MCS™-86 ASSEMBLER OPERATING INSTRUCTIONS FOR ISIS-II USERS

Manual Order No. 9800641A

Copyright © 1978 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

ICE	LIBRARY MANAGER	PROMPT
INSITE	MCS	RMX
INTEL	MEGACHASSIS	UPI
INTELLEC	MICROMAP	μSCOPE
iSBC	MULTIBUS	

PRINTED IN U.S.A/A32/379/10K/CP

PREFACE

This manual is directed to those of you who have read the MCS-86 Assembly Language Reference Manual, have coded your program and are ready to run the MCS-86 Assembler. This manual covers the six basic controls that this version of the assembler has and how to use them. Included in these pages are the error messages and how to recover from the conditions that caused them. Information relative to this document that might prove helpful includes these manuals:

MCS-86 Assembly Language Reference Manual	9800640
MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users	9800639
MCS-86 User's Manual	9800722
MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users	9800642
ISIS-II User's Guide	9800306
PL/M-86 Compiler Operator's Manual for ISIS-II Users	9800478

CONTENTS

ILLUSTRATIONS & TABLES

CHAPTER 1
BEFORE USING THE MCS-86 ASSEMBLER

If this is the first time that you have used the MCS-86 assembler, check your Intellec Microcomputer Development System (MDS) for these items as they are required for assembler operation:

64K RAM memory
a console device, such as a CRT or a TTY
(at least) one disk drive and
ISIS-II with a version number of 2.2 or later

You may want to add a lineprinter to this configuration, however, this is the minimum configuration for the MCS-86 assembler. For more detailed information on how to use the MDS, consult the ISIS-II User's Guide.

Next, check that the software that you need, the MCS-86 assembler and its overlays, is on a diskette. You can leave it on that diskette or you can copy it to the system diskette, as your needs dictate. Another important piece of software to check for is the Relocation and Linkage software, either QRL-86 or LINK86, and LOCate86. Be sure to label the diskette appropriately for ease of identification later on. This is a list of the primary pieces of software needed and to point out that it is wise to know what it is that you have and where it is.

Have your MCS-86 Assembly Language Reference Manual nearby for that document and this one are interdependent. This manual assumes that you have the knowledge gained from study of the language manual. For instance, the successful use of the assembler is more likely if you are familiar with the directives, SEGMENT and ASSUME. Since you will need to use at least some of the Relocation and Linkage software package, the manual containing that information should also be nearby (and have been read).

This manual instructs you in the use of the MCS-86 assembler through the use of the six basic controls. It is according to these controls that the assembler creates an object file. Assembly language instructions are converted to object code. In addition, the assembler creates an assembly listing file.

This listing file contains information on your code, your source file, a summary of assembly errors, if any, and the symbols that you have defined in your source program.

During assembly, the MCS-86 assembler creates files for its own use that you need to know about. Their names are not crucial to your program design and execution but their location (on the diskette) is, for they are always created (and deleted from) the diskette which contains the assembler. There is a restriction on the location of the assembler and its overlays, for they (ASM86, ASM86.OV0, ASM86.OV1, ASM86.OV2), must be on the same diskette. The diskette may be on any drive. There is no restriction as to where the listing and the object files are directed, but the default conditions send both these files to the same drive as the source file.

The MCS-86 assembler is a two pass assembler. The first pass reads your source program and produces the temporary file, ASM86I.TMP. This file provides communication between pass 1 and pass 2. If you want your list file to contain your symbol table, ASM86X.TMP is created by the overlay file, ASM86.OV1. Then ASM86.OV1 sets up the object file and the symbol table. In pass 2 the source file is completely read and formatted for printing. This final pass resolves all forward references and generates the object file, listing file and error messages. Both passes

catch errors but output occurs only after pass 2. The temporary files, ASM86I.TMP and ASM86X.TMP, are deleted at the end of pass 2. Figure 1-1 displays the MCS-86 assembler and its related files.

