of DMA, CRT refresh and memory refresh.

- The processor locks out the initiation of a DMA transfer for a minimum of two clock cycles and a maximum of 64 clock cycles. The times involved for DMA transfers are –

  DMA read = 3 + (10n + d) + lockout time
  DMA write = 3 + (9n + d) + lockout time

  where n is the number of words transferred and d is the dual-port conflict time (0 = no conflict...5 = continuous conflict). Since DMA transfers take priority over instruction execution, these transfers can take up to 100% of the processor time, depending on the data transfer rate of the peripheral device. The worst case involves data transfers to and from a high-speed, hard disc.

- Due to bus conflicts resulting from two processors requesting one bus, processor interference can affect timing. If a background program is executed entirely from the ICOM region, processor interference does not come into play. This is the typical case. The worst case involves executing a BASIC program simultaneously with an ISR. In this case, program execution time can be as much as doubled.

Appendix C
Pseudo-Instructions

The following table lists the available assembler pseudo-instructions with a short description of each.

Instruction	Form	Description
ANY	ANY	Specifies a common or subroutine declaration to be any type
BSS	BSS {expression}	Reserves a block of memory
COM	COM	Preface for assembly language common declarations
DAT	DAT {expression} [, {expression} [, ...]]	Defines data generators
END	END {name}	Designates the end of a module
ENT	ENT {symbol} [, {symbol} [, ...]]	Identifies entry points in the module
EQU	EQU {expression}	Defines a symbol
EXT	EXT {symbol} [, {symbol} [, ...]]	Identifies external entry points
FIL	FIL	Specifies a subroutine declaration to be a file number
HED	HED {comment}	Source listing control for top-of-page with change of heading
IFA IFB IFC IFD IFE IFF IFG IFH IFP	IFA IFB IFC IFD IFE IFF IFG IFH IFP {numeric expression}	Beginning of conditional assembly
INT	INT [(*)]	Specifies a common or subroutine declaration to be an integer
IOF	IOF	Turns off automatic indirection by the assembler
ION	ION	Turns on automatic indirection by the assembler
LIT	LIT {expression}	Reserve memory for literals and links
LST	LST	Source listing control for enabling the listing

Instruction	Form	Description
NAM	NAM {name}	Designates the beginning of a module
REL	REL [(*)]	Specifies a common or subroutine declaration to be full-precision
REP	REP {expression}	Repeats instructions
SET	SET {expression}	Defines a symbol
SHO	SHO [(*)]	Specifies a common or subroutine declaration to be short-precision
SKP	SKP	Source listing control for top-of-page
SPC	SPC {integer expression}	Source listing control for printing blank lines
STR	STR [(*)]	Specifies a common or subroutine declaration to be a string
SUB	SUB	Preface for a subroutine entry point Contains actual number of parameters passed by ICALL statement after assembly.
UNL	UNL	Source listing control for disabling the listing
XIF	XIF	End of a conditional-assembly block

Appendix D
Assembly Language BASIC Language Extensions Formal Syntax

The following is an alphabetical list of the BASIC Language extensions provided by the Assembly Language ROMs.

Assembled Location

{symbol} [, {BASIC numeric expression}]
{expression} [, {BASIC numeric expression}]

where:
{BASIC numeric expression} serves as a decimal offset from the given label or constant.

{symbol} is an assembly location. It may be either a label for a particular machine instruction (in which case the address of the associated instruction is used), or an assembler-defined symbol (in which case the associated absolute address is used), or a symbol defined by an EQU instruction (in which case the associated value is used).

{expression} may be a numeric expression or a string expression. If numeric, a decimal calculation is performed and the result is interpreted as an octal value; if the result is not an octal representation or an integer, an error results. If a string expression is used, the string must be interpretable as either an octal integer constant or a known assembly symbol (see {symbol} above).

DECIMAL Function

DECIMAL ({BASIC numeric expression})

IADR Function

IADR ({assembled location})

IASSEMBLE

`IASSEMBLE` {module} [, {module} [, ...]] [; {option} [, {option} [, ...]]]

`IASSEMBLE [ALL]` [; {option} [, {option} [, ...]]]

where {module} is the name of an existing module in the source program.

{option} may be any of the following:

`A`
`B`
`C`
`D`
`E`
`EJECT`
`F`
`G`
`H`
`LINES` {numeric expression}
`LIST`
`XREF`

IBREAK

`IBREAK` [`DATA`] {address} [; {counter}] [`CALL` {subprogram}]

`IBREAK` [`DATA`] {address} [; {counter}] [`GOSUB` {line identifier}]

`IBREAK` [`DATA`] {address} [; {counter}] [`GOTO` {line identifier}]

`IBREAK ALL` [`CALL` {subprogram}]

`IBREAK ALL` [`GOSUB` {line identifier}]

`IBREAK ALL` [`GOTO` {line identifier}]

where:

{address} is an assembled location.
{subprogram} is the name of a BASIC subprogram.
{counter} is a numeric expression.
{line identifier} is a line in the BASIC program.

ICALL

`ICALL` {routine} [({argument} [, {argument} [, ...]])]

where {routine} is the label associated with a SUB pseudo-instruction sequence and {data item} takes on the same forms and attributes as parameters in BASIC's CALL statement.

ICHANGE

ICHANGE {assembled location} TO {octal expression}

ICOM

ICOM {integer constant}

IDELETE

IDELETE {module} [, {module} [, ...]]

IDELETE [ALL]

where {module} is the name of an existing module in the ICOM region.

IDUMP

IDUMP {location} [; {location} [; ...]]

where {location} has the following syntax:

[{mode selection}] {address} [TO {address}]

with {address} an assembled location and {mode selection} taking on any of the following forms —

ASC	for ASCII character representation
BIN	for binary representation
DEC	for decimal representation
HEX	for hexadecimal representation
OCT	for octal representation

ILOAD

ILOAD {file specifier}

where {file specifier} is of the same form as elsewhere in BASIC (see Mass Storage Techniques manual, or Operating and Programming manual).

IMEM Function

IMEM ({assembled location})

INORMAL

`INORMAL` [{address}]

where {address} is an assembled location.

IPAUSE OFF

`IPAUSE OFF`

IPAUSE ON

`IPAUSE ON`

ISOURCE

`ISOURCE` {source line}

where {source line} may take either of the following forms —

[{label} :] {action} [! {comment}]
[{label} :] ! {comment}

and:

{label} is of the same form as elsewhere in BASIC;
{action} is a machine instruction, pseudo-instruction, or data generator;
{comment} is any combination of characters

ISTORE

`ISTORE` {module} [, {module} [, ...]] ; {file specifier}
`ISTORE [ALL];` {file specifier}

where:

{module} is the name of a module currently existing in the ICOM region.

{file specifier} is of the same form as elsewhere in BASIC (see the Mass Storage Techniques manual or the Operating and Programming manual).

LITERALS

= {expression} [, {expression} [, ...]]
{expression} may be absolute or relocatable

OCTAL Function

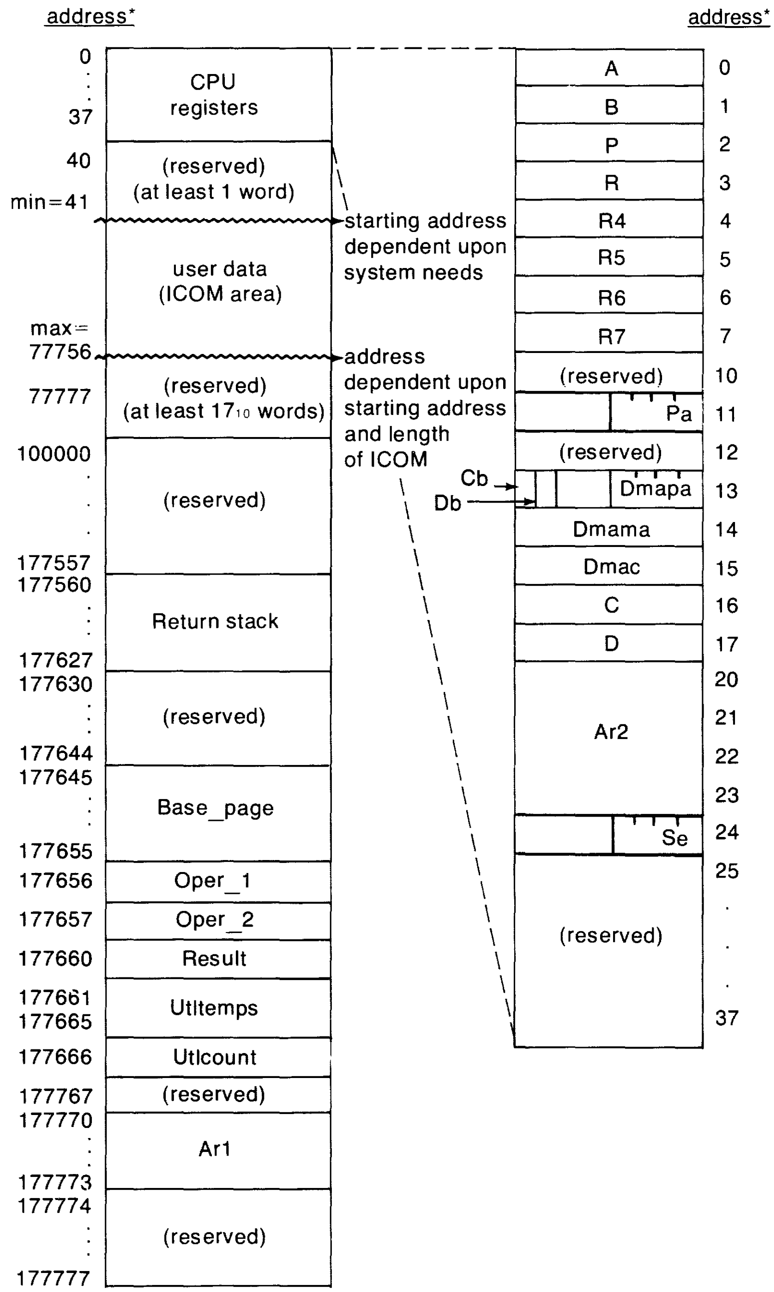
`OCTAL (` {numeric expression} `)`

Appendix E
Predefined Assembler Symbols

The assembler has predefined a number of symbols and has reserved them as references to special locations in memory. Each of these locations has a special meaning and function. You may not redefine these symbols. They are —

Name	Description
A	Arithmetic accumulator
Ar1	BCD arithmetic accumulators
Ar2	
B	Arithmetic accumulator
Base_page	Base page temporary area (9 words)
C	Stack pointer
Cb	Block bit for byte pointer in C(most significant bit of address 13B)
D	Stack pointer
Db	Block bit for byte pointer in D(second most significant bit of address 13B)
Dmac	DMA count register
Dmama	DMA memory address register
Dmapa	DMA peripheral register (lower 4 bits of address 13B)
End_isr_high	Reserved symbols for use with interrupt service routines
End_isr_low	
Isr_psw	
Oper_1	Arithmetic utility operand address registers
Oper_2	
P	Program counter
Pa	Peripheral address register (lower 4 bits of address 11B)
R	Return stack pointer
R4	I/O registers
R5	
R6	
R7	
Result	Arithmetic utility result address register
Se	Shift-extend register
Utlcount	Reserved symbols for writing utilities
Utlend	
Utltemps	

Each predefined symbol references a particular location in memory, except for the Utlend symbol, which refers to an execution address of a system routine. A graphical representation of these locations, plus others of interest, is presented on the next page.



*in octal representation

Utility Name	LDA with:	LDB with:	Exits	Other	Description	"Minimum R-stack Entries"
Busy	address of bit pattern	address of parameter	RET 1		Retrieves busy bits for a BASIC variable	2
Error_exit	error number	N/A	None — returns to BASIC		Aborts execution of ICALL statement, setting an error number	5
Get_bytes	address of storage area	address of parameter	RET 1	Storage area consists of: lst word — starting byte 2nd word — number of bytes to be transferred 3rd word on — sufficient space for string	Accesses substrings (or parts of arguments)	2
Get_elem_bytes	address of storage area	address of array info	RET 1	Array info obtained by Get_info utility. Relative element number must be stored in array pointer (word 16) of array info. Storage area same as in Get_bytes.	Same as "Get_bytes" used for accessing elements of string arrays	2
Get_file_info	address of storage area	file number	RET 2 — normal RET 1 — file unassigned	Storage area contents after return: word 0 – lower 16 bits of file address word 1 – number of defined records word 2 – current record number word 3 – current word in current record word 4 – size of defined record word 5 – mass storage unit specifier word 6 – buffer address word 7 – check read (0=off, 1=on) word 8 – high 7 bits of file address word 9 – (reserved by system)	Accesses a file-pointer	2
Get_info	address of storage area	address of array info	RET 1	Storage area must be at least: 3 words — simple variables 18 words — arrays for arrays, add 3 words for each 64K bytes in your machine's memory	Returns the characteristics of a variable passed as a parameter or existing in common	3
Get_element	address of storage area	address of parameter	RET 1	Array info obtained by Get_info utility. Relative element_number must be stored in array pointer (word 16) of array info. Storage area must be sufficient size to hold value.	Same as "Get_value", used for elements in an array	2
Get_value	address of storage area	address of parameter	RET 1	Storage area must be sufficient size to hold value	Returns the value of a BASIC variable	2
Int_to_rel	N/A	N/A	RET 1	Load address of integer into Oper_1 and address of storage area into Result. Storage area must be at least 4 words.	Data type conversion from integer to full-precision	2
Isr_access	address of ISR	select code in bits 0-3; access code in bits 4-5; trial counter bits 8-14	RET 1 — linkage not established for reason found in register A: – 1 = resources unobtainable – 2 = select code linked to another ISR RET 2 — normal	select code is 0-7 for low-level or 8-15 for high-level; resource code is: 0 — no resources 1 — asynchronous access 2 — asynchronous access with DMA 3 — synchronous access trial counter is number of attempts before aborting (RET 1, with A set to – 1)	Establishes linkages for interrupts	5+u*
Mm_read_start	address of mass storage descriptor	N/A	RET 1 — memory overflow RET 2 — normal (A contains mass storage transfer ID)	Mass storage descriptor is 3 words containing: word 1 — mass storage unit specifier word 2 — least significant 16 bits of record number word 3 — most significant 7 bits of record number	Prepares to read a physical record from mass storage	5+u*
Mm_read_xfer	mass storage transfer ID	address of storage area	RET 1 — transfer incomplete RET 2 — transfer complete (A contains 0, or error number encountered during transfer)	Storage area must be at least 128 words Mass storage transfer ID would be returned from Mm_read_start utility. Storage area receives transferred information	Reads a physical record from mass storage	5+u*

Utility Name	LDA with:	LDB with:	Exits	
Mm_write_start	address of mass storage descriptor	address of storage area	RET 1 — memory overflow RET 2 — normal (A contains mass storage transfer ID)	Mas Stor inf
Mm_write_test	mass storage transfer ID	N/A	RET 1 — transfer incomplete RET 2 — transfer complete (A contains 0, or error number encountered during transfer)	Mas Mn
Printer_select	select code	printer width	RET 1 (A contains previous printer select code; B contains previous printer width)	
Print_no_lf	address of string	N/A	RET 1 — memory overflow RET 2 — STOP pressed RET 3 — normal	Strir
Print_string	address of string	N/A	RET 1 — memory overflow RET 2 — STOP pressed RET 3 — normal	Strir
Put_bytes	address of storage area	address of parameter	RET 1	Stor
Put_elem_bytes	address of storage area	address of array info	RET 1	Sam
Put_element	address of storage area	address of array info	RET 1	Sam
Put_file_info	address of storage area	file number	RET 1 — file unassigned RET 2 — normal	Sam
Put_value	address of storage area	address of parameter	RET 1	
Rel_math	number of operands	execution address	RET 1 (A contains 0, or an error number)	Add Ope into area is fo
Rel_to_int	N/A	N/A	RET 1 Overflow bit may be set	Add con in O area
Rel_to_sho	N/A	N/A	RET 1 Overflow bit may be set (A contains error number)	Add con Ope area shou
Sho_to_rel	N/A	N/A	RET 1	Sam
To_system	N/A	N/A	RET 1	Use to p

(Be sure to save the contents of valuable processor registers before calling a utility.

