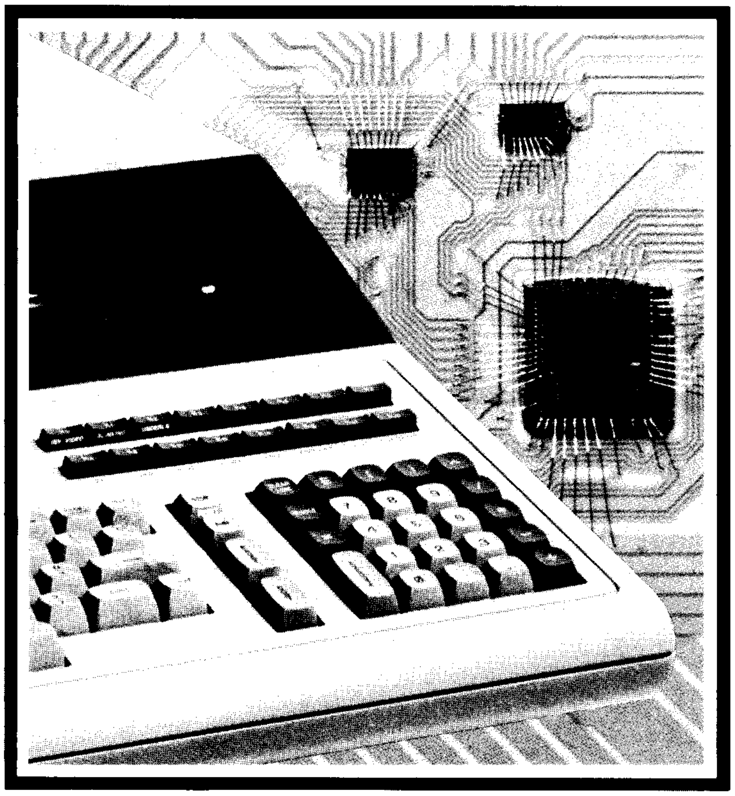
Assembly Execution ROM Manual

For the HP 9845



HEWLETT PACKARD

Assembly Execution ROM

Part No. 09845-91082
Microfiche No. 09845-98082

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

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

February 1980...First Edition

June 1981...Second Edition. Updated pages: ii, iii, 3, 12, 23; pages 27-35 were removed

Table of Contents

Chapter 1
General Information

The Assembly Execution Read Only Memory (ROM) has been provided to you so that you can load and execute assembly language programs which have been written using the Assembly Execution and Development ROM. When installed, the Execution ROM reserves 198 16-bit words of read/write memory if the I/O ROM is present and 266 words if the I/O ROM is not present. This read/write memory cannot be accessed for storage of programs or data.

It is assumed throughout this manual that you are familiar with the basic operation and language of the 9845[1]. It is not necessary, however, that you be in any way familiar with the Assembly Development ROM itself in order to use the Assembly Execution ROM or this manual. All of the capabilities provided by the Assembly Execution ROM are in the form of BASIC language extensions and are used as any other BASIC statement may be.

Equipment Supplied

The following items are supplied with the Assembly Execution ROM –

Item	Part Number
Assembly Execution ROM manual	09845-91082
Error Label	7120-8770

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

Purpose of the ROM

The Assembly Execution ROM (HP part number 98438A) is used to load, store, and execute assembly language routines written using the Assembly Development ROM.

The routines are provided by HP in some instances, or are created by others. Instructions for the effective use of the routines themselves are the responsibility of the people (the "authors") who developed those routines. Thus, when calling a routine (or even deciding which routine to call) consult the documentation provided by the authors of the routine.

Buzzwords

During the course of the discussion in this manual, phrases are used which are in common circulation in the computer industry. While the meaning of most are either well-known or deducible from the context, there are a few which may be new to the user not exposed to assemblers before —

bit — the most elementary unit of computer information. It can assume one of two possible states, usually designated as "0" or "1".

byte — a group of 8 binary digits (bits) operated upon as a unit.

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.

word — two bytes, or a group of 16 binary digits (bits) operated upon as a unit.

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 should appear within your program 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 through the Assembly Language series of manuals —

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

Chapter 2
Modules and Routines

There are three basic activities associated with using assembled routines and modules. 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 **use** of a module, or of the routines which it contains. 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).

This chapter deals which these activities. It demonstrates how you can, within a BASIC program (or from the keyboard), use modules which have been previously created. The fundamental statements involved are —

ICOM	used to set aside memory to hold modules and routines
ILOAD	used to retrieve modules from mass storage
ICALL	used to access routines like a subprogram
ISTORE	used to save modules on mass storage
IDELETE	used to free space in memory for other modules

Modules vs. Routines

What is the difference between a "module" and a "routine"?