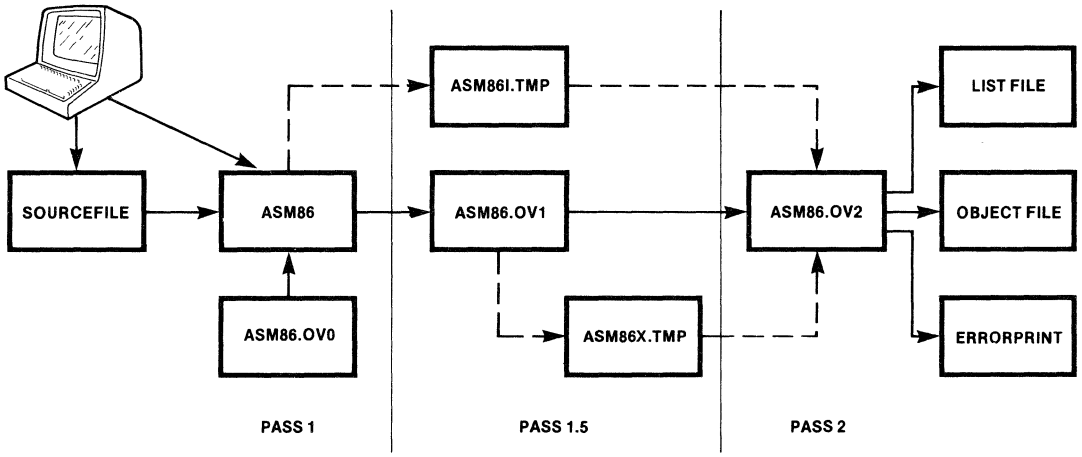


Figure 1-1. MCS-86 Assembler Logical Files

CHAPTER 2
INVOKING THE MCS-86 ASSEMBLER

To invoke the MCS-86 assembler, enter this command at the keyboard:

– [:drivenumber:]ASM86 [:drivenumber:]yourprogram

In the above format of invoking the assembler, the hyphen is the ISIS-II command prompt that signals that the operating system is ready to accept a command (in this case a command to load and execute the assembler.) The phrase [:drivenumber:] that appears twice in this command indicates where the assembler resides and where the source file resides if those locations are other than drive 0. The square brackets [ ] indicate that drivenumber is an optional item in the command. ASM86 must be in the command line as it is the filename of the assembler program itself. The final item is your program, which is the filename of your program to be assembled. This command line does not have controls indicated on it; for a detailed discussion of the controls see the next chapter.

Immediately after you enter the command line, the assembler sends its sign-on message to the console:

ISIS-II MCS-86 ASSEMBLER, V1.0

When the assembly of your program is complete, the sign-off message and error summary are sent to the console in this format

ASSEMBLY COMPLETE,NO ERRORS FOUND

If the assembler detected errors, the message is:

ASSEMBLY COMPLETE,1 ERROR FOUND

(The number of errors quoted varies according to the number of errors encountered.)

It is likely that you will want to use the assembler as specified above for it is the default operation, an automatic mode of operation.

There are some restrictions however that you need to know about. Since these restrictions are mostly quantitative they are noted here in table form.

Table 2-1. MCS-86 Assembler Parameters (Rules of Thumb)

Source file:	
Item	**Number**
Characters/line	128 (including CR/LF); if more are entered, they are processed but not listed and an error message noted.
Characters/ID	31; if more are entered, they are ignored.
Symbols/module	400 (approximately); relative to the length of the name and type of symbol.
Source lines/program	No assembler imposed limit.
Cont. lines/statement	No assembler imposed limit.
Characters/string	40
Characters/classname	40

Table 2-1. MCS-86 Assembler Parameters (Rules of Thumb) (Cont'.d)

Source file:

Item	Number
PROC/SEG nesting	16 (up to 16 total open at one time).
Items/PUBLIC,EXTRN,PURGE	No assembler imposed limits.
Items/GROUP	36 SEGs per GROUP
Codemacro size	60 bytes (approximately) of assembler generated code.
SEGMENT or PROC size	No assembler imposed limits.
Record limit	16 fields.
Record size	16 bits.

Listing file:

Item	Number
Characters/list file line	120; if more are entered, wraparound* occurs.
Pages/file	No assembler imposed limit.
Lines/file	No assembler imposed limit.
Errors reported/line	10

Internal:

Item	Number
Items/storage initialization-list	Cannot exceed 16.
Expression complexity	Relative to stack size; with normal operation, this is not a practical consideration
Memory required	64K
Dup nesting	Up to 8 levels of nested parentheses.
Intermediate file size.	
ASM86I.TMP	Proportional to source file size
ASM86X.TMP	30 (approximate) characters per user-defined symbol.