Appendix F
Utilities

Utility Name	LDA with:	LDB with:	Exits	Other	Description	"Minimum R-stack Entries"
Mm_write_start	address of mass storage descriptor	address of storage area	RET 1 — memory overflow RET 2 — normal (A contains mass storage transfer ID)	Mass storage descriptor same as in Mm_read_start. Storage area must be at least 128 words and contain information to be transferred	Writes a physical record to mass storage	5+u*
Mm_write_test	mass storage transfer ID	N/A	RET 1 — transfer incomplete RET 2 — transfer complete (A contains 0, or error number encountered during transfer)	Mass storage transfer ID is returned from Mm_write_start utility.	Verifies a physical record was written to mass storage	5+u*
Printer_select	select-code	printer width	RET 1 (A contains previous printer select code; B contains previous printer width)		Changes or interrogates select-code for standard printer	1
Print_no_lf	address of string	N/A	RET 1 — memory overflow RET 2 — STOP pressed RET 3 — normal	String must be in same for as standard string.	Gives the operating system a chance to complete I/O operations	5+u*
Print_string	address of string	N/A	RET 1 — memory overflow RET 2 — STOP pressed RET 3 — normal	String must be in same form as standard string	Outputs a string to the standard printer	5+u*
Put_bytes	address of storage area	address of parameter	RET 1	Storage area same as Get_bytes	Replaces substrings (or parts of arguments)	2
Put_elem_bytes	address of storage area	address of array info	RET 1	Same as Get_elem_bytes	Same as "Put_bytes", used for accessing elements of string arrays	2
Put_element	address of storage area	address of array info	RET 1	Same as Get_element	Same as "Put_value", used for elements in an array	2
Put_file_info	address of storage area	file number	RET 1 — file unassigned RET 2 — normal	Same as Get_file_info	Manipulates a file-pointer	2
Put_value	address of storage area	address of parameter	RET 1		Changes the value of a BASIC variable	2
Rel_math	number of operands	execution address	RET 1 (A contains 0, or an error number)	Address of first operand into Oper_1 and address of second operand into Oper_2. Address of result area into Result. Execution address is for the desired routine.	Provides access to all the arithmetic routines	5+u*
Rel_to_int	N/A	N/A	RET 1 Overflow bit may be set	Address of the value to be converted should be stored in Oper_1, address of storage area of integer into Result	Data type conversion from full-precision to integer	2
Rel_to_sho	N/A	N/A	RET 1 Overflow bit may be set (A contains error number)	Address of the value to be converted should be stored in Oper_1; address of storage area for converted number should be stored in Result	Data type conversion from full-precision to short	3
Sho_to_rel	N/A	N/A	RET 1	Same as Rel_to_sho	Data type conversion from short-precision to full	2
To_system	N/A	N/A	RET 1	Used within a loop, executed as many times as lines to print. Expedites printing process.	Outputs string to standard printer without carriage-return linefeed sequence.	5+u*

(Be sure to save the contents of valuable processor registers before calling a utility.

*u = the number of levels of JSMs called by the user immediately after the utility is invoked.

Appendix G
Writing Utilities

A utility is a "special" assembly language subroutine. What makes it special is a set of instructions which keeps it from being displayed when a program is being stepped through using the STEP key. By creating a utility, you can make your STEP actions in debugging simpler. If you already know what a section of code does, and don't want to have to step through each instruction in that section each time it is encountered, you can make it into a utility. Then, whenever it is encountered, the section is stepped through as if it were a single statement. The stepping of programs is explained in Chapter 8, Debugging.

The following must be done to make a section of code into a utility —

1. The entry point for the utility must consist of the instruction —

   `ISZ Utlcount`

2. Each exit point from the utility must consist of the following instructions —

   `DSZ Utlcount`
   `RET` n (n may be any number, − 32 through + 31, depending upon the desired returning point)
   `JSM Utlend`

For example, here is a simple utility —

```
10    ISOURCE Temp_result:   BSS 1
20 !                          .
30 !                          .
40    ISOURCE Users:         ISZ Utlcount        ! Utility entry point.
50    ISOURCE                LDA Temp_result        !
60    ISOURCE                SAL 1                  ! Example code segment.
70    ISOURCE                STA Temp_result        !
80    ISOURCE                DSZ Utlcount         !
90    ISOURCE                RET 1                ! Utility exit instructions.
100   ISOURCE                JSM Utlend           !
110 !                         .
120 !                         .
```

The locations Utltemps, Utltemps+1, Utltemps+2, Utltemps+3 and Utltemps+4 are available to you for temporary storage. The absolute addresses of these locations are 177661 through 177665. The locations can be used at any point in your assembly language routine but are most convenient for use within utilities.

System utilities also use the Utltemps locations. If you are calling system utilities from your own utilities, the Utltemps locations should be saved before the system utility call or avoided altogether.

The Utltemps locations as well as the locations Oper_1 and Oper_2 cannot be stepped through for debugging purposes.

It is not required that a utility actually be a subroutine. It may also be in-line code by replacing the RET with JMP *+2.

Utilities, and calls to utilities, are not allowed in interrupt service routines (ISRs).

Appendix H
I/O Sample Programs

Handshake String Output

```
10   ! THIS PROGRAM OUTPUTS A STRING USING HANDSHAKE TO A GPIO-LIKE INTERFACE.
20   !
30   ! INTERFACE CARDS APPLICABLE ARE:
40   !
50   !     98032   16 BIT PARALLEL
60   !     98035   REAL TIME CLOCK
70   !     98036   SERIAL INTERFACE
80   !
90   IDELETE ALL
100  ICOM 1000
110  DIM Input$[160]                     ! ALLOW FOR 160 CHARACTER STRING
120  INTEGER Select_code                 ! BASIC VARIABLE TO HOLD THE SELECT CODE
130  IASSEMBLE
140  INPUT "SELECT CODE TO WRITE TO?",Select_code
150  !
160 Input:  LINPUT "STRING TO WRITE?",Input$ ! ASK USER FOR STRING TO OUTPUT
170  ICALL Output_gpio_hs(Select_code,Input$)
180  GOTO Input
190  !
200       ISOURCE              NAM Output_gpio_hs
210       ISOURCE              EXT Get_value,Error_exit
220       ISOURCE Select_code:BSS 1                ! RESERVED TO HOLD SELECT CODE
230       ISOURCE String:      BSS 81               ! RESERVED FOR 160 CHAR STRING
240       ISOURCE Cr:          EQU 13               ! EQUATES FOR CR/LF
250       ISOURCE Lf:          EQU 10
260       ISOURCE !
270       ISOURCE ! ROUTINE TO OUTPUT A STRING FOLLOWED BY CR/LF TO A GPIO-LIKE
280       ISOURCE ! INTERFACE USING HANDSHAKE.
290       ISOURCE !
300       ISOURCE ! ENTRY POINT:  OUTPUT_gpio_hs
310       ISOURCE !
320       ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
330       ISOURCE !              2)  STRING TO BE OUTPUT
340       ISOURCE !
350       ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
360       ISOURCE !                  164  CARD OR PERIPHERAL DOWN
370       ISOURCE !
380       ISOURCE              SUB
390       ISOURCE Parm_sc:     INT
400       ISOURCE Parm_str:    STR
410       ISOURCE Output_gpio_hs: LDA =Select_code ! GET THE SELECT CODE PARM
420       ISOURCE              LDB =Parm_sc
430       ISOURCE              JSM Get_value
440       ISOURCE              LDA Select_code      ! COPY TO PA
450       ISOURCE              STA Pa
460       ISOURCE              ADA =-1              ! CHECK FOR VALID RANGE (1-14)
470       ISOURCE              SAM Sc_error
480       ISOURCE              ADA =-15+1
490       ISOURCE              SAM Sc_ok
500       ISOURCE Sc_error:    LDA =19              ! GIVE ERROR 19 IF SELECT_CODE
```

```
510     ISOURCE                JSM Error_exit        ! IS OUT OF RANGE
520     ISOURCE !
530     ISOURCE Sc_ok:         LDA =String           ! GET THE STRING PARAMETER
540     ISOURCE                LDB =Parm_str
550     ISOURCE                JSM Get_value
560     ISOURCE                LDA =String+1         ! SET UP C TO GET BYTES FROM
570     ISOURCE                SAL 1                 ! THE STRING
580     ISOURCE                STA C
590     ISOURCE                CBL
600     ISOURCE                LDA String            ! IF THE STRING LENGTH IS ZERO
610     ISOURCE                SZA Done              ! THEN THERE IS NOTHING TO DO.
620     ISOURCE Write_loop:    WBC A,I               ! GET THE NEXT CHAR FOR OUTPUT
630     ISOURCE                JSM Write_byte        ! OUTPUT THE CHARACTER TO CARD
640     ISOURCE                DSZ String            ! SEE IF DONE
650     ISOURCE                  JMP Write_loop      ! IF NOT, REPEAT
660     ISOURCE Done:          LDA =Cr               ! NOW OUTPUT CR/LF
670     ISOURCE                JSM Write_byte
680     ISOURCE                LDA =Lf
690     ISOURCE                JSM Write_byte
700     ISOURCE                RET 1                 ! RETURN TO BASIC
710     ISOURCE !
720     ISOURCE ! SUBROUTINE TO OUTPUT ONE CHARACTER TO GPIO-LIKE CARD.
730     ISOURCE !    CHARACTER IS PASSED IN A
740     ISOURCE !
750     ISOURCE Write_byte:    SSC Card_down         ! SKIP IF CARD IS DOWN
760     ISOURCE                SFC Write_byte        ! ELSE WAIT FOR CARD
770     ISOURCE                STA R4                ! OUTPUT DATA TO CARD
780     ISOURCE                STA R7                ! TRIGGER HANDSHAKE
790     ISOURCE                RET 1
800     ISOURCE !
810     ISOURCE Card_down:     LDA =164              ! RETURN ERROR 164 TO BASIC
820     ISOURCE                JSM Error_exit
830     ISOURCE !
840     ISOURCE                END Output_gpio_hs
```

Handshake String Input

```
10   ! THIS PROGRAM INPUTS A STRING USING HANDSHAKE FROM A GPIO-LIKE DEVICE.
20   !
30   ! INTERFACE CARDS APPLICABLE ARE:
40   !      98032   16 BIT PARALLEL
50   !      98033   BCD
60   !      98035   REAL TIME CLOCK
70   !      98036   SERIAL INTERFACE
80   !
90   IDELETE ALL
100  ICOM 200
110  DIM Input$[160]                    ! ALLOW FOR 160 CHARACTER STRING
120  INTEGER Select_code                ! BASIC VARIABLE TO HOLD THE SELECT CODE
130  IASSEMBLE
140  INPUT "SELECT CODE TO READ FROM?",Select_code
150  !
160  ICALL Read_gpio(Select_code,Input$)
170  PRINT "STRING READ=";Input$
180  END
190  !
200       ISOURCE             NAM Gpio_input
210       ISOURCE             EXT Get_value,Put_value,Error_exit
220       ISOURCE Select_code:BSS 1                ! RESERVED TO HOLD SELECT CODE
230       ISOURCE String:     BSS 81               ! RESERVED FOR 160 CHAR STRING
240       ISOURCE Cr:         EQU 13               ! EQUATES FOR CR/LF
250       ISOURCE Lf:         EQU 10
260       ISOURCE !
270       ISOURCE ! ROUTINE TO INPUT A STRING FOLLOWED BY LF FROM A GPIO-LIKE
280       ISOURCE ! INTERFACE.
290       ISOURCE ! A MAX OF 160 CHARACTERS WILL BE READ. CR'S ARE IGNORED.
300       ISOURCE !
310       ISOURCE ! ENTRY POINT:  READ_gpio
320       ISOURCE !
330       ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
340       ISOURCE !              2)  STRING TO HOLD RESULT
350       ISOURCE !
360       ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
370       ISOURCE !                  164  CARD OR PERIPHERAL DOWN
380       ISOURCE !
390       ISOURCE             SUB
400       ISOURCE Parm_sc:    INT
410       ISOURCE Parm_str:   STR
420       ISOURCE Read_gpio:  LDA =Select_code     ! GET THE SELECT CODE PARM
430       ISOURCE             LDB =Parm_sc
440       ISOURCE             JSM Get_value
450       ISOURCE             LDA Select_code      ! COPY TO PA
460       ISOURCE             STA Pa
470       ISOURCE             ADA =-1              ! CHECK FOR VALID RANGE (1-14)
480       ISOURCE             SAM Sc_error
490       ISOURCE             ADA =-15+1
500       ISOURCE             SAM Sc_ok
510       ISOURCE Sc_error:   LDA =19              ! GIVE ERROR 19 IF SELECT_CODE
520       ISOURCE             JSM Error_exit       ! IS OUT OF RANGE
530       ISOURCE !
540       ISOURCE Sc_ok:      LDA =0               ! INITIALIZE THE STRING LENGTH
550       ISOURCE             STA String
560       ISOURCE             LDA =String          ! SET UP C TO PUT BYTES INTO
570       ISOURCE             SAL 1                ! THE STRING
```

```
580     ISOURCE             ADA =1
590     ISOURCE             STA C
600     ISOURCE             CBL
610     ISOURCE             SSC Card_down        ! SKIP IF CARD/PERIPH ARE DOWN
620     ISOURCE             SFC *-1              ! ELSE WAIT FOR CARD
630     ISOURCE             LDA R4               ! SIGNAL THIS IS AN INPUT
640     ISOURCE Read_loop:  STA R7               ! TRIGGER THE INPUT HANDSHAKE
650     ISOURCE             SFC *                ! WAIT FOR CARD TO COMPLETE
660     ISOURCE             LDA R4               ! THEN GET THE BYTE
670     ISOURCE             CPA =Lf              ! IF LINE FEED
680     ISOURCE              JMP Done            !    THEN WE ARE DONE
690     ISOURCE             CPA =Cr              ! IF CARRIAGE RETURN
700     ISOURCE              JMP Read_loop       ! THEN IGNORE IT
710     ISOURCE             PBC A,I              ! ELSE PUT CHARACTER IN STRING
720     ISOURCE             LDA String           ! AND BUMP STRING LENGTH
730     ISOURCE             ADA =1
740     ISOURCE             STA String
750     ISOURCE             CPA =160             ! HAVE WE INPUT 160 CHARS?
760     ISOURCE              JMP Done            ! YES! SO QUIT NOW
770     ISOURCE             JMP Read_loop        ! IF NOT THEN REPEAT
780     ISOURCE Done:       LDA =String          ! SEND THE STRING TO BASIC
790     ISOURCE             LDB =Parm_str
800     ISOURCE             JSM Put_value
810     ISOURCE             RET 1                ! RETURN TO BASIC
820     ISOURCE !
830     ISOURCE Card_down:  LDA =164             ! RETURN ERROR 164 TO BASIC
840     ISOURCE             JSM Error_exit
850     ISOURCE !
860     ISOURCE             END Gpio_input
```