- A **routine** is a program intended for your use. It is callable, like a subprogram, once it is in memory.
- A **module** is a collection of one or more routines which are closely associated to one another and are considered as a unit. One or more modules may be stored on a file on mass storage.

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 the authors of the routines, and you.

The names of routines and modules are given to them by the authors of the routines. The documentation provided you should indicate the names with which you need to be concerned. You may not change these names.

The naming of files is flexible. Originally, the names were assigned by the authors of the routines. They may have been subsequently changed by others. In addition, **you** may change them, depending upon your own needs and desires. Conventions for file names and methods for file manipulation can be found in the BASIC Programming Manual and in the Mass Storage ROM manual.

Overview

To briefly sketch the functional relationships of modules and routines, please refer to Figure 2.

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.

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 away into an OPRM file (for tape cartridges) or an ASMB 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 up") by using the "IDELETE" command.

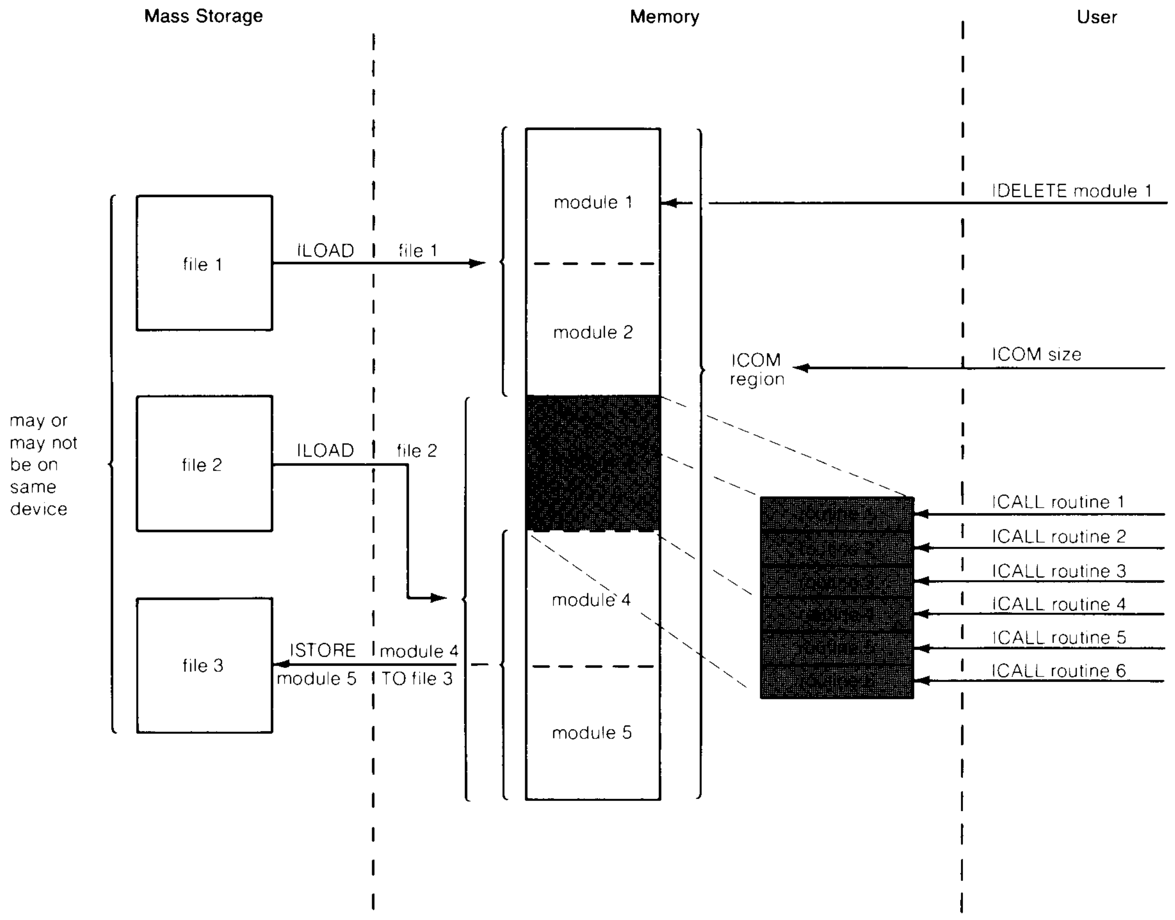


Figure 2. Overview of Routines and Modules

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 modules which you want to have in memory at any one time.

Each module contains one or more routines for your use. The number varies, depending upon what the author of the routines has provided you. 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 parameters (arguments) which you need to "pass" (send down) to the routine itself. What these parameters may be and what meaning they hold depends upon what the author had in mind. You can find out that information in the documentation provided with the routine itself.

Setting Aside Memory

As indicated by Figure 2, you cannot load a module until there is an ICOM region into which to load it.