*Wraparound allows you to enter more than 120 characters on a list file line; it places the characters that are in excess on the next line.

CHAPTER 3
DEFINING ASSEMBLY CONDITIONS

List of Controls

There are six basic controls for the MCS-86 assembler that define files or identify devices and generally act as programming aids. Each control is a keyword that is typed on the same line as the line invoking the MCS-86 assembler. Each control can be on or off depending on what kind of (optional) output you require and where you want it. These controls can be entered as many times as you like. The assembler ignores all but the final (rightmost) definition of each control. If however, a control is other than the ones noted in this chapter, then the assembler terminates and the command must be re-entered. The ISIS-II restriction on the length of lines that invoke a program applies. See your ISIS-II User's Guide if you are not familiar with it.

The six controls are listed here:

OBJECT or NOOBJECT	(to create or not to create an object file)
PRINT or NOPRINT	(to create or not to create a listing file, often called a list file or a print file)
PAGING or NOPAGING	(to create or not to create numbered pages in that listing file noted above)
SYMBOLS or NOSYMBOLS	(to append or not append the symbol table at the end of the listing file)
NOERRORPRINT or ERRORPRINT	(not to create or to create a list only of the errors encountered by the assembler)
NODEBUG or DEBUG	(not to put or to put all local symbol information in object file for relocation and linkage software)

Default Controls

In this list, the left column of controls is the default or automatic mode of operation. These are the controls that are invoked with the invocation of the asembler. If you want to define conditions other than those, you must specify them in the command.

The formal syntax, then, for invoking the MCS-86 assembler with controls is:

-[:devicenumber:]ASM86 [:devicenumber:]yourprogramname [control-list]

Remember, if you choose not to enter any controls from the above control-list the default controls are implied, i.e., they are automatic. This means that the controls on the left side of the list are in effect. If, however, you specify any control from above, it must be entered as shown. How to shorten those controls is in table 3-1. Controls must be separated by at least one blank character, either a space or a tab. Further, if the control specifies a file then the filename must be in parenthesis. The ISIS-II conventions on filenames and devicenames (diskettes) apply here, also. See your ISIS-II User's Guide if you are not familiar with them.

An example of invoking the MCS-86 assembler is

-ASM86 LOOT.SRC OBJECT PRINT PAGING SYMBOLS

The command generates the object file LOOT.OBJ on drive 0, creates a listing file LOOT.LST with paged information and symbol table and error messages within the body of the list file. It is identical to this command:

-ASM86 LOOT.SRC

This is due to the automatic or implicit invocation of the default controls. It is wise to note that the object and listing files are created on the same drive as the source file.

Explicit Controls

If you want to define filenames or devices other than the default, you must specify them in the command. An example of specifying filenames is to define output files with different names from the input file as this command does

```
-ASM86 LOOT.SRC OBJECT (COOT.OBJ) PRINT (COOT.LST) PAGING
```

Here you obtain object and listing files called COOT.OBJ and COOT.LST, respectively. Filenames are indicated by parentheses; if you include a filename without a parenthesis the command cannot be read correctly.

SUMMARY: TO INVOKE THE ASSEMBLER,GENERATE AN OBJECT FILE AND CREATE A PAGED ASSEMBLY LISTING FILE IS THE DEFAULT OPERATION. THE INVOCATION COMMAND IN ITS MOST ABBREVIATED FORM IS

```
-ASM86 YOURPROGRAM
```

THIS IS THE COMMAND IN ITS MOST EXPLICIT FORM

```
-:DEVICENUMBER:ASM86 :DEVICENUMBER:YOURPROGRAM CONTROL (FILENAME)
```

In programming, however, often you must consider both time and storage or memory space when you assemble your program. You may require flexibility beyond that in the default conditions. With this in mind, you can direct the files created during assembly to be located on different devices or drives, as your system permits. If storage space is at a premium and this is your first pass at assembling what may amount to a large program, you probably do not want to produce object code in your first assembly; this command accomplishes that

```
-ASM86 LOOT.SRC NOOBJECT
```

With this command you verify your program without wasting time or space. Another example of negating the default control is

```
-ASM86 LOOT.SRC NOPRINT
```

This command suppresses the creation of the listing file. As a further consequence it also invalidates ERRORPRINT, PAGING and SYMBOLS.