Interrupt String Output

```
10   ! THIS PROGRAM OUTPUTS A STRING USING INTERRUPT TO A GPIO-LIKE INTERFACE.
20   !
30   ! INTERFACE CARDS APPLICABLE ARE:
40   !
50   !      98032   16 BIT PARALLEL
60   !      98036   SERIAL INTERFACE (INTERRUPT ENABLE BYTE SHOULD BE CHANGED)
70   !
80   IDELETE ALL
90   ICOM 1000
100  DIM Input$[160]                      ! ALLOW FOR 160 CHARACTER STRING
110  INTEGER Select_code                  ! BASIC VARIABLE TO HOLD THE SELECT CODE
120  IASSEMBLE
130  INPUT "SELECT CODE TO WRITE TO?",Select_code
140  ON INT #Select_code GOTO Isr_done   ! SET UP END OF LINE BRANCH
150  !
160 Input:  LINPUT "STRING TO WRITE?",Input$ ! ASK USER FOR STRING TO OUTPUT
170  ICALL Output_gpio_int(Select_code,Input$)
180  !
190  DISP I                               ! DO OTHER WORK WHILE INTERRUPT
200  I=I+1                                ! OUTPUT IS IN PROGRESS
210  GOTO 190
220  !
230 Isr_done:  DISP " OUTPUT COMPLETE...NEXT "; ! GET HERE WHEN ISR OUTPUT IS
240  GOTO Input                           ! COMPLETE...SO REPEAT
250  !
260       ISOURCE              NAM Output_gpio_int
270       ISOURCE              EXT Get_value,Error_exit,Isr_access
280       ISOURCE Select_code:BSS 1                 ! RESERVED TO HOLD SELECT CODE
290       ISOURCE String:      BSS 81               ! RESERVED FOR 160 CHAR STRING
300       ISOURCE Byte_pointer:BSS 1                ! BYTE POINTER FOR ISR
310       ISOURCE Eol_mask:    BSS 1                ! TEMP FOR ISR
320       ISOURCE Save35:      BSS 1                ! TEMP FOR ISR
330       ISOURCE Cr:          EQU 13               ! EQUATES FOR CR/LF
340       ISOURCE Lf:          EQU 10
350       ISOURCE Enable_mask:EQU 200B              ! 98032 INTERRUPT ENABLE MASK
360       ISOURCE !
370       ISOURCE ! ROUTINE TO OUTPUT A STRING FOLLOWED BY CR/LF TO A GPIO-LIKE
380       ISOURCE ! INTERFACE USING INTERRUPT.
390       ISOURCE !
400       ISOURCE ! ENTRY POINT:  Output_gpio_int
410       ISOURCE !
420       ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
430       ISOURCE !              2)  STRING TO BE OUTPUT
440       ISOURCE !
450       ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
460       ISOURCE !                  164  CARD OR PERIPHERAL DOWN
470       ISOURCE !
480       ISOURCE              SUB
490       ISOURCE Parm_sc:     INT
500       ISOURCE Parm_str:    STR
510       ISOURCE Output_gpio_int:LDA =Select_code ! GET THE SELECT CODE PARM
520       ISOURCE              LDB =Parm_sc
530       ISOURCE              JSM Get_value
540       ISOURCE              LDA Select_code      ! LOAD A WITH SELECT CODE
550       ISOURCE              ADA =-1              ! CHECK FOR VALID RANGE (1-14)
560       ISOURCE              SAM Sc_error
570       ISOURCE              ADA =-15+1
```

```
580    ISOURCE              SAM Sc_ok
590    ISOURCE Sc_error:    LDA =19              ! GIVE ERROR 19 IF SELECT_CODE
600    ISOURCE              JSM Error_exit       ! IS OUT OF RANGE
610    ISOURCE !
620    ISOURCE Sc_ok:       LDA Select_code      ! SEE IF CARD IS OK
630    ISOURCE              STA Pa               ! FIRST COPY SELECT CODE TO PA
640    ISOURCE              SSC Card_down        ! SKIP IF DOWN
650    ISOURCE              LDA =Isr             ! SET UP AN ISR
660    ISOURCE              LDB =(10*256)+(1*16)! 10 TRIES, RESOURCE=1=ASYNC
670    ISOURCE              ADB Select_code
680    ISOURCE              JSM Isr_access
690    ISOURCE               JMP Sc_ok           ! IF COULDN'T GET IT, RETRY
700    ISOURCE              LDA =String          ! GET THE STRING PARAMETER
710    ISOURCE              LDB =Parm_str
720    ISOURCE              JSM Get_value
730    ISOURCE              LDA =String+1        ! SET UP BYTE POINTER FOR ISR
740    ISOURCE              SAL 1                ! TO GET CHARS FROM STRING
750    ISOURCE              STA Byte_pointer
760    ISOURCE              ADA String           ! ADD CR/LF TO END OF STRING
770    ISOURCE              ADA =-1
780    ISOURCE              STA C
790    ISOURCE              CBL
800    ISOURCE              LDA =Cr
810    ISOURCE              PBC A,I
820    ISOURCE              LDA =Lf
830    ISOURCE              PBC A,I
840    ISOURCE              LDA String           ! BE SURE AND ADD 2 TO LENGTH
850    ISOURCE              ADA =2               ! SO ISR WILL OUTPUT CR/LF
860    ISOURCE              STA String
870    ISOURCE              LDA =Enable_mask     ! ENABLE THE CARD TO INTERRUPT
880    ISOURCE              STA R5
890    ISOURCE              RET 1                ! GO BACK TO BASIC.
900    ISOURCE !
910    ISOURCE Card_down:   LDA =164
920    ISOURCE              JSM Error_exit
930    ISOURCE !
940    ISOURCE Isr:         LDA 35B              ! SINCE I AM GOING TO DO STACK
950    ISOURCE              STA Save35           ! OPERATIONS, I MUST SAVE 35
960    ISOURCE              LDA 34B              ! AND INITIALIZE IT
970    ISOURCE              STA 35B
980    ISOURCE              LDA Byte_pointer     ! SET UP THE BYTE POINTER
990    ISOURCE              STA C                ! SO I CAN GET A DATA BYTE
1000   ISOURCE              CBL
1010   ISOURCE              WBC R4,I             ! SEND THE DATA BYTE TO CARD
1020   ISOURCE              STA R7               ! DO HANDSHAKE
1030   ISOURCE              LDA C                ! RESAVE BYTE POINTER
1040   ISOURCE              STA Byte_pointer
1050   ISOURCE              DSZ String           ! SEE IF DONE
1060   ISOURCE               JMP Exit            ! IF NOT, THEN EXIT THE ISR
1070   ISOURCE              LDA =0               ! DISABLE THE CARD
1080   ISOURCE              STA R5
1090   ISOURCE              LDA Pa               ! DEPENDING ON WHETHER THE
1100   ISOURCE              ADA =-8              ! SELECT CODE IS HIGH, OR LOW
1110   ISOURCE              SAP *+3              ! CALL THE CORRECT TERMINATION
1120   ISOURCE              JSM End_isr_low,I    ! ROUTINE
1130   ISOURCE              JMP *+2
1140   ISOURCE              JSM End_isr_high,I
1150   ISOURCE              LDA Pa               ! AND NOW TRIGGER AN END OF
1160   ISOURCE              ADA =-1              ! LINE BRANCH. TO DO THIS, THE
1170   ISOURCE              IOR Sb11             ! CORRECT MASK WORD MUST BE
```

```
1180      ISOURCE              LDB =1              ! CALCULATED BY A COMPUTED
1190      ISOURCE              EXE A               ! SHIFT INSTRUCTION
1200      ISOURCE              STB Eol_mask        ! SAVE THIS MASK
1210      ISOURCE              LDB Isr_psw         ! AND USE MAGIC CODE TO
1220      ISOURCE              LDA =103B           ! TRIGGER THE EOL BRANCH
1230      ISOURCE              STA B,I
1240      ISOURCE              ADB =3
1250      ISOURCE              LDA Eol_mask
1260      ISOURCE              DIR
1270      ISOURCE              IOR B,I
1280      ISOURCE              STA B,I
1290      ISOURCE              EIR
1300      ISOURCE Exit:        LDA Save35          ! RESTORE 35
1310      ISOURCE              STA 35B
1320      ISOURCE              RET 1               ! RETURN FROM INTERRUPT
1330      ISOURCE Sbl1:        SBL 1               ! BIT MASK FOR INSTRUCTION
1340      ISOURCE !
1350      ISOURCE              END Output_gpio_int
```

Interrupt String Input

```
10   ! THIS PROGRAM INPUTS A STRING USING INTERRUPT FROM A GPIO-LIKE INTERFACE.
20   !
30   ! INTERFACE CARDS APPLICABLE ARE:
40   !
50   !     98032   16 BIT PARALLEL
60   !     98033   BCD
70   !     98036   SERIAL INTERFACE (INTERRUPT ENABLE BYTE SHOULD BE CHANGED)
80   !
90   IDELETE ALL
100  ICOM 1000
110  DIM Input$[160]                    ! ALLOW FOR 160 CHARACTER STRING
120  INTEGER Select_code                ! BASIC VARIABLE TO HOLD THE SELECT CODE
130  IASSEMBLE
140  INPUT "SELECT CODE TO READ FROM?",Select_code
150  ON INT #Select_code GOTO Isr_done   ! SET UP END OF LINE BRANCH
160  ICALL Enter_gpio_int(Select_code)  ! START THE READ OPERATION
170  !
180  ICALL Read_result(Input$)          ! WHILE WAITING FOR IT TO COMPLETE,
190  DISP "PARTIAL RESULT=";Input$      ! DISPLAY THE PARTIAL RESULTS
200  GOTO 180
210  !
220 Isr_done:  ICALL Read_result(Input$)
230  DISP " INPUT COMPLETE...STRING=";Input$
240  END
250  !
260       ISOURCE              NAM Enter_gpio_int
270       ISOURCE              EXT Get_value,Put_value,Error_exit,Isr_access
280       ISOURCE Select_code:BSS 1                 ! RESERVED TO HOLD SELECT CODE
290       ISOURCE String:      BSS 81               ! RESERVED FOR 160 CHAR STRING
300       ISOURCE Byte_pointer:BSS 1                ! BYTE POINTER FOR ISR
310       ISOURCE Eol_mask:    BSS 1                ! TEMP FOR ISR
320       ISOURCE Save35:      BSS 1                ! TEMP FOR ISR
330       ISOURCE Cr:          EQU 13               ! EQUATES FOR CR/LF
340       ISOURCE Lf:          EQU 10
350       ISOURCE Enable_mask:EQU 200B              ! 98032 INTERRUPT ENABLE MASK
360       ISOURCE !
```

```
370   ISOURCE ! ROUTINES TO INPUT A STRING FOLLOWED BY LF FROM A GPIO-LIKE
380   ISOURCE ! INTERFACE USING INTERRUPT.
390   ISOURCE !
400   ISOURCE ! ENTRY POINT:  Enter_gpio_int
410   ISOURCE !
420   ISOURCE ! PARAMETER:   1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
430   ISOURCE !
440   ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
450   ISOURCE !                  164  CARD OR PERIPHERAL DOWN
460   ISOURCE !
470   ISOURCE ! ENTRY POINT:  Read_result
480   ISOURCE !
490   ISOURCE ! PARAMETER:   1)  STRING TO CONTAIN THE INPUT DATA
500   ISOURCE !
510   ISOURCE                SUB
520   ISOURCE Parm_sc:       INT
530   ISOURCE Enter_gpio_int: LDA =Select_code ! GET THE SELECT CODE PARM
540   ISOURCE                LDB =Parm_sc
550   ISOURCE                JSM Get_value
560   ISOURCE                LDA Select_code     ! LOAD A WITH SELECT CODE
570   ISOURCE                ADA =-1             ! CHECK FOR VALID RANGE (1-14)
580   ISOURCE                SAM Sc_error
590   ISOURCE                ADA =-15+1
600   ISOURCE                SAM Sc_ok
610   ISOURCE Sc_error:      LDA =19             ! GIVE ERROR 19 IF SELECT_CODE
620   ISOURCE                JSM Error_exit      ! IS OUT OF RANGE
630   ISOURCE !
640   ISOURCE Sc_ok:         LDA Select_code     ! SEE IF CARD IS OK
650   ISOURCE                STA Pa              ! FIRST COPY SELECT CODE TO PA
660   ISOURCE                SSC Card_down       ! SKIP IF DOWN
670   ISOURCE                LDA =Isr            ! SET UP AN ISR
680   ISOURCE                LDB =(10*256)+(1*16)! 10 TRIES, RESOURCE=1=ASYNC
690   ISOURCE                ADB Select_code
700   ISOURCE                JSM Isr_access
710   ISOURCE                 JMP Sc_ok          ! IF COULDN'T GET IT, RETRY
720   ISOURCE                LDA =0              ! INITIALIZE BYTE COUNT OF
730   ISOURCE                STA String          ! STRING BUFFER AREA
740   ISOURCE                LDA =String         ! SET UP BYTE POINTER FOR ISR
750   ISOURCE                SAL 1               ! TO PUT CHARS INTO STRING
760   ISOURCE                ADA =1
770   ISOURCE                STA Byte_pointer
780   ISOURCE                SFC *               ! WAIT FOR CARD
790   ISOURCE                LDA R4              ! START FIRST INPUT OPERATION
800   ISOURCE                STA R7
810   ISOURCE                LDA =Enable_mask    ! ENABLE THE CARD TO INTERRUPT
820   ISOURCE                STA R5
830   ISOURCE                RET 1               ! GO BACK TO BASIC.
840   ISOURCE !
850   ISOURCE Card_down:     LDA =164
860   ISOURCE                JSM Error_exit
870   ISOURCE !
880   ISOURCE                SUB
890   ISOURCE Parm_str:      STR
900   ISOURCE Read_result:LDA =String
910   ISOURCE                LDB =Parm_str
920   ISOURCE                JSM Put_value
930   ISOURCE                RET 1
940   ISOURCE !
950   ISOURCE Isr:           LDA 35B             ! SINCE I AM GOING TO DO STACK
960   ISOURCE                STA Save35          ! OPERATIONS, I MUST SAVE 35
970   ISOURCE                LDA 34B             ! AND INITIALIZE IT
```

```
980     ISOURCE                STA 35B
990     ISOURCE                LDA Byte_pointer     ! SET UP THE BYTE POINTER
1000    ISOURCE                STA C                ! SO I CAN PUT A DATA BYTE
1010    ISOURCE                CBL                  ! INTO THE STRING
1020    ISOURCE                LDA R4               ! GET THE NEXT CHARACTER FROM
1030    ISOURCE                CPA =Cr              ! THEN CARD...IGNORE CR'S
1040    ISOURCE                 JMP Do_another
1050    ISOURCE                CPA =Lf              ! IF LINE FEED, THE TERMINATE
1060    ISOURCE                 JMP Terminate       ! THE ISR TRANSFER
1070    ISOURCE                PBC A,I              ! ELSE PUT CHARACTER IN STRING
1080    ISOURCE                LDA C                ! SAVE NEW BYTE POINTER
1090    ISOURCE                STA Byte_pointer
1100    ISOURCE                LDA String           ! AND BUMP STRING LENGTH
1110    ISOURCE                ADA =1
1120    ISOURCE                STA String
1130    ISOURCE                CPA =160             ! HAVE WE RECEIVED 160 CHARS
1140    ISOURCE                 JMP Terminate       ! IF YES, THEN SHUT DOWN
1150    ISOURCE Do_another:    STA R7               ! START ANOTHER HANDSHAKE
1160    ISOURCE                JMP Exit             ! THEN EXIT THE ISR
1170    ISOURCE !
1180    ISOURCE Terminate:     LDA =0               ! DISABLE THE CARD
1190    ISOURCE                STA R5
1200    ISOURCE                LDA Pa               ! DEPENDING ON WHETHER THE
1210    ISOURCE                ADA =-8              ! SELECT CODE IS HIGH, OR LOW
1220    ISOURCE                SAP *+3              ! CALL THE CORRECT TERMINATION
1230    ISOURCE                JSM End_isr_low,I    ! ROUTINE
1240    ISOURCE                JMP *+2
1250    ISOURCE                JSM End_isr_high,I
1260    ISOURCE                LDA Pa               ! AND NOW TRIGGER AN END OF
1270    ISOURCE                ADA =-1              ! LINE BRANCH. TO DO THIS, THE
1280    ISOURCE                IOR Sbl1             ! CORRECT MASK WORD MUST BE
1290    ISOURCE                LDB =1               ! CALCULATED BY A COMPUTED
1300    ISOURCE                EXE A                ! SHIFT INSTRUCTION
1310    ISOURCE                STB Eol_mask         ! SAVE THIS MASK
1320    ISOURCE                LDB Isr_psw          ! AND USE MAGIC CODE TO
1330    ISOURCE                LDA =103B            ! TRIGGER THE EOL BRANCH
1340    ISOURCE                STA B,I
1350    ISOURCE                ADB =3
1360    ISOURCE                LDA Eol_mask
1370    ISOURCE                DIR
1380    ISOURCE                IOR B,I
1390    ISOURCE                STA B,I
1400    ISOURCE                EIR
1410    ISOURCE Exit:          LDA Save35           ! RESTORE 35
1420    ISOURCE                STA 35B
1430    ISOURCE                RET 1                ! RETURN FROM INTERRUPT
1440    ISOURCE Sbl1:          SBL 1                ! BIT MASK FOR INSTRUCTION
1450    ISOURCE !
1460    ISOURCE                END Enter_gpio_int
```