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 can only be included as a line in a BASIC program, and cannot be executed from the keyboard. This is similar to a DIM or COM statement. The actual region is created when the program is run.

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 later.

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.

In most cases, the space which is freed up by reducing the size of the ICOM region, is returned to your available memory space. Sometimes, however, it is not returned, depending upon the status of common (the area created by the COM statements executed to that point) and other option ROMs. The space will be returned whenever —

- There was never common in existence; or,
- SCRATCH C has been executed on existing common and no COM statement has been executed since then; and,
- The requirements of another option ROM, which may be present, 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). 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 only, not in any subprogram.

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

```
20    ICOM 984
 .
 .
 .
300   ICOM 492
 .
 .
 .
610   ICOM 2000
 .
 .
 .
```

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

All ICOM statements in a program must appear **before** any COM statement. This is to assure that the ICOM region is allocated before the common is allocated.

If, after running this program, you loaded in another program with no ICOM statements in it at all, the ICOM region is still there (with the full 2 000 words).

There are only two ways to completely eliminate an ICOM region —

- Execute SCRATCH A.
- Execute ICOM 0 in a program.

After either of these, 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. The documentation provided on the routines you have loaded should tell you if there are any such active ISRs.

The ICOM 0 procedure can be used in a program to assure that all previous modules are deleted. For example, the following sequence —

```
[illegible]
```

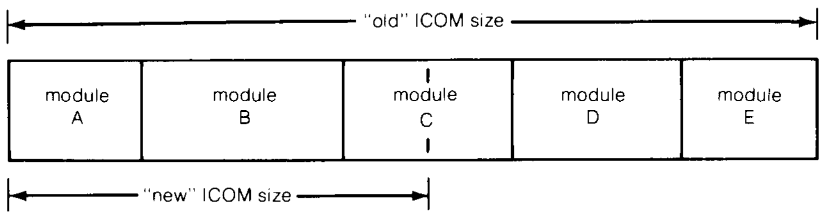
assures that an ICOM region of precisely 2 000 words is in existence at the running of the program, and one completely clear of any previously loaded modules.

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 BASIC Programming Manual for details of 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 execution 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). The documentation provided on the routines should tell you if there are any active ISRs.

Retrieving Modules

Modules are stored in files on mass 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. 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 (MSUS) of the device to form a file specifier.[2] 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.

If you do not want all the modules in a given file, but instead just a few, 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 modules 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
```

1 Some OPRM-type files are not assembly language files, but are created by other option ROMs available on the System 45. However, for those that are assembler files, they are exactly equivalent to the ASMB – type.

2 For a full discussion of file specifiers, consult the BASIC Programming Manual or the Mass Storage ROM manual.

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 it previously occupied is made available for use in loading another module. 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). The documentation provided on the routines should tell you if there are any active ISRs.

Of course, to use the IDELETE statement, you must be aware of the module names. Your source for finding these names must be the documentation provided by the authors of the modules. 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
```

Storing 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 msus given in the {file specifier} is created and the modules named 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. The file can then be used in ILOAD statements at a later time.

If you want to store **all** of the routines currently in the ICOM region into a particular file, you should use the following statement —

ISTORE; {file specifier}

Accessing Routines

A module may contain one or more routines for your use. Which routines a given module contains should be documented by the author of those routines. Once the module has been loaded, all of its routines are immediately available to you through the ICALL statement. This statement has the form —

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

This ICALL is very much like the CALL of a subprogram in BASIC. If there are arguments required by the routine, the requirements for them should be detailed in the documentation for the routine provided by the routine's author. It is necessary, when using arguments, that you follow the rules for them laid down by the author of the routine.

Thus, for example, if an author stated the following —

> "The SORT routine requires one argument, the array identifier
> of the string array to be sorted."

then the ICALL statement to be used would probably look something like this —

```
ICALL Sort (Temp$(*))
```

Upon execution of the ICALL statement, execution transfers to the routine named. Upon completion of execution of the routine, control is returned to the BASIC statement following the ICALL. This is identical in effect to the CALL statement in BASIC.

Chapter 3 Handling Interrupts

An "interrupt" is a request for service from a device connected to the computer. The actual type of service being requested depends upon the device. For instance, some devices send interrupts when they have some information they want your program to take, other devices send them when they want some information from your program. How you handle them depends upon what the device wants.

An assembly language routine which you are using may have the capability of handling interrupts from external devices. It may also inform your BASIC program of special conditions detected during the processing of an interrupt, e.g., end of interrupt service, input data error, etc. You, in turn, may take this information and cause a branch to another part of your program.

To determine whether the routine you are using handles interrupts, you should consult the documentation provided with the routine by the authors of the routine. The documentation should tell you what kind of interrupts to expect and what kind of special processing ("handling") may be required, if any, on the part of your program.