If you want your object code to reside on a drive other than the one that contains your source code, this command does it

```
-ASM86 LOOT.SRC OBJECT (:F1:LOOT)
```

The object file in the above example is on drive 1, the source file is on drive 0. Similarly, if you want your object code to have a filename distinguished from the source code entirely, you must specify it

```
-ASM86 LOOT.SRC OBJECT (COOT)
```

If you want the listing file generated and output to a device other than the default (the same device as the source file) you specify another drive, or as in this case, the lineprinter

```
-ASM86 LOOT.SRC PRINT (:LP:)
```

This results in the list file printing on the lineprinter.

It is likely that a listing file with a different name than the source file could be a great convenience. Here is the command that accomplishes this:

```
-ASM86 LOOT.SRC PRINT (:F1:SOOT.LST)
```

If you want a summary of errors but not a listing file this is the command:

```
-ASM86 LOOT.SRC PRINT(:BB:) ERRORPRINT
```

Note that the :BB: is the "byte bucket"; ISIS-II ignores I/O commands from and to this "device". It is a null device.

SUMMARY: TO "TURN OFF" OR TO OVERRIDE A DEFAULT CONTROL, YOU MUST INCLUDE EXPLICITLY THE CONTROL, THE DEVICE, OR THE FILENAME THAT IS DISTINCT.

Summary of All Assembler Controls

The following table summarizes the controls and their shortened forms for the MCS-86 Assembler.

Table 3-1. MCS-86 Assembler Controls Summary

OBJECT[(FILE.EXT)] OJ	An object code file is generated and is output to the specified diskette file. If neither OBJECT nor NOOBJECT is put on the command line, OBJECT, as the default, is assumed. The result is that the object file has the same filename as the source file with an extension of OBJ and resides on the same diskette as the source file.
NOOBJECT NOOJ	No object code is generated. NODEBUG is implied.
PRINT[(FILE.EXT)] PR	A listing file is generated and is output to the specified file or device. If neither PRINT nor NOPRINT is put on the command line, PRINT, as the default is assumed. The result is that the listing file has the same name as the source file and resides on the same diskette as the source file.
NOPRINT NOPR	The listing file is suppressed. The result is that NOERRORPRINT, NOPAGING, NOSYMBOLS are implied.
PAGING PI	The listing file is formatted into numbered pages with headers at each page break. If neither PAGING or NOPAGING is put on the command line, PAGING, as the default, is assumed. The result is that the listing file has sequentially numbered pages with header information at the top of each page. PAGING is suppressed if NOPRINT is put on the command line.
NOPAGING NOPI	The listing file does not have formatted pages. The symbol table is separated from the source file by 4 lines.
SYMBOLS SB	The symbol table appears at the end of the listing file. If neither SYMBOLS or NOSYMBOLS is put on the command line, SYMBOLS, as the default, is assumed. The result is that at the end of the listing file is the alphabetized symbol table. SYMBOLS is suppressed if NOPRINT is put on the command line.
NOSYMBOLS NOSB	No symbol table appears at the end of the listing file.

Table 3-1. MCS-86 Assembler Controls Summary (Cont'd.)

NOERRORPRINT[(FIL E)] NOEP	No summary of the errors encountered appears. If neither NOERRORPRINT or ERRORPRINT is put on the command line, NOERRORPRINT, as the default, is assumed.
ERRORPRINT[(FILE)] EP	A list that summarizes the errors encountered in assembly concludes the listing file. ERRORPRINT is supressed if NOPRINT has been put on the command line. If you have specified :CO: as the device where the file is to be created, then no header appears.
NODEBUG NODB	No local symbol information is placed in the object file. If neither DEBUG or NODEBUG is put on the command line, NODEBUG, as the default, is assumed.
DEBUG DB	Local symbol information is placed in the object file for symbolic debugging.