DMA String Output

```
10   ! THIS PROGRAM OUTPUTS A STRING USING DMA TO A GPIO INTERFACE.
20   !
30   ! INTERFACE CARDS APPLICABLE ARE:
40   !
50   !     98032   16 BIT PARALLEL
60   !
70   IDELETE ALL
80   ICOM 1000
90   DIM Input$[160]                   ! ALLOW FOR 160 CHARACTER STRING
100  INTEGER Select_code               ! BASIC VARIABLE TO HOLD THE SELECT CODE
110  IASSEMBLE
120  INPUT "SELECT CODE TO WRITE TO?",Select_code
130  ON INT #Select_code GOTO Dma_done  ! SET UP END OF LINE BRANCH
140  !
150 Input:  LINPUT "STRING TO WRITE?",Input$ ! ASK USER FOR STRING TO OUTPUT
160  ICALL Output_gpio_dma(Select_code,Input$)
170  !
180  DISP I                            ! DO OTHER WORK WHILE INTERRUPT
190  I=I+1                             ! OUTPUT IS IN PROGRESS
200  GOTO 180
210  !
220 Dma_done:  DISP " OUTPUT COMPLETE...NEXT "; ! GET HERE WHEN ISR OUTPUT IS
230  GOTO Input                        ! COMPLETE...SO REPEAT
240  !
250       ISOURCE              NAM Output_gpio_dma
260       ISOURCE              EXT Get_value,Error_exit,Isr_access
270       ISOURCE Select_code:BSS 1                  ! RESERVED TO HOLD SELECT CODE
280       ISOURCE String:      BSS 81                ! RESERVED FOR 160 CHAR STRING
290       ISOURCE              BSS 80                ! RESERVED TO EXPAND STRING
300       ISOURCE Eol_mask:    BSS 1                 ! TEMP FOR ISR
310       ISOURCE Save35:      BSS 1                 ! TEMP FOR ISR
320       ISOURCE Cr:          EQU 13                ! EQUATES FOR CR/LF
330       ISOURCE Lf:          EQU 10
340       ISOURCE Enable_mask:EQU 320B               ! 98032 DMA/INT/AH ENABLE MASK
350       ISOURCE !
360       ISOURCE ! ROUTINE TO OUTPUT A STRING FOLLOWED BY CR/LF TO A GPIO-LIKE
370       ISOURCE ! INTERFACE USING DMA.
380       ISOURCE !
390       ISOURCE ! ENTRY POINT:  Output_gpio_dma
400       ISOURCE !
410       ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
420       ISOURCE !              2)  STRING TO BE OUTPUT
430       ISOURCE !
440       ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
450       ISOURCE !                  164  CARD OR PERIPHERAL DOWN
460       ISOURCE !
470       ISOURCE              SUB
480       ISOURCE Parm_sc:     INT
490       ISOURCE Parm_str:    STR
500       ISOURCE Output_gpio_dma:LDA =Select_code ! GET THE SELECT CODE PARM
510       ISOURCE              LDB =Parm_sc
520       ISOURCE              JSM Get_value
530       ISOURCE              LDA Select_code      ! LOAD A WITH SELECT CODE
540       ISOURCE              ADA =-1              ! CHECK FOR VALID RANGE (1-14)
550       ISOURCE              SAM Sc_error
560       ISOURCE              ADA =-15+1
570       ISOURCE              SAM Sc_ok
580       ISOURCE Sc_error:    LDA =19              ! GIVE ERROR 19 IF SELECT_CODE
590       ISOURCE              JSM Error_exit       ! IS OUT OF RANGE
```

```
600      ISOURCE !
610      ISOURCE Sc_ok:      LDA Select_code      ! SEE IF CARD IS OK
620      ISOURCE             STA Pa               ! FIRST COPY SELECT CODE TO PA
630      ISOURCE             SSS Card_ok          ! SKIP IF CARD IS UP
640      ISOURCE             LDA =164             ! ELSE GIVE ERROR 164
650      ISOURCE             JSM Error_exit
660      ISOURCE !
670      ISOURCE Card_ok:    LDA =Isr             ! SET UP AN ISR
680      ISOURCE             LDB =(10*256)+(2*16)! 10 TRIES, RESOURCE=2=DMA
690      ISOURCE             ADB Select_code
700      ISOURCE             JSM Isr_access
710      ISOURCE              JMP Sc_ok           ! IF COULDN'T GET IT, RETRY
720      ISOURCE             LDA =String          ! GET THE STRING PARAMETER
730      ISOURCE             LDB =Parm_str
740      ISOURCE             JSM Get_value
750      ISOURCE ! FOR DMA, THE NORMAL STRING FORMAT WON'T DO. THE DATA MUST
760      ISOURCE ! BE STORED ONE BYTE PER WORD, SO THE FOLLOWING LOOP WILL
770      ISOURCE ! EXPAND THE STRING AND ADD A CR/LF
780      ISOURCE             LDA =String+1        ! FIRST SET UP BYTE POINTER TO
790      ISOURCE             SAL 1                ! WITHDRAW THE LAST CHARACTER
800      ISOURCE             ADA String           ! FIRST
810      ISOURCE             ADA =-1
820      ISOURCE             STA C                ! USE C FOR THE BYTE POINTER
830      ISOURCE             CBL
840      ISOURCE             LDA =String+3        ! NOW COMPUTE A WORD POINTER
850      ISOURCE             ADA String           ! TO WHERE TO PLACE THE LF
860      ISOURCE             STA D
870      ISOURCE             LDA =Lf              ! MOVE IN A LF
880      ISOURCE             PWD A,D
890      ISOURCE             LDA =Cr              ! AND CR
900      ISOURCE             PWD A,D
910      ISOURCE             LDB String           ! NOW LOOP TO COPY ALL BYTES
920      ISOURCE             TCB
930      ISOURCE             SIB *+4
940      ISOURCE             WBC A,D
950      ISOURCE             PWD A,D
960      ISOURCE             RIB *-2
970      ISOURCE             LDA String           ! SET UP DMA CONTROL REGISTERS
980      ISOURCE             ADA =1               ! COUNT = #CHARS-1
990      ISOURCE             STA Dmac
1000     ISOURCE             LDA =String+1        ! DMAA = DATA ADDRESS
1010     ISOURCE             STA Dmama
1020     ISOURCE             SDO                  ! SET DMA OUTWARDS
1030     ISOURCE             LDA =Enable_mask     ! ENABLE THE CARD TO INTERRUPT
1040     ISOURCE             STA R5
1050     ISOURCE             DMA
1060     ISOURCE             RET 1                ! GO BACK TO BASIC.
1070     ISOURCE !
1080     ISOURCE Isr:        LDA 35B              ! I WILL GET THE INTERRUPT
1090     ISOURCE             STA Save35           ! THE DMA TRANSFER IS COMPLETE
1100     ISOURCE             LDA 34B
1110     ISOURCE             STA 35B
1120     ISOURCE             LDA =0               ! SO DISABLE THE CARD
1130     ISOURCE             STA R5
1140     ISOURCE             DDR                  ! DISABLE DMA
1150     ISOURCE             LDA Pa               ! DEPENDING ON WHETHER THE
1160     ISOURCE             ADA =-8              ! SELECT CODE IS HIGH, OR LOW
1170     ISOURCE             SAP *+3              ! CALL THE CORRECT TERMINATION
1180     ISOURCE             JSM End_isr_low,I    ! ROUTINE
1190     ISOURCE             JMP *+2
1200     ISOURCE             JSM End_isr high,I
```

```
1210      ISOURCE              LDA Pa              ! AND NOW TRIGGER AN END OF
1220      ISOURCE              ADA =-1             ! LINE BRANCH. TO DO THIS, THE
1230      ISOURCE              IOR Sbl1            ! CORRECT MASK WORD MUST BE
1240      ISOURCE              LDB =1              ! CALCULATED BY A COMPUTED
1250      ISOURCE              EXE A               ! SHIFT INSTRUCTION
1260      ISOURCE              STB Eol_mask        ! SAVE THIS MASK
1270      ISOURCE              LDB Isr_psw         ! AND USE MAGIC CODE TO
1280      ISOURCE              LDA =103B           ! TRIGGER THE EOL BRANCH
1290      ISOURCE              STA B,I
1300      ISOURCE              ADB =3
1310      ISOURCE              LDA Eol_mask
1320      ISOURCE              DIR
1330      ISOURCE              IOR B,I
1340      ISOURCE              STA B,I
1350      ISOURCE              EIR
1360      ISOURCE              LDA Save35          ! RESTORE 35
1370      ISOURCE              STA 35B
1380      ISOURCE              RET 1               ! RETURN FROM INTERRUPT
1390      ISOURCE Sbl1:        SBL 1               ! BIT MASK FOR INSTRUCTION
1400      ISOURCE !
1410      ISOURCE              END Output_gpio_dma
```