Branching on Interrupts

Since interrupts are a program-independent occurrence, the handling of an interrupt is sometimes a reason for causing the program to suspend whatever it is doing and do something else (i.e., "branch"). Like the ON KEY statement (see the 9845 BASIC Programming Manual), 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}
```

These statements are provided by the ROM in order to allow the assembly language routine to signal your BASIC program that a special condition has arisen and to indicate where it came from. When you have executed an ON INT statement and an interrupt occurs, the following sequence ensues —

1. The assembly language routine assigned to the interrupting select code services the interrupt.
2. If the assembly language ISR is so programmed, it signals BASIC that an interrupt occurred on the select code of the ISR's choosing (which may not be the one where the interrupt actually occurred).
3. Upon completion of the current BASIC line, the ON INT for the select code with the assembly language ISR defined interrupt is honored and the branch indicated for it (be it a CALL, GOSUB, or GOTO) is taken.

In the GOTO version, the branch is "absolute", which is to say that your program will go to the line indicated and pick up its execution there, forgetting where it was before. This has the effect of an "abortive" type of branch, and should be used only when you want the program to resume execution at some pre-determined point after handling the 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 will return 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} you specify with the 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 any select code and the assembly language routine may decide under some circumstances to have triggered the ON INT with some other select code value. This can be a way of allowing more than one branch for interrupts from a single interrupting device.

As an example —

```
100 ON INT #2 GOSUB Take_reading
110 ON INT #7 GOSUB Take_reading
120 ON INT #12 CALL Process_data
```

Should an interrupt occur anywhere in the program, causing the assembly language routine to indicate select codes 2 or 7, the subroutine "Take_reading" would be performed and then resume program execution at the point of interruption. Should an interrupt be received from select code 12, then the subprogram "Process_data" would be performed.

Prioritizing Interrupts

Since more than one interrupt may occur while a single BASIC statement is executing, it is possible that by the time the line finishes you may have 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" number 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 will take 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.

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 will then be executed.

[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 will be executed before an ON INT with low priority.

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 would proceed 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.

The CALL version of an ON INT is **always** in effect, whether in the main program or in any subprogram.

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. 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 Interrupt 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 will have no effect.

DISABLE and ENABLE deactivate and activate, respectively, the ON INT as well as the ON KEY and ON KBD declaratives.

Chapter 4 Errors and Error Processing

While you are using or accessing an assembly language routine, it is possible that an error may occur which is associated with your attempts to use the routine. It is intended that this chapter give some guidance as to how certain errors can be handled. It is not 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 accessing and using assembly language routines. A complete listing of error messages can be found in either the BASIC Programming manual or the ROM Reference Tables and Index.

The following list is of the messages you may receive should there be an assembly language-related problem of some sort. Possible corrective actions are included in the discussion of each error.

ERROR 1	ROM missing, or configuration error. To operate the System 45, all system ROMs and the Assembly Execution ROM must be in place. 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. Things to try to get things to fit: select a smaller ICOM size; execute SCRATCH C (if no important data remains in common), delete modules and reduce the ICOM size; segment your program. The error may also be caused by trying to load modules which are too large for the current ICOM region or by placing a COM statement before an ICOM statement.
ERROR 9	The number of arguments passes by an ICALL statement exceeds the number of parameter declarations in the subroutine entry section.

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	No ICOM region found. You have failed to create the ICOM region, or have inadvertently deleted it. Program an ICOM statement of adequate size and re-run the program.
ERROR 191	Module not found. The module indicated in an ISTORE statement is not currently resident in the ICOM region. Check your module names used in the statement to find the one which is missing.
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 has an active interrupt service routine. The only ways to allow the action to take place are to SCRATCH A (which affects a number of other things), to press CONTROL STOP, or to inactivate the ISR. To inactivate the ISR, consult the routine's documentation.
ERROR 195	Routine not found. You may have specified the wrong routine name or failed to load the correct module. Double check the documentation indicating the location and name of the routine.
ERROR 196	Unsatisfied externals. You may not have loaded all of the modules necessary to run the routine. Double check the routine's documentation for the other resources you may need. May be an error in the programming of the module, in which case check with the routine's author.
ERROR 197	Missing COM statement. The routine you have called is expecting to find or place some of its data in common, and you have not provided the COM statement required. Check the documentation for the routine to determine the common requirements.
ERROR 198	Common area does not correspond to 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 the variables correctly in either type or size. Check the documentation for the routine to determine the common requirements.

ERROR 199 Insufficient number of items in BASIC COM declarations. The routine you have called is expecting to find or place some of its data in common, but your COM statement does not provide enough variables to satisfy the routine's needs. Check the documentation for the routine to determine the common requirements.

Subject Index

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.