CHAPTER 4
LISTING FILE AND THE ERRORPRINT FILE

The Listing File

The listing file, often called the list file or print file, provides you with information on the assembly of your program. As a programming tool, it presents both assembler generated information and user generated information. The wealth of information possible in the listing file is great; the entire span of the MCS-86 assembly language can be contained within its pages. You most often will consult it as a debugging tool; however, you will also find that it exists as an educational tool. To serve these purposes and yours better it has a format that is suitable for hardcopy documentation.

The example in this chapter contains some of the most used features of the MCS-86 assembly language; however, it does not cover all of them. Use this example to identify where and how you might find information in the list file. As you use this chapter it is important to note that the primary purpose of this example is to illustrate the list file; it is not intended as an example of excellent programming techniques.

Generally speaking, the listing file contains your program and your symbol table. An error summary concludes the listing file in addition to the object code.

Header

Header information is at the top of the page. It identifies the assembler program name and the page number. The width of the page of the list file is 120 columns; the length of the page (if PAGING has been specified) is 60 lines long. Figure 4-1 notes the header lines, width and length of the list file.

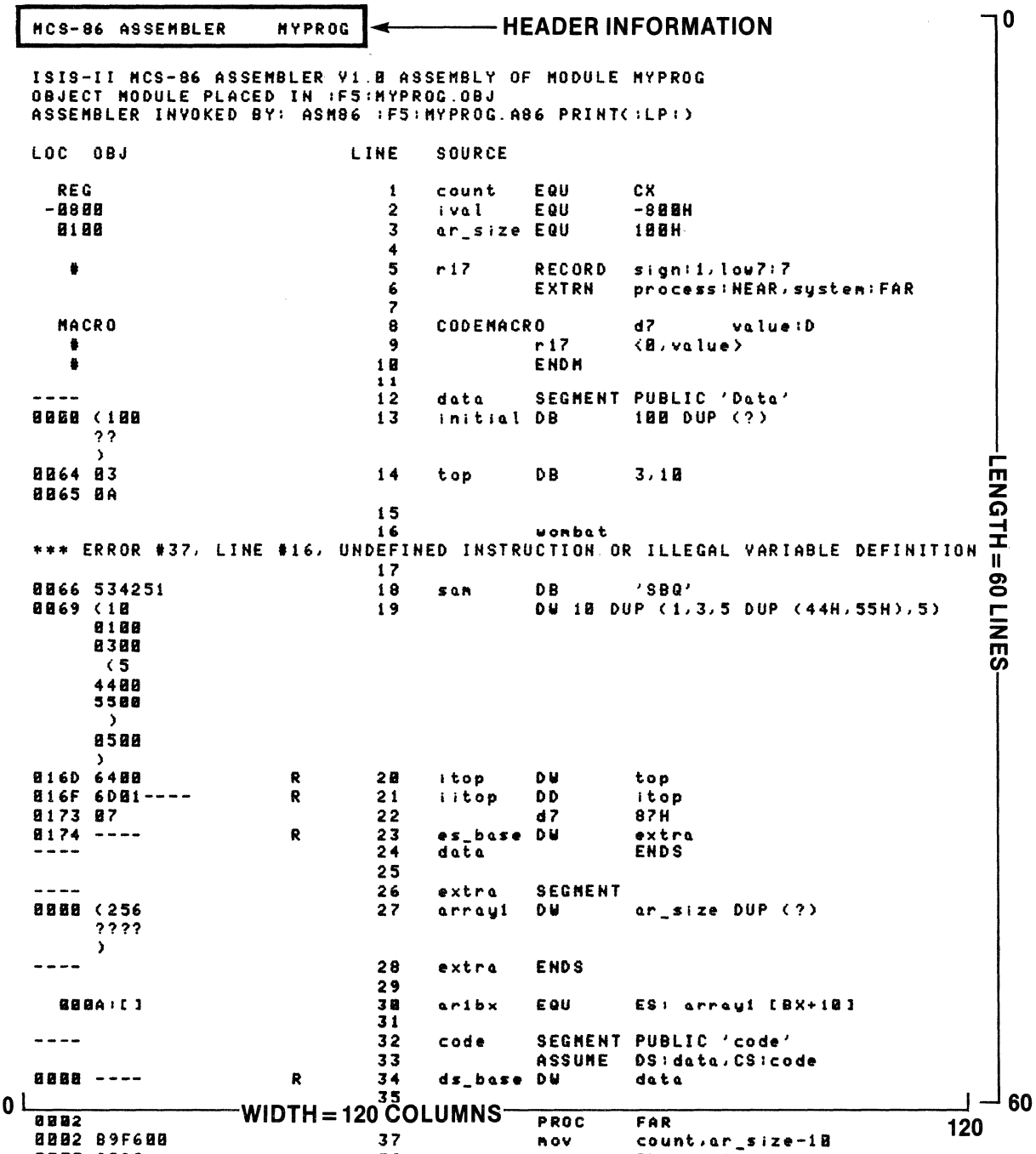
Additional headerlines display this information.

```
ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE YOURPROGRAM
OBJECT MODULE PLACED IN :DEVICENUMBER:YOURPROGRAM.OBJ
ASSEMBLER INVOKED BY :DEVICENUMBER:YOURPROGRAM.86S
```