DMA String Input

```
10   ! THIS PROGRAM INPUTS A STRING USING DMA FROM A GPIO INTERFACE.
20   !
30   ! INTERFACE CARDS APPLICABLE ARE:
40   !
50   !      98032   16 BIT PARALLEL
60   !
70   IDELETE ALL
80   ICOM 1000
90   DIM Input$[160]                   ! ALLOW FOR 160 CHARACTER STRING
100  INTEGER Select_code               ! BASIC VARIABLE TO HOLD THE SELECT CODE
110  INTEGER Character_count           ! VARIABLE TO HOLD INPUT CHARACTER COUNT
120  INTEGER A,C                       ! VARIABLES FOR "BACKGROUND PROCESS"
130  IASSEMBLE
140  INPUT "SELECT CODE TO READ FROM?",Select_code
150  ON INT #Select_code GOTO Isr_done  ! SET UP END OF LINE BRANCH
160  INPUT "NUMBER OF CHARACTERS TO READ?",Character_count
170  ICALL Enter_gpio_dma(Select_code,Character_count)  ! START THE READ
180  !
190  ICALL Test_dma(C,A)               ! WHILE WAITING, DISPLAY DMA COUNT AND
200  DISP "DMA COUNT=";C,"ADDRESS=";A,I ! ADDRESS
210  I=I+1
220  GOTO 190
230  !
240 Isr_done:  ICALL Read_result(Input$)
250  DISP " INPUT COMPLETE...STRING=";Input$
260  END
270  !
280       ISOURCE              NAM Enter_gpio_dma
290       ISOURCE              EXT Get_value,Put_value,Error_exit,Isr_access
300       ISOURCE Select_code:BSS 1                ! RESERVED TO HOLD SELECT CODE
310       ISOURCE String:      BSS 81              ! RESERVED FOR 160 CHAR STRING
320       ISOURCE              BSS 80              ! RESERVED FOR EXPANDED STRING
330       ISOURCE Eol_mask:    BSS 1               ! TEMP FOR ISR
340       ISOURCE Save35:      BSS 1               ! TEMP FOR ISR
```

```
350       ISOURCE Enable_mask:EQU 320B           ! 98032 DMA/INT/AH ENABLE MASK
360       ISOURCE !
370       ISOURCE ! ROUTINES TO INPUT A FIXED LENGTH STRING FROM A GPIO
380       ISOURCE ! INTERFACE USING DMA.
390       ISOURCE !
400       ISOURCE ! ENTRY POINT:  Enter_gpio_dma
410       ISOURCE !
420       ISOURCE ! PARAMETER:    1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
430       ISOURCE !               2)  NUMBER OF CHARACTERS TO READ ( 1 TO 80 )
440       ISOURCE !
450       ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OR CHAR COUNT OUT OF RANGE
460       ISOURCE !                   164  CARD OR PERIPHERAL DOWN
470       ISOURCE !
480       ISOURCE ! ENTRY POINT:  Test_dma
490       ISOURCE !
500       ISOURCE ! PARAMETERS:   1)  INTEGER TO HOLD CURRENT DMA COUNT
510       ISOURCE !               2)  INTEGER TO HOLD CURRENT DMA ADDRRESS
520       ISOURCE !
530       ISOURCE ! ENTRY POINT:  Read_result
540       ISOURCE !
550       ISOURCE ! PARAMETER:    1)  STRING TO CONTAIN THE INPUT DATA
560       ISOURCE !
570       ISOURCE               SUB
580       ISOURCE Parm_sc:      INT
590       ISOURCE Parm_count:   INT
600       ISOURCE Enter_gpio_dma: LDA =Select_code ! GET THE SELECT CODE PARM
610       ISOURCE               LDB =Parm_sc
620       ISOURCE               JSM Get_value
630       ISOURCE               LDA Select_code      ! LOAD A WITH SELECT CODE
640       ISOURCE               ADA =-1              ! CHECK FOR VALID RANGE (1-14)
650       ISOURCE               SAM Sc_error
660       ISOURCE               ADA =-15+1
670       ISOURCE               SAM Sc_ok
680       ISOURCE Sc_error:     LDA =19              ! GIVE ERROR 19 IF SELECT_CODE
690       ISOURCE               JSM Error_exit       ! IS OUT OF RANGE
700       ISOURCE !
710       ISOURCE Sc_ok:        LDA =String          ! GET BYTE COUNT PARAMETER
720       ISOURCE               LDB =Parm_count
730       ISOURCE               JSM Get_value
740       ISOURCE               LDA String           ! CHECK IT FOR RANGE
750       ISOURCE               SAM Sc_error
760       ISOURCE               SZA Sc_error
770       ISOURCE               ADA =-81
780       ISOURCE               SAP Sc_error
790       ISOURCE Check_card:   LDA Select_code      ! SEE IF CARD IS OK
800       ISOURCE               STA Pa               ! FIRST COPY SELECT CODE TO PA
810       ISOURCE               SSS Card_ok          ! SKIP IF CARD IS OK
820       ISOURCE               LDA =164             ! ELSE GIVE ERROR 164
830       ISOURCE               JSM Error_exit
840       ISOURCE !
850       ISOURCE Card_ok:      LDA =Isr             ! SET UP AN ISR
860       ISOURCE               LDB =(10*256)+(2*16)! 10 TRIES, RESOURCE=2=DMA
870       ISOURCE               ADB Select_code
880       ISOURCE               JSM Isr_access
890       ISOURCE                JMP Check_card      ! IF COULDN'T GET IT, RETRY
900       ISOURCE               LDA String           ! INITIALIZE DMA REGISTERS
910       ISOURCE               ADA =-1
920       ISOURCE               STA Dmac
930       ISOURCE               LDA =String+1
940       ISOURCE               STA Dmama
950       ISOURCE               SDI
```

```
960    ISOURCE                SFC *                ! WAIT FOR CARD
970    ISOURCE                LDA R4               ! START FIRST INPUT OPERATION
980    ISOURCE                STA R7
990    ISOURCE                LDA =Enable_mask     ! ENABLE THE CARD TO INTERRUPT
1000   ISOURCE                STA R5
1010   ISOURCE                DMA                  ! ENABLE PROCESSER FOR DMA
1020   ISOURCE                RET 1                ! GO BACK TO BASIC.
1030   ISOURCE !
1040   ISOURCE                SUB
1050   ISOURCE C_parm:        INT
1060   ISOURCE A_parm:        INT
1070   ISOURCE Test_dma:      LDA Dmac
1080   ISOURCE                STA Temp
1090   ISOURCE                LDA =Temp
1100   ISOURCE                LDB =C_parm
1110   ISOURCE                JSM Put_value
1120   ISOURCE                LDA Dmama
1130   ISOURCE                STA Temp
1140   ISOURCE                LDA =Temp
1150   ISOURCE                LDB =A_parm
1160   ISOURCE                JSM Put_value
1170   ISOURCE                RET 1
1180   ISOURCE Temp:          BSS 1
1190   ISOURCE !
1200   ISOURCE                SUB
1210   ISOURCE Parm_str:      STR
1220    ISOURCE Read_result:LDA =String+1          ! I MUST PACK THE STRING FROM
1230   ISOURCE                STA D                ! FROM 1 BYTE TO 2 BYTES PER
1240   ISOURCE                SAL 1
1250   ISOURCE                ADA =-1
1260   ISOURCE                STA C
1270   ISOURCE                CBL
1280   ISOURCE                LDA String           ! GET CHARACTER COUNT
1290   ISOURCE                TCA
1300   ISOURCE                SIA *+4
1310   ISOURCE                WWD B,I              ! GET A BYTE
1320   ISOURCE                PBC B,I              ! PACK IT
1330   ISOURCE                RIA *-2
1340   ISOURCE                LDA =String          ! RETURN RESULT TO BASIC
1350   ISOURCE                LDB =Parm_str
1360   ISOURCE                JSM Put_value
1370   ISOURCE                RET 1
1380   ISOURCE !
1390   ISOURCE Isr:           LDA 35B              ! I WILL GET AN INTERRUPT WHEN
1400   ISOURCE                STA Save35           ! THE DMA IS COMPLETE
1410   ISOURCE                LDA 34B
1420   ISOURCE                STA 35B
1430   ISOURCE                LDA Dmac             ! I GET TO HERE WHEN DMA DONE
1440   ISOURCE                ADA =1               ! COMPUTE ACTUAL NUMBER OF
1450   ISOURCE                TCA                  ! CHARACTERS TRANSFERED
1460   ISOURCE                ADA String
1470   ISOURCE                STA String           ! SAVE IN STRING LENGTH WORD
1480   ISOURCE                LDA =0               ! DISABLE THE CARD
1490   ISOURCE                STA R5
1500   ISOURCE                DDR                  ! DISABLE DMA
1510   ISOURCE                LDA Pa               ! DEPENDING ON WHETHER THE
1520   ISOURCE                ADA =-8              ! SELECT CODE IS HIGH, OR LOW
1530   ISOURCE                SAP *+3              ! CALL THE CORRECT TERMINATION
1540   ISOURCE                JSM End_isr_low,I    ! ROUTINE
1550   ISOURCE                JMP *+2
1560   ISOURCE                JSM End_isr_high,I
```

```
1570    ISOURCE             LDA Pa              ! AND NOW TRIGGER AN END OF
1580    ISOURCE             ADA =-1             ! LINE BRANCH. TO DO THIS, THE
1590    ISOURCE             IOR Sbl1            ! CORRECT MASK WORD MUST BE
1600    ISOURCE             LDB =1              ! CALCULATED BY A COMPUTED
1610    ISOURCE             EXE A               ! SHIFT INSTRUCTION
1620    ISOURCE             STB Eol_mask        ! SAVE THIS MASK
1630    ISOURCE             LDB Isr_psw         ! AND USE MAGIC CODE TO
1640    ISOURCE             LDA =103B           ! TRIGGER THE EOL BRANCH
1650    ISOURCE             STA B,I
1660    ISOURCE             ADB =3
1670    ISOURCE             LDA Eol_mask
1680    ISOURCE             DIR
1690    ISOURCE             IOR B,I
1700    ISOURCE             STA B,I
1710    ISOURCE             EIR
1720    ISOURCE             LDA Save35
1730    ISOURCE             STA 35B
1740    ISOURCE             RET 1               ! RETURN FROM INTERRUPT
1750    ISOURCE Sbl1:       SBL 1               ! BIT MASK FOR INSTRUCTION
1760    ISOURCE !
1770    ISOURCE             END Enter_gpio_dma
```

HP-IB Output/Input Drivers

```
10   ! 98034A HPIB CARD DRIVER
20   !
30   ON KEY #0 GOSUB Output
40   ON KEY #1 GOSUB Enter
50   ON KEY #6 GOTO Last
60   PRINT " HPIB DRIVER"
70   PRINT
80   PRINT
90   PRINT "TWO ASSEMBLY LANGUAGE DRIVERS ARE PROVIDED...ONE FOR OUTPUT AND ONE"
100  PRINT "FOR INPUT.  BOTH HAVE PROVISIONS FOR INCLUDING A BUS COMMAND STRING"
110  PRINT "FOR ADDRESSING THE BUS."
120  !
130  PRINT "SYNTAX:"
140  !
150  PRINT "ICALL Hpib_output( <ISC>, <CMD$>, [ <DATA$> ] )"
160  PRINT "ICALL Hpib_enter ( <ISC>, <CMD$>, [ <VAR$> ] )"
170  PRINT
180  PRINT "   <ISC>   ::= INTERFACE SELECT CODE (1 TO 14) (INTEGER)"
190  PRINT "   <CMD$>  ::= STRING TO OUTPUT WITH ATN TRUE"
200  PRINT "   <DATA$> ::= STRING TO OUTPUT WITH ATN FALSE"
210  PRINT "   <VAR$>  ::= STRING VARIABLE TO HOLD DATA READ FROM BUS"
220  PRINT LIN(5);"Press key #6 to exit."
230  DISP "Press CONTINUE to execute program"
240  PAUSE
245  ICOM 1000
250  !   POSSIBLE ERRORS:
260  !
270  !      164   CARD WAS NOT AN HPIB CARD
280  !      500   <CMD$> WAS NON-NULL BUT THE CARD WAS NOT ACTIVE CONTROLLER
290  !      501   <DATA$> WAS NON-NULL BUT THE CARD WAS NOT ACTIVER TALKER
300  !      502   <VAR$> WAS SPECIFIED BUT THE CARD WAS NOT ACTIVE LISTENER
310  !
320  INTEGER Select_code
330  DIM Cmd$[160],Data$[160],Var$[160]
340  IASSEMBLE
```

```
350  INPUT "HPIB SELECT CODE?",Select_code
360  PRINT "KEY 0 - OUTPUT     KEY 1 - ENTER     KEY 6 - EXIT"
370  DISP "IDLE"
380  GOTO 370
390 Output:  GOSUB Linput_cmd
400  LINPUT "DATA TO SEND?",Data$
410  ICALL Hpib_output(Select_code,Cmd$,Data$)
420  PRINT "      DATA SENT  =";Data$
430  RETURN
440 Enter:  GOSUB Linput_cmd
450  ICALL Hpib_enter(Select_code,Cmd$,Var$)
460  PRINT "      DATA READ  =";Var$
470  RETURN
480  !
490 Linput_cmd:  LINPUT "COMMAND BYTES?",Cmd$
500  RETURN
510  ! ast: SUBEXIT
520       ISOURCE              NAM Hpib
530       ISOURCE              EXT Get_value,Put_value,Error_exit
540       ISOURCE Cmd:         BSS 81                 ! STRING TO HOLD CMD BYTES
550       ISOURCE Data:        EQU Cmd                ! STRING TO HOLD DATA BYTES
560       ISOURCE Select_code:BSS 1                   ! INTERFACE SELECT CODE
570       ISOURCE Parm_ptr:    BSS 1                  ! POINTER TO PARM PSEUDO OPS
580       ISOURCE Lf:          EQU 10                 ! EQUATES
590       ISOURCE Cr:          EQU 13
600       ISOURCE Status1:     BSS 1                  ! 4 WORDS TO CONTAIN STATUS
610       ISOURCE Status2:     BSS 1                  ! BYTES FROM 98034
620       ISOURCE Status3:     BSS 1
630       ISOURCE Status4:     BSS 1
640       ISOURCE !
650       ISOURCE Out_parm:    SUB
660       ISOURCE              INT
670       ISOURCE              STR
680       ISOURCE P_data:      STR
690       ISOURCE Hpib_output:LDB =Out_parm           ! CALL SETUP ROUTINE
700       ISOURCE              JSM Hpib_setup
710       ISOURCE              LDA Out_parm           ! IS THERE A DATA PARAMETER?
720       ISOURCE              CPA =2
730       ISOURCE No_output:   RET 1                  ! NO, RETURN TO BASIC
740       ISOURCE              LDA =Data              ! YES, FETCH IT
750       ISOURCE              LDB =P_data
760       ISOURCE              JSM Get_value
770       ISOURCE              LDA Data               ! CHECK BYTE COUNT
780       ISOURCE              SZA No_output          ! IF ZERO, DO NOTHING
790       ISOURCE              JSM Hpib_status        ! MAKE SURE WE ARE ADDRESSED
800       ISOURCE              LDA Status4            ! TO TALK
810       ISOURCE              AND =40B
820       ISOURCE              RZA *+3
830       ISOURCE              LDA =501               ! ELSE GIVE ERROR 501
840       ISOURCE              JSM Error_exit
850       ISOURCE              LDA =Data+1            ! ELSE COMPUTE BYTE POINTER
860       ISOURCE              SAL 1                  ! SO WE CAN WITHDRAW BYTES
870       ISOURCE              STA C                  ! FROM THE STRING
880       ISOURCE              CBL
890       ISOURCE Data_loop:   SFC *                  ! WAIT FOR CARD
900       ISOURCE              WBC R4,I               ! OUTPUT A BYTE
910       ISOURCE              DSZ Data               ! SEE IF DONE WITH STRING
920       ISOURCE              JMP Data_loop          ! NO
930       ISOURCE              RET 1                  ! DONE, SO GO BACK TO BASIC
940       ISOURCE !
950       ISOURCE Ent_parm:    SUB
```

```
960     ISOURCE              INT
970     ISOURCE              STR
980     ISOURCE Ent_var:     STR
990     ISOURCE Hpib_enter: LDB =Ent_parm        ! CALL SETUP ROUTINE
1000    ISOURCE              JSM Hpib_setup
1010    ISOURCE              LDA =Ent_parm       ! IS THERE A DATA PARAMETER?
1020    ISOURCE              CPA =2
1030    ISOURCE              RET 1               ! NO, THEN I'D DONE
1040    ISOURCE              JSM Hpib_status     ! MAKE SURE I'M A LISTENER
1050    ISOURCE              LDA Status4
1060    ISOURCE              AND =20B
1070    ISOURCE              RZA *+3
1080    ISOURCE              LDA =502            ! ELSE GIVE ERROR 502
1090    ISOURCE              JSM Error_exit
1100    ISOURCE              LDA =0              ! CLEAR DATA STRING COUNTER
1110    ISOURCE              STA Data
1120    ISOURCE              LDA =Data           ! SET UP BYTE POINTER FOR DATA
1130    ISOURCE              SAL 1
1140    ISOURCE              ADA =1
1150    ISOURCE              STA C
1160    ISOURCE              CBL
1170    ISOURCE Enter_loop: SFC *               ! WAIT FOR CARD
1180    ISOURCE              LDA R4              ! START ACCEPTOR HANDSHAKE
1190    ISOURCE              SFC *               ! WAIT FOR DATA
1200    ISOURCE              LDA R6              ! READ DATA FROM CARD
1210    ISOURCE              CPA =Cr             ! IS IT A RETURN?
1220    ISOURCE              JMP Enter_loop      ! IF SO, IGNORE IT
1230    ISOURCE              CPA =Lf             ! IS IT TERMINATOR?
1240    ISOURCE              JMP Ent_done        ! YES, SKIP
1250    ISOURCE              PBC A,I             ! ELSE PUT BYTE INTO STRING
1260    ISOURCE              ISZ Data            ! BUMP STRING LENGTH
1270    ISOURCE              JMP Enter_loop      ! REPEAT FOR NEXT BYTE
1280    ISOURCE Ent_done:    LDA =Data           ! RETURN DATA TO PARAMETER
1290    ISOURCE              LDB =Ent_var
1300    ISOURCE              JSM Put_value
1310    ISOURCE              RET 1
1320    ISOURCE !
1330    ISOURCE ! HPIB SETUP ROUTINE
1340    ISOURCE !     B POINTS TO SUB PSEUDO OP (CONTAINS PARM COUNT)
1350    ISOURCE !    1) VERIFY PARAMETER COUNT >=2
1360    ISOURCE !    2) FETCH SELECT CODE AND VERIFY CARD IS A 98034A
1370    ISOURCE !    3) FETCH COMMAND STRING PARAMETER AND OUTPUT IT
1380    ISOURCE !
1390    ISOURCE Hpib_setup: LDA B,I             ! CHECK PARM COUNT
1400    ISOURCE              ADA =-2
1410    ISOURCE              SAP *+3             ! SKIP IF >=2
1420    ISOURCE              LDA =8              ! IF <2, GIVE ERROR 8
1430    ISOURCE              JSM Error_exit
1440    ISOURCE              ADB =1              ! POINT TO SELECT CODE PARM
1450    ISOURCE              STB Parm_ptr
1460    ISOURCE              LDA =Select_code    ! FETCH IT
1470    ISOURCE              JSM Get_value
1480    ISOURCE              LDA Select_code     ! CHECK RANGE FOR 1 TO 14
1490    ISOURCE              ADA =-1
1500    ISOURCE              SAM Sc_error
1510    ISOURCE              ADA =-15+1
1520    ISOURCE              SAM *+3
1530    ISOURCE Sc_error:    LDA =19             ! IF OUT OF RANGE, GIVE ERROR
1540    ISOURCE              JSM Error_exit      ! 19
1550    ISOURCE              LDA Select_code     ! SET UP PA AND DO STATUS SEQ
1560    ISOURCE              STA Pa              ! ON CARD TO VERIFY IT IS A
1570    ISOURCE              JSM Hpib_status     ! 98034A INTERFACE
```

```
1580      ISOURCE                 LDB Parm_ptr         ! NOW FETCH COMMAND STRING
1590      ISOURCE                 ADB =3
1600      ISOURCE                 LDA =Cmd
1610      ISOURCE                 JSM Get_value
1620      ISOURCE                 LDA Cmd              ! SEE IF THERE IS ANYTHING
1630      ISOURCE                 SZA No_cmd           ! OUTPUT, IF NOT, SKIP
1640      ISOURCE                 LDA Status4          ! MAKE SURE I AM ACTIVE
1650      ISOURCE                 AND =100B            ! CONTROLLER
1660      ISOURCE                 RZA *+3              ! SKIP IF YES
1670      ISOURCE                 LDA =500             ! ELSE GIVE ERROR 500
1680      ISOURCE                 JSM Error_exit
1690      ISOURCE                 LDA =Cmd+1           ! NOW OUTPUT THE COMMANDS
1700      ISOURCE                 SAL 1
1710      ISOURCE                 STA C
1720      ISOURCE                 CBL
1730      ISOURCE Cmd_loop:       SFC *
1740      ISOURCE                 WBC R6,I             ! SEND OUT CMD BYTE
1750      ISOURCE                 DSZ Cmd              ! SEE IF DONE
1760      ISOURCE                 JMP Cmd_loop         ! NOT YET
1770      ISOURCE No_cmd:         RET 1                ! DONE!
1780      ISOURCE !
1790      ISOURCE ! STATUS SEQUENCE FOR 98034 CARD. NOTE THAT THIS SEQUENCE
1800      ISOURCE ! COULD FORCE THE CARD TO VIOLATE THE IFC TIME SPECS IF
1810      ISOURCE ! THE FOLLOWING CONDITIONS EXIST:
1820      ISOURCE !     1) CARD IS NOT SYSTEM CONTROLLER
1830      ISOURCE !     2) A HARDWARE INTERRUPT OCCURS AFTER THE LDA R5 BUT
1840      ISOURCE !        BEFORE THE DIR
1850      ISOURCE !     3) THE CONTROLLER PULLS IFC AFTER THE LDA R5 BUT BEFORE
1860      ISOURCE !        THE DIR
1870      ISOURCE ! THE ONLY ALTERNATIVE TO THIS IS TO DIR BEFORE THE LDA R5.
1880      ISOURCE ! THIS HOWEVER COULD COMPROMISE ANY SYNCHRONUS INTERRUPT
1890      ISOURCE ! TRANSFER IN PROGRESS ( FOR EXAMPLE THE TAPE CARTRIDGE ).
1900      ISOURCE !
1910      ISOURCE Hpib_status:SFC *                    ! GET THE CARD INTO
1920      ISOURCE                 LDA R5               ! IT'S STATUS SEQUENCE.
1930      ISOURCE                 AND =60B             ! MAKE SURE IT IS A 98034
1940      ISOURCE                 CPA =60B
1950      ISOURCE                 JMP *+3              ! YES
1960      ISOURCE                 LDA =164             ! IF NOT, GIVE ERROR 164
1970      ISOURCE                 JSM Error_exit
1980      ISOURCE                 SFC *                ! (THIS IS THE CRITICAL TIME)
1990      ISOURCE                 DIR                  ! MADE IT, SO DISABLE MY
2000      ISOURCE                 SFC *                ! INTERRUPTS FOR THE REST OF
2010      ISOURCE                 LDA R6               ! THE STATUS SEQUENCE.
2020      ISOURCE                 STA Status1
2030      ISOURCE                 SFC *
2040      ISOURCE                 LDA R6
2050      ISOURCE                 STA Status2
2060      ISOURCE                 SFC *
2070      ISOURCE                 LDA R6
2080      ISOURCE                 STA Status3
2090      ISOURCE                 SFC *
2100      ISOURCE                 LDA R6
2110      ISOURCE                 EIR
2120      ISOURCE                 STA Status4
2130      ISOURCE                 RET 1
2140      ISOURCE                 END Hpib
```

Real Time Clock Example

```
10   ! PROGRAM TO DEMONSTRATE USING THE CLOCK FOR INTERRUPTS
20   !
30   ! THIS EXAMPLE SHOWS HOW TO USE THE CLOCK INTERRUPT TO PUT THE TIME
40   ! OF DAY INTO THE SYSTEM MESSAGE AREA AS LONG AS THE PROGRAM IS RUNNING.
50   !
60   ! THE CLOCK IS PROGRAMMED TO GENERATE AN INTERRUPT EVERY SECOND. THE
70   ! ASSEMBLY INTERRUPT SERVICE ROUTINE TRIGGERS AN END OF LINE BRANCH. THE
80   ! EOL BRANCH ROUTINE CALLS AN ASSEMBLY ROUTINE TO PUT THE TIME OF DAY
90   ! INTO THE SYSTEM MESSAGE AREA.
100  !
110  IDELETE ALL
120  ICOM 200
130  IASSEMBLE
140  ICALL Setup_clock                  ! SET UP ISR AND START CLOCK
150  ON INT #9 CALL Time                ! SET UP EOL BRANCH
160  !
170  ! BACKGROUND PROGRAM:
180  !
190  DISP I
200  I=I+1
210  GOTO 190
220  !
230  SUB Time
240  ICALL Display_time
250  SUBEXIT
260       ISOURCE              NAM Time
270       ISOURCE              EXT Error_exit,Printer_select,Print_string
280       ISOURCE              EXT Isr_access
290       ISOURCE Select_code:EQU 9
300       ISOURCE Eol_mask:    SET 1                ! GET ASSEMBLER TO COMPUTE
310       ISOURCE              REP Select_code      ! THE EOL MASK FOR TRIGGERING
320       ISOURCE Eol_mask:    SET Eol_mask*2       ! EOL BRANCHES
330       ISOURCE Cr:          EQU 13               ! OTHER EQUATES
340       ISOURCE Lf:          EQU 10
350       ISOURCE String:      BSS 20               ! AREA TO HOLD TIME OF DAY
360       ISOURCE Old_pi:      BSS 1                ! TWO WORDS TO HOLD CURRENT
370       ISOURCE Old_pw:      BSS 1                ! PRINTER IS AND PRINTER WIDTH
380       ISOURCE !
390       ISOURCE              SUB
400       ISOURCE Setup_clock:LDA =Select_code      ! MAKE SURE THE CLOCK CARD
410       ISOURCE              STA Pa               ! IS ALIVE
420       ISOURCE              SSS Card_ok
430       ISOURCE Card_down:    LDA =164            ! IF NOT, GIVE ERROR 164
440       ISOURCE               JSM Error_exit
450       ISOURCE Card_ok:     LDA =Isr             ! SET UP ISR LINKAGE
460       ISOURCE              LDB =(10*256)+(1*16)+Select_code
470       ISOURCE              JSM Isr_access
480       ISOURCE               JMP *+2             ! IF ERROR, THEN JUMP
490       ISOURCE               JMP Start_card      ! ELSE GO START UP THE CARD
500       ISOURCE              CPA =-1              ! IF DIDN'T GET RESOURCES
510       ISOURCE               JMP Setup_clock     ! THEN TRY AGAIN
520       ISOURCE              RET 1                ! IF ISR ALREADY LINKED,RETURN
530       ISOURCE Start_card: LDA =="U4H/U4=04/U4P1000/U4G"+Lf
540       ISOURCE              SAL 1                ! SET UP C TO POINT TO STRING
550       ISOURCE              STA C                ! WHICH I WILL OUTPUT TO THE
560       ISOURCE              CBL                  ! CLOCK TO PROGRAM IT.
570       ISOURCE              LDB =-21             ! B IS -(CHAR COUNT-1)
580       ISOURCE Out_loop:    SFC *                ! WAIT FOR CARD
```

```
590   ISOURCE               WBC R4,I           ! SHOVE NEXT BYTE OUT TO CARD
600   ISOURCE               STA R7             ! TRIGGER HANDSHAKE
610   ISOURCE               RIB Out_loop       ! LOOP UNTIL DONE
620   ISOURCE               LDA =200B          ! ENABLE THE CARD TO INTERRUPT
630   ISOURCE               STA R5
640   ISOURCE               RET 1
650   ISOURCE !
660   ISOURCE               SUB
670   ISOURCE Display_time:LDA =Select_code    ! FETCH TIME FROM CLOCK
680   ISOURCE               STA Pa
690   ISOURCE               SSC Card_down      ! OOPS...CARD WENT DOWN
700   ISOURCE               LDA ='R            ! OUTPUT "R" TO CLOCK TO GET
710   ISOURCE               SFC *              ! IT TO GIVE ME THE TIME
720   ISOURCE               STA R4
730   ISOURCE               STA R7
740   ISOURCE               LDA =Lf
750   ISOURCE               SFC *
760   ISOURCE               STA R4
770   ISOURCE               STA R7
780   ISOURCE               LDA =String        ! SET UP C TO PUT TIME OF DAY
790   ISOURCE               SAL 1              ! DATA INTO STRING
800   ISOURCE               ADA =1
810   ISOURCE               STA C
820   ISOURCE               CBL
830   ISOURCE               LDA =0             ! CLEAR THE STRING COUNT
840   ISOURCE               STA String
850   ISOURCE               SFC *              ! WAIT FOR CARD
860   ISOURCE               LDA R4             ! START INPUT OPERATION
870   ISOURCE Read_loop:    STA R7             ! TRIGGER HANDSHAKE
880   ISOURCE               SFC *              ! WAIT FOR CARD
890   ISOURCE               LDA R4             ! GET THE NEXT BYTE
900   ISOURCE               CPA =Cr            ! IGNORE CR'S
910   ISOURCE                JMP Read_loop
920   ISOURCE               CPA =Lf            ! TERMINATE ON LINEFEED
930   ISOURCE                JMP Got_time
940   ISOURCE               PBC A,I            ! ELSE PUT CHARACTER INTO
950   ISOURCE               ISZ String         ! STRING AND BUMP COUNT
960   ISOURCE               JMP Read_loop      ! REPEAT
970   ISOURCE Got_time:     LDA =18            ! SET UP "PRINTER IS" FOR THE
980   ISOURCE               LDB =80            ! MESSAGE AREA
990   ISOURCE               JSM Printer_select
1000  ISOURCE               STA Old_pi         ! SAVE OLD
1010  ISOURCE               STB Old_pw
1020  ISOURCE               LDA =String        ! DO THE PRINT
1030  ISOURCE               JSM Print_string
1040  ISOURCE                JMP Memov         ! JUMP IF MEMORY OVERFLOW
1050  ISOURCE                NOP               ! IGNORE STOP KEY
1060  ISOURCE Restore_pi: LDA Old_pi           ! RESET "PRINTER IS"
1070  ISOURCE               LDB Old_pw
1080  ISOURCE               JSM Printer_select
1090  ISOURCE               RET 1              ! RETURN TO BASIC
1100  ISOURCE Memov:        JSM Restore_pi     ! RESTORE THE PRINTER IS
1110  ISOURCE               LDA =2             ! AND GIVE ERROR 2
1120  ISOURCE               JSM Error_exit
1130  ISOURCE !
1140  ISOURCE Isr:          LDA =0             ! SIGNAL CARD THAT WE GOT THE
1150  ISOURCE               STA R5             ! INTERRUPT BY DISABLING AND
1160  ISOURCE               LDA =200B          ! THEN RE-ENABLING THE CARD
```

```
1170    ISOURCE          STA R5
1180    ISOURCE          LDB Isr_psw         ! TRIGGER EOL BRANCH
1190    ISOURCE          LDA =103B
1200    ISOURCE          STA B,I
1210    ISOURCE          ADB =3
1220    ISOURCE          LDA =Eol_mask
1230    ISOURCE          DIR
1240    ISOURCE          IOR B,I
1250    ISOURCE          STA B,I
1260    ISOURCE          EIR
1270    ISOURCE          RET 1               ! DONE
1280    ISOURCE          END Time
```

Appendix I
Demonstration Cartridge

Along with the Assembly Language Development and Execution ROMs, a tape cartridge has been provided to demonstrate the capabilities of the assembly language system. This Demonstration Cartridge (HP part number 11141-10155) is specifically intended to —

- Graphically display the kind of speed increases which can be obtained by using assembly lánguage subprograms for certain types of applications.
- Provide a number of programs which can serve as examples of how to write assembly language subprograms.[1]
- Provide a set of definitions for some of the special function keys so that those keys can be used as typing aids.