Beneath the headerlines is another line that prints out the names of the fields of information. Strictly speaking these are known as fields of information; in visual terms it is easier to see them as columns. Because there is so much information, it is helpful to think of it in these broad terms:

- any information to the left of the line number is assembler generated
- any information to the right of the line number is user generated.

Figure 4-2 notes the fields of information in the list file.



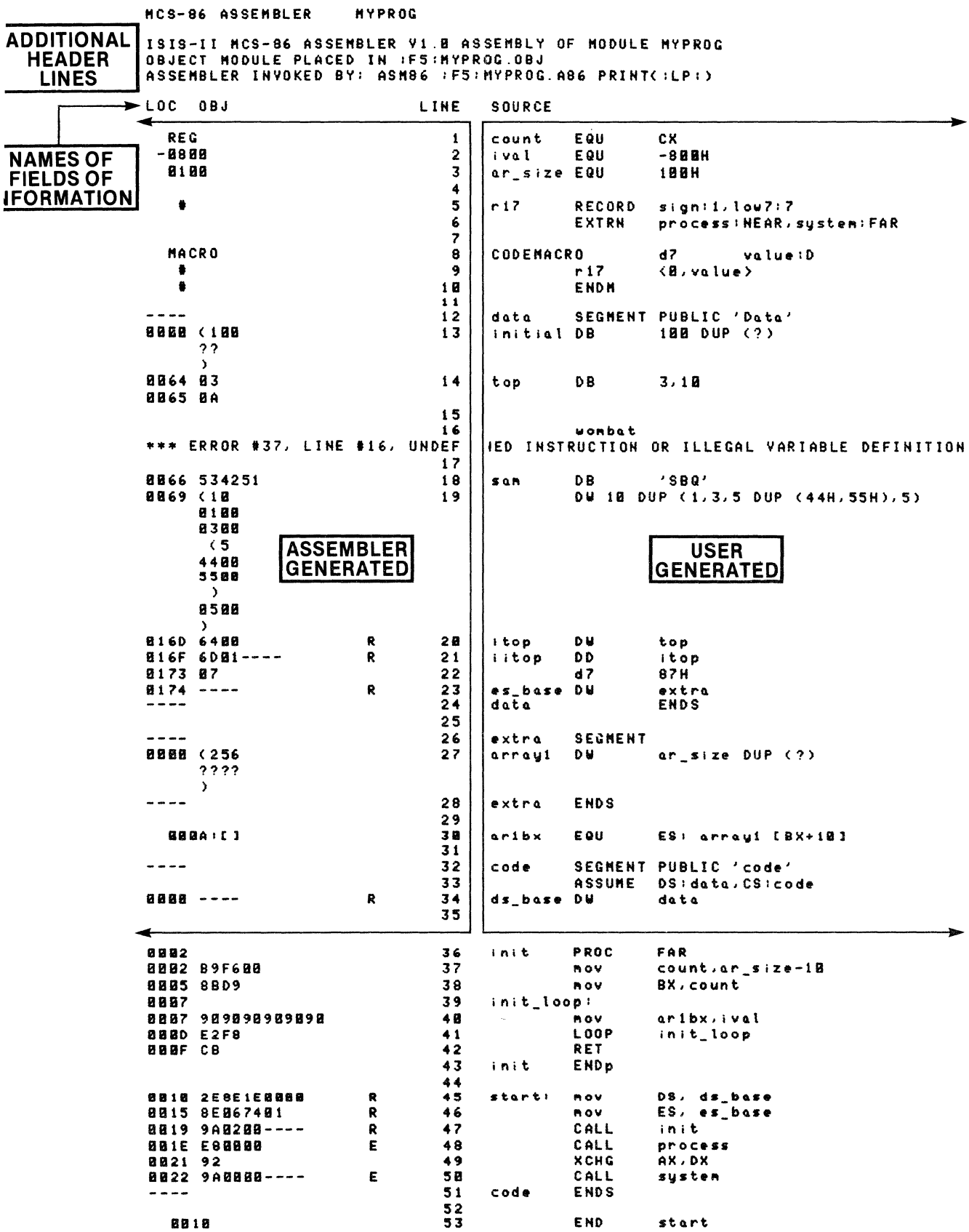

```
MCS-86 ASSEMBLER    MYPROG

ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE MYPROG
OBJECT MODULE PLACED IN :F5:MYPROG.OBJ
ASSEMBLER INVOKED BY: ASM86 :F5:MYPROG.A86 PRINT(:LP:)

LOC  OBJ                    LINE   SOURCE

  REG                          1   count   EQU     CX
 -0800                         2   ival    EQU     -800H
  0100                         3   ar_size EQU     100H
                               4
   #                           5   r17     RECORD  sign:1,low7:7
                               6           EXTRN   process:NEAR,system:FAR
                               7
  MACRO                        8   CODEMACRO       d7      value:D
   #                           9           r17     (0,value)
   #                          10           ENDM
                              11
----                          12   data    SEGMENT PUBLIC 'Data'
0000 (100                     13   initial DB      100 DUP (?)
     ??
     )
0064 03                       14   top     DB      3,10
0065 0A
                              15
                              16           wombat
*** ERROR #37, LINE #16, UNDEFINED INSTRUCTION OR ILLEGAL VARIABLE DEFINITION
                              17
0066 534251                   18   sam     DB      'SBQ'
0069 (10                      19           DW 10 DUP (1,3,5 DUP (44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
016D 6400           R         20   itop    DW      top
016F 6D01----       R         21   iitop   DD      itop
0173 07                       22           d7      07H
0174 ----           R         23   es_base DW      extra
----                          24   data            ENDS
                              25