Using the Tape

To run any of the demonstration programs, execute the statement —

```
LOAD "DEMO",1
```

A set of instructions is displayed which can then be followed interactively.

Typing Aids

The starting and final cursor positions of the typing aids were chosen with assembly listings in mind. The intent in selecting these positions was to make it easy to enter source as it would appear when listed within an assembly listing.

The following table gives, for each key, the typing aid, the position where the typing aid begins, and the position where the cursor will finally reside. Because some typing aids end with a blank, the triangle (Δ) has been chosen to indicate the end of the typing aid. All blanks after the start of the typing aid, and before the triangle, will appear when the key is pressed.

1 The commented source for the chess program is contained in file CHESS.

Key	Typing Aid	Typing Aid Starting Position	Final Cursor Position
0	`ISOURCE Δ`	11	31
1	`ISOURCE Δ`	11	19
2	`ISOURCE ! Δ`	11	21
3	[CLEAR LINE] `PRINT "A=: ",IMEM(A), "B=: ",IMEM(B)` [EXEC]	home	
4	[CLEAR LINE] `LINK ""Δ`	home	7 (over second quote mark in insert character mode)
5	[CLEAR LINE] `CONT`	home	6
6	[CLEAR LINE] `REWIND ":T"Δ`	home	11 (over second quote mark in insert character mode
7	`ISOURCE ! Δ`	11	53
8	[CLEAR LINE] `GET ""Δ`	home	6 (over second quote mark in insert character mode)
9	[CLEAR LINE] `LOAD ""Δ`	home	7 (over second quote mark in insert character mode)
10	[CLEAR LINE] `SAVE ""Δ`	home	7 (over second quote mark in insert character mode)
11	[CLEAR LINE] `STORE ""Δ`	home	8 (over second quote mark in insert character mode)
12	[CLEAR LINE] `EDIT Δ`	home	6
13	[CLEAR LINE] `CAT ""Δ`	home	6 (over second quote mark in insert character mode)
14	[CLEAR LINE] `LIST Δ`	home	6
15	[CLEAR LINE] `SCRATCH Δ`	home	9
16	[CLEAR LINE] `PRINTER IS Δ`	home	12
17	[CLEAR LINE] `PRINTALL IS Δ`	home	13
18	[CLEAR LINE] `IBREAK Δ`	home	8
19	[CLEAR LINE] `IPAUSE Δ`	home	8
20	[CLEAR LINE] `IASSEMBLE Δ`	home	11
21	(used by other keys)		
22	(used by other keys)		
23	`!Δ`	51	53
24	[CLEAR LINE] `MASS STORAGE IS ""Δ`	home	18 (over second quote mark in insert character mode)
25	`BIN` (use only after using keys 9 or 11)	current − 1	current + 4 (over second quote mark in insert character mode)

Key	Typing Aid	Typing Aid Starting Position	Final Cursor Position
26	RE- (EXEC) (use before keys 10 or 11)	home	
27	KEY (use only after using Keys 9 or 11)	current − 1	current + 4 (over second quote mark in insert character mode)
28	:TΔ	current	current + 2
29	:FΔ	current	current + 2
30	(CLEAR LINE) PURGE ""Δ	home	8 (over second quote mark in insert character mode)
31	(CLEAR LINE) CREATE "" Δ	home	9 (over second quote mark in insert character mode)

Appendix J
Error Messages

Mainframe Errors

1 Missing ROM or configuration error. Also, check to see if all option ROMs are installed properly.

2 Memory overflow; subprogram larger than block of memory. Also check to see if your arrays are too large to fit in memory.

3 Line not found or not in current program segment. Check the spelling of line labels and line identifiers.

4 Improper return. Branched into the middle of a subroutine.

5 Abnormal program termination; no END or STOP statement.

6 Improper FOR/NEXT matching.

7 Undefined function or subroutine. Check spellings.

8 Improper parameter matching. Check the parameter lists in SUB and CALL, and DEF FN and FN statements to see if they match in number and type.

9 Improper number of parameters. Check the number of arguments used in an FN or CALL reference.

10 String value required.

11 Numeric value required.

12 Attempt to redeclare variable. Once a variable name has been declared in a DIM, COM, REAL, SHORT or INTEGER statement, it can't be redeclared in that program segment.

13 Array dimensions not specified. You must dimension the array, either explicitly or implicitly.

14 Multiple OPTION BASE statements or OPTION BASE statement preceded by variable declarative statements.

15 Invalid bounds on array dimension or string length in DIM, COM, REAL, SHORT or INTEGER statement. Strings can't be longer than 32 767 characters. The range of array subscripts is −32 767 through 32 767.

16	Dimensions are improper or inconsistent; more than 32 767 elements in an array. Check for wrong number of subscripts in an array reference. Check any matrix multiplication for proper sizes.
17	Subscript out of range.
18	Substring out of range or string too long. Check substring specifiers against length of string.
19	Improper value. Check numbers being entered, especially their exponents.
20	Integer precision overflow. The range is −32 768 through 32 767.
21	Short precision overflow. Short-precision numbers have six significant digits and an exponent in the range −63 through 63.
22	Real precision overflow. Full-precision numbers have twelve significant digits and an exponent in the range −99 through 99.
23	Intermediate result overflow.
24	TAN ($n * \pi/2$), when n is odd
25	Magnitude of argument of ASN or ACS is greater than 1.
26	Zero to negative power.
27	Negative base to non-integer power.
28	LOG or LGT of negative number.
29	LOG or LGT of zero.
30	SQR of negative number.
31	Division by zero; or X MOD Y with Y = 0.
32	String does not represent valid number or string response when numeric data required. Check any use of VAL function and its argument. Check for correct spelling of variable name.
33	Improper argument for NUM, CHR$, or RPT$ function.
34	Referenced line is not IMAGE statement. Check the line identifier in the PRINT USING statement.
35	Improper format string.
36	Out of DATA. Make sure READ and DATA statements correspond. Use RESTORE if appropriate.

37 EDIT string longer than 160 characters. Try using a substring.

38 I/O function not allowed. TYP and other I/O functions aren't allowed in any I/O statement like DISP or PRINT. Place the value into a variable.

39 Function subprogram not allowed. An FN reference isn't allowed in any I/O statement, or in redim subscripts. Place the value into a variable.

40 Improper replace, delete or REN command. SUB and DEF FN can only be replaced by another SUB or DEF FN. They can only be deleted if the rest of the corresponding subprogram is deleted. A renumbering may cause out-of-range line numbers if completed, so an error occurs; check increment value.

41 First line number greater than second.

42 Attempt to replace or delete a busy line or subprogram. Typically, this is caused by trying to delete an input statement that is still requesting values.

43 Matrix not square. The dimensions of an identity matrix or of one used to find an inverse or determinant must be the same size.

44 Illegal operand in matrix transpose or matrix multiply. The result matrix can't be one of the operands.

45 Nested keyboard entry statements.

46 No binary in memory for STORE BIN or no program in memory for SAVE. Check line numbers in SAVE against program in memory.

47 Subprogram COM declaration is not consistent with main program. Check number, type and dimensions of variables.

48 Recursion in single-line DEF FN function. Only subprograms can be called recursively.

49 Line specified in ON declaration not found.

50 File number less than 1 or greater than 10.

51 File not currently assigned. Execute an ASSIGN statement for the file, or check the accuracy of the file number used.

52 Improper mass storage unit specifier. Check the values of the select code, unit code and controller address.

53 Improper file name. A file name can have 1-6 characters and can't contain a

53	Improper file name. A file name can have 1-6 characters and can't contain a colon, quote mark, NULL or CHR$(255).
54	Duplicate file name. Choose another name or PURGE the old one.
55	Directory overflow. There is a maximum number of files that a mass storage medium can hold. A file will have to be removed to add another.
56	File name is undefined. Check the spelling.
57	Mass Storage ROM is missing. Check to see that the ROM is installed properly.
58	Improper file type. Use LOAD for PROG files, ASSIGN and GET on DATA files and LOADKEY for KEYS files.
59	Physical or logical end-of-file found. Attempting to READ# or PRINT# past the end of the file. Compare the data list to the file size.
60	Physical or logical end-of-record found in random mode. Compare the data list to the record size.
61	Defined record size is too small for data item. You can either PURGE and RE-CREATE the file with longer records or regroup the data being recorded.
62	File is protected or wrong protect code specified. Check to see that the protect code is included and spelled properly.
63	The number of physical records is greater than 32 767. That's the limit; use something smaller.
64	Medium overflow (out of user storage space). A file can't be set up because there isn't enough space. Use another medium or purge unwanted files.
65	Incorrect data type. You can't use GET on a DATA file that doesn't contain a program. Use TYP to find out what kind of data the computer is trying to be read.
66	Excessive rejected tracks during a mass storage initialization. The medium can't be initialized. If the medium is a flexible disk, use a different one. If the medium is a hard disc, call your HP Sales and Service Office for assistance, to determine whether there has been a hardware failure.
67	Mass storage parameter less than or equal to 0. Check values of variables. Record numbers, record lengths and number of defined records must be positive numbers.

68	Invalid line number in GET or LINK operation. Check line numbers. May be trying to LINK to file that doesn't contain a program.
69	Format switch on the disc off. Turn it on.
70	Not a disc interface. Check mass storage unit specifier.
71	Disc interface power off. Turn it on.
72	Incorrect controller address, controller power off, or disc time out. Check mass storage unit specifier; make sure controller is on.
73	Incorrect device type in mass storage unit specifer.
74	Drive missing or power off.
75	Disc system error, type I[1].
76	Incorrect unit code in mass storage unit specifier.
77	Disc system error, type II[1].
78-79	Reserved for future use.
80	Cartridge out or door open. Also check to see if interface is connected properly.
81	Mass storage device failure. Possible power failure.
82	Mass storage device not present. Check mass storage unit specifier.
83	Write protected. Check the write-protection device on the medium or drive.
84	Record not found. There is a bad spot on the medium.
85	Mass storage medium is not initialized.
86	Not a compatible tape cartridge.
87	Record address error; information can't be read. Hardware failure. Check for a dirty read head.
88	Read data error. Hardware failure. Check for a dirty read head.
89	Check read error.
90	Mass storage system error.
91-99	Reserved for future use.
100	Item in print using list is string but image specifier is numeric.

1 See the Mass Storage Techniques Manual.

101	Item in print using list is numeric but image specifier is string.
102	Numeric field specifier wider than printer width.
103	Item in print using list has no corresponding image specifier.
104	ON KBD or TOPEN not allowed in subprogram.
105–109	Reserved for future use.
110	Plotter type specification not recognized. Check spelling of "GRAPHICS", "9872A" or "INCREMENTAL".
111	Plotter has not been specified. Check select codes.
112	No graphics hardware installed in the System 45B.
113	LIMIT specifications out of range.
114	98036 card improperly configured.
115	TDISP not allowed unless peripheral keyboard active.
116	TOPEN is active on another select code.
117–149	Reserved for future use.
150	Improper select code.
151	A negative select code was specified that does not match present bus addressing.
152	Parity error.
153	Either insufficient input data to satisfy enter list, attempt to ENTER from source into source or enter count exhausted without linefeed.
154	Integer overflow, or ENTER count greater than 32 767 bytes or 16 383 words.
155	Invalid interface register number. (Can only specify 4-7.)
156	Improper expression type in READIO, WRITEIO, or STATUS list.
157	No linefeed was found to satify % ENTER image specifier, or no linefeed record delimiter was found in 512 characters of input.
158	Improper image specifier or nesting image specifiers more than 4 levels deep.
159	Numeric data was not received for numeric enter list item.
160	Repetition of input character more than 32 768 times.

161	Attempted to create CONVERT table or EOL sequence for source or destination variable which is locally defined in a subprogram.
162	Attempted to delete a nonexistent CONVERT table or EOL sequence.
163	I/O error, such as interface card not present, device timeout, interface or peripheral failure (Interface FLAG line=0.), stop key pressed or improper interface card type.
164	Transfer type specified is incorrect type for interface card.
165	A FHS or DMA transfer with no format specifies a count that exceeds th size of the variable, or an image specifier indicates more characters than will fit in the specified variable.
166	A NOFORMAT FHS or DMA type transfer does not start on an odd numbered character position, such as A$[3].
167	Interface status error, TRL Character or an EOI was received on an HP-IB Interface before ENTER list or image specification was satisfied.
168-183	Reserved for future use.
184	Improper argument for OCTAL or DECIMAL function or assembled location.
185	Break Table overflow.
186	Undefined BASIC label or subprogram name used in IBREAK statement.
187	Attempt to write into protected memory; or, attempt to execute instruction not in ICOM region.
188	Label used in an assembled location not found.
189	Doubly-defined entry point or routine.
190	Missing ICOM statement.
191	Module not found.
192	Errors in assembly.
193	Attempt to move or delete module containing an active interrupt service routine.
194	IDUMP specification too large. Resulting dump would be more than 32 768 elements.
195	Routine not found.

196	Unsatisfied externals.
197	Missing COM statement.
198	BASIC's common area does not correspond to assembly module requirements.
199	Insufficient number of BASIC COM items.
200–206	Reserved for future use.
207	Binaries not allowed in LOAD SUB file. Do LOAD, SAVE, SCRATCH A, GET and STORE on the file to get rid of binaries. However, the loaded program may not run after the binaries are removed.
208	Volume not mounted. Mount it and execute a VOLUME DEVICES ARE statement.
209	Operation not allowed on tape. Only the BKUP file used in DBBACKUP and DBRECOVER is allowed on tape.
210	Bad status array. It must be defined as integer precision with $\geq$ 10 elements. Check spelling and current size.
211	Improper data base specified or data base not open. Improper name, or performing data base operation with invalid name.
212	Data set not found. Check set name or number and make sure it is on the volume specified in the schema.
213	Reserved for future use.
214	Data base requires creation. Perform a DBCREATE.
215–217	Reserved for future use.
218	Volume name not part of data base. Check spelling.
219	Out of available memory for a DBOPEN, DBBACKUP or DBRECOVER. Out of read/write memory if executed from main program. Out of special area if executed from subprogram, so perform the DBOPEN in the main program.
220	Improper or illegal use of maintenance word. Check spelling and leading or trailing blanks.
221	Data set not created.
222	Reserved for future use.

223	Improper backup file. In DBRECOVER, backup file has incorrect information in header or no primary DBBACKUP/RECOVER currently in progress (for secondary operation).
224	Incomplete backup file. More than one volume in backup; probably mounted in the wrong order. Start the recovery over.
225	Improper utility version number in root file. Rerun Schema Processor to generate new root file.
226	Corrupt data base – must purge and redefine. Purge root file and run Schema Processor.
227	Corrupt data base – all sets require erasure.
228	Data sets cannot be re-created without root file.
229	Operation not allowed while DBOPEN current. Perform a DBCLOSE mode 1.
230	Improper set list in DBBACKUP, DBCREATE, DBERASE, DBPURGE or duplicate sets in the set list.
231–232	Reserved for future use.
233	Required data set root file not mounted. Mount it and perform a VOLUME DEVICES ARE.
234	Referenced line not a PACKFMT statement. Make sure line identifier is correct and that it references a PACKFMT statement.
235	Reserved for future use.
236	Insufficient length in a PACK statement, or insufficient current length in an UNPACK. Insufficient length in a DBBACKUP or DBRECOVER statement.
237	List length > 32 767 in PACK or UNPACK. Array in PACKFMT too large. Make sure it is the correct variable; redimension if necessary.
238	Numeric conversion error. Improper real number found. Check PACKFMT to make sure a REAL or SHORT variable, not INTEGER is being unpacked.
239	UNPACK requires a source string of greater length.
240–329	Reserved for future use.
300	CCOM area not allocated
301	Not allowed when channel is active

302	CMODEL statement required
303	Not allowed when trace is active
304	Too many characters in CWRITE
305	New CCOM size not allowed when channel is active
306	98046 card failure
307	Insufficient CCOM allocation
308	Illegal character in CWRITE of non-TRANSPARENT data
309	Not allowed for this CMODEL
310	CCONNECT statement required
311	Not allowed while Data Comm is suspended
312	Improper CSTATUS array
313–329	Reserved for future use.
330	Lexical table size exceeds array size.
331	Improper pointer array*.
332	Non-existent dimension specified in MAT REORDER.
333	Pointer array contains out-of-range subscript value.
334	Pointer array length does not equal number of records.
335	Pointer array is not one-dimensioned.
336	Number of records (plus twice the number of secondary keys plus twice the number of substrings) exceeds 16 383.
337	Subscript extends beyond dimensioned maximum length.
338	Subscript out-of-range in key specifier.
339	Starting location is an out-of-range subscript value.
340	Lexical table is too small to include all characters.
341	Main lexical table length plus mode section length does not equal specified table length.

* This error occurs when data is lost in the process of reordering the array. If this error does not occur, it does not necessarily imply that the pointer array contains a permutation.

342 Array is not one-dimensioned or is not integer.

343 Lexical mode section pointer out-of-range.

344 Lexical table length exceeds 16 383.

900–999 Reserved for user.

System Error octal number; octal number

This error indicates a malfunction in the machine's firmware system. Contact your Sales and Service Office.

I/O Device Errors

Two error messages can occur when attempting to direct an operation to an I/O device that is not ready for use. A printer which is out of paper or no device at a specifed select code are examples. The first message that appears is –

I/O ERROR ON SELECT CODE select code

If the condition is not corrected, the machine beeps intermittently and the following message replaces the first –

I/O TIMEOUT ON SELECT CODE select code

The I/O device can be made usable by correcting the error (loading paper, or changing the select code, for example), then executing the READY# command –

READY# select code

This command readies the I/O device and the operation which was attempted is attempted again. The select code must be specified by an integer.

If you get an I/O error on select code 0 and the printer is not out of paper, call your Sales and Service Office.

In some cases, such as an interface which is not connected, READY# for that select code may not solve the I/O error. In this case, STOP should be pressed to regain control of the computer. Be sure to turn the power off before inserting an interface. After the problem is remedied, the operation or program can be tried again.

If you get an I/O error and you have an ON KBD statement in effect, you must press STOP to gain control of the computer. Otherwise, the READY# command will be trapped by ON KBD.

CSTATUS Element 0 Errors

10	Timeout before connection
11	Clear to Send line false or missing clock
100	Channel MEMLIMIT overflow
101	Illegal protocol from remote
102	Input buffer overflow
103	Internal buffer overflow
104	Autodisconnect forced
105	RETRIES count exceeded
106	NOACTIVITY timeout
200	98046 buffer overflow

Assembly-Time Errors

DD	Doubly-defined label
EN	END instruction missing; or module name does not match.
EX	Expression evaluation error.
LT	Literal pools full or out of range.
MO	ICOM region overflow.
RN	Operand out of range.
SQ	Argument declaration pseudo-instruction out of sequence.
TP	Incorrect type of operand used.
UN	Undefined symbol.

IMAGE Status Errors

The following are possible values and meanings of the condition word (first element of the status array). After an error, the status array is as follows –

Element	Description
1	Condition word is non-zero
2-4	No change
5	DBOPEN mode
6	Statement identification number
7	Program line number
8	0
9	Value of the mode parameter
10	Integer-for system use only

Each statement has an identification number.

Number	Statement
401	DBOPEN
402	DBINFO
403	DBCLOSE
404	DBFIND
405	DBGET
406	DBUPDATE
407	DBPUT
408	DBDELETE

Condition Word Value	Error Description
0	Successful execution – no error
-1	Improper data base name; already have read/write access to the data base
-10	You may not open additional data bases; five are already opened
-11	Bad data base name or preceding blanks missing. Don't change the first two characters. Data base may not be open.
-14	DBPUT, DBDELETE and DBUPDATE not allowed in DBOPEN mode 8
-21	Bad password – grants access to nothing or not to that set. Check spelling. Data item, data set, or volume nonexistent or inaccessible. Check spelling and DBOPEN password. Volume references must be numeric for DBINFO.

-22	Detail data set required
-23	You lack write access to this data set
-24	DBPUT or DBUPDATE not allowed on Automatic Master. Check correctness of set reference.
-31	Improper mode in data base statement. DBGET mode 5 bad – specified data set lacks chains
-52	Item specified is not an accessible key item in the specified set. Bad @ parameter – must be "@;" or "@ " or "@".
-74	Root file name in disc directory and name in root file are different. Make sure root file not moved or renamed.
-91	Root file version not compatible with current IMAGE/45 statements. Incorrect version of Schema Processor used.
-92	Data base requires creation
-94	Data or structure information lost. Data base must be erased or redefined.
-95	Cannot DBOPEN while a DBBACKUP or DBRECOVER is going on.
11	End of file on serial DBGET; no entries following the current record.
12	Negative record number on directed DBGET. Check record number and spelling.
13	Record number greater than capacity on directed DBGET. Check record number and spelling.
15	End of chain encountered
16	The data set is full
17	No current record or the current record is empty; make sure that a current record is defined for this set. There is no chain for the key item value. There is no entry with the specified key value
18	Broken chain. Must UNLOAD the data base.
41	DBUPDATE will not alter a key item. Make sure correct key item values are in the correct places in the buffer string.
43	Duplicate key item value in master not allowed.

44	Can't delete a Master entry with non-empty detail chains
50	Buffer string is too small for requested data. Redimension if necessary.
53	Argument parameter type incompatible with key field type (DBGET, mode 7 or DBFIND) or current length of string argument is less than the string length of the key item value.
80	Data set's volume is not on line; or set not created.
94	Corrupt data base successfully opened in mode 8
1xx	There is no chain head for path xx
3xx	The automatic master for path xx is full
4xx	The master data set for path xx is not on-line (Applies to DBPUT and DBDELETE for detail data sets)
500	Root file volume isn't mounted.
5xx	Needed volume on-line; created data set xx isn't there

Appendix K
Maintenance

Maintenance Agreements

Service is an important factor when you buy Hewlett-Packard equipment. If you are to get maximum use from your equipment, it must be in good working order. An HP Maintenance Agreement is the best way to keep your equipment in optimum running condition.

Consider these important advantages —

- Fixed Cost — The cost is the same regardless of the number of calls, so it is a figure that you can budget.
- Priority Service — Your Maintenance Agreement assures that you receive priority treatment, within an agreed-upon response time.
- On-Site Service — There is no need to package your equipment and return it to HP. Fast and efficient modular replacement at your location saves you both time and money.
- A Complete Package — A single charge covers labor, parts, and transportation.
- Regular Maintenance — Periodic visits are included, per factory recommendations, to keep your equipment in optimum operating condition.
- Individualized Agreements — Each Maintenance Agreement is tailored to support your equipment configuration and your requirements.

After considering these advantages, we are sure you will see that a Maintenance Agreement is an important and cost-effective investment.

For more information, please contact your local HP Sales and Service Office.

Appendix L
9835/9845 Compatibility

System 35 and System 45 assembly language programs are for the most part source code compatible. The exceptions to this are noted below. For example, a GET command can be used by a System 45 to retrieve source code which has been SAVE'd on a System 35, and vice versa. However, object code files (ILOAD, ISTORE) are not compatible.

The following items specify the differences between the two assembly language systems.

1. The following 9835/9845 differences affect source code compatibility –
   - The 9845 has 9 Base_page temporaries; the 9835 has 50.
   - The absolute addresses of the routines within the Rel_math utility are different, and must be changed between the 9845 and the 9835.
   - The 9845 has two fewer return stack entries than the 9835.
   - The Get_info utility returns additional information when used with the 9845. The number of words returned depends upon the memory size of the machine used –
     - 33 words for machines over 256K
     - 36 words for machines over 320K
     - 39 words for machines over 384K

     Additional space for this information may need to be reserved in assembly language programs which are moved to larger machines.
   - The Isr_flag link is not needed in the code that notifies BASIC of an interrupt, on the 9845. This link is used only in the 9835 code, and should be removed from any code run on the 9845.
   - The keyboards of the 9845 and 9835 differ. Keyboard and printer register operations differ also. (See the Assembly Language Quick Reference manual.)
2. The 9845 has two additional utilities, To_system and Print_no_lf.
3. The LINES option to the IASSEMBLE statement has been expanded on the 9845 to include a negative line number. If a negative number is used, no additional carriage-return, linefeed characters are sent after each module has been printed. Of course, if the EJECT option has been specified, a formfeed character is sent after each page.

4. The 9845 allows symbolic debugging (e.g., IDUMP Test) of all ENT and SUB symbols, regardless of whether they appear in assembled code or in ILOAD'ed code. The 9835 allows symbolic debugging only if the symbols appear in assembled code which is in its original, unmoved position in the ICOM region.
5. IOF and ION have been added as pseudo-instructions in the 9845 Assembly Language. They are used to control the automatic setting of indirect bits in generated code.
6. Rel_to_sho returns 0 or an error number in the A register, for the 9845.

Note that two processors are used in the 9845 and one is used in the 9835. (For non-ISR assembly language code, the two 9845 processors function together as a single unit to maintain compatibility with the 9835.) The advantages of two processors are –

- Overlapped I/O (in the OVERLAP mode) can in some cases bring about speed enhancements.
- An ISR (interrupt service routine) can be executed simultaneously with a BASIC program.

Subject Index

a

b

e

f

t

u

v

w

x

Your Comments, Please...

Your comments assist us in improving the usefulness of our publications; they are an important part of the inputs used in preparing updates to the publications.

In order to write this manual, we made certain assumptions about your computer background. By completing and returning the comments card on the following page you can assist us in adjusting our assumptions and improving our manuals.

Feel free to mark more than one reply to a question and to make any additional comments.

Please do not use this form for questions about technical applications of your system or requests for additional publications. Instead, direct those inquiries or requests to your nearest HP Sales and Service Office.

If the comments card is missing, please address your comments to:

HEWLETT-PACKARD COMPANY
Desktop Computer Division
3404 East Harmony Road
Fort Collins, Colorado 80525 U.S.A.

Attn. Customer Documentation
Dept. 4231

All comments and suggestions become the property of Hewlett-Packard.

Comments Card for the Assembly Development ROM Manual

Yourself

1. What is your major application of your Series 9800 Desktop Computer?

____

2. Which Series 9800 Desktop Computer do you have?

   ○ 9835A ○ 9835B ○ 9845B ○ 9845C

3. What was your level of programming knowledge before you started using this manual?

   ○ none ○ beginner ○ intermediate ○ expert

The Manual

1. Did you have any difficulty in:

	none	minimal	some	considerable
understanding material in manual?	○	○	○	○
applying that information?	○	○	○	○

2. How would you rate the:

	excellent	good	fair	poor
areas covered	○	○	○	○
depth of coverage	○	○	○	○
examples	○	○	○	○
indexing	○	○	○	○
organization	○	○	○	○
overall manual	○	○	○	○

What do you suggest we do to improve the areas that you consider weak?

The Method

1. By which method would you have preferred to learn the use of your computer?

   ○ Present set of manuals
   ○ Manuals using programmed-learning approach
   ○ Training workbooks with corresponding tape cartridge
   ○ Manuals resident in computer's memory, accessible through the keyboard and displayed on the CRT
   ○ Training in classroom situation at Hewlett-Packard

General comments:

Name: ____

Address: ____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Desktop Computer Division
Attn: Cust. Documentation/Dept. 4231
3404 East Harmony Road
Fort Collins, Colorado 80525

Assembly Language ROM Errors

184	Improper argument for OCTAL or DECIMAL function or assembled location.
185	Break Table overflow.
186	Undefined BASIC label or subprogram name used in IBREAK statement.
187	Attempt to write into protected memory; or, attempt to execute instruction not in ICOM region.
188	Label used in an assembled location not found.
189	Doubly-defined entry point or routine.
190	Missing ICOM statement.
191	Module not found.
192	Errors in assembly.
193	Attempt to move or delete module containing an active interrupt service routine.
194	IDUMP specification too large. Resulting dump would be more than 32 768 elements.
195	Routine not found.
196	Unsatisfied external symbols.
197	Missing COM statement.
198	BASIC's common area does not correspond to assembly module requirements.
199	Insufficient number of items in BASIC COM declarations.

Assembly-Time Errors

DD	Doubly-defined label
EN	END instruction missing; or module name does not match.
EX	Expression evaluation error.
LT	Literal pools full or out of range.
MO	ICOM region overflow.
RN	Operand out of range.
SQ	Argument declaration pseudo-instruction out of sequence.
TP	Incorrect type of operand used.
UN	Undefined symbol.

Assembly Development ROM

HEWLETT PACKARD

Part No. 09845-91083
Microfiche No. 09845-98083

Printed in U.S.A.
March 1, 1980