----                          26   extra   SEGMENT
0000 (256                     27   array1  DW      ar_size DUP (?)
     ????
     )
----                          28   extra   ENDS
                              29
  000A:[]                     30   arlbx   EQU     ES: array1 [BX+10]
                              31
----                          32   code    SEGMENT PUBLIC 'code'
                              33           ASSUME  DS:data,CS:code
0000 ----           R         34   ds_base DW      data
                              35
0002                          36           PROC    FAR
0002 B9F600                   37           mov     count,ar_size-10
0005 8BD9                     38           mov     BX,count
0007                          39   init_loop:
0007 909090909090             40           mov     arlbx,ival
000D E2F8                     41           LOOP    init_loop
000F CB                       42           RET
                              43   init    ENDp
                              44
0010 2E8E1E0000     R         45   start:  mov     DS, ds_base
0015 8E067401       R         46           mov     ES, es_base
0019 9A0200----     R         47           CALL    init
001E E80000         E         48           CALL    process
0021 92                       49           XCHG    AX,DX
0022 9A0000----     E         50           CALL    system
----                          51   code    ENDS
                              52
  0010                        53           END     start
```

Figure 4-1. The List File




```
MCS-86 ASSEMBLER    MYPROG

ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE MYPROG
OBJECT MODULE PLACED IN :F5:MYPROG.OBJ
ASSEMBLER INVOKED BY: ASM86 :F5:MYPROG.A86 PRINT(:LP:)

LOC  OBJ                    LINE   SOURCE

  REG                          1   count   EQU     CX
 -0800                         2   ival    EQU     -800H
  0100                         3   ar_size EQU     100H
                               4
   #                           5   r17     RECORD  sign:1,low7:7
                               6           EXTRN   process:NEAR,system:FAR
                               7
  MACRO                        8   CODEMACRO       d7      value:D
   #                           9           r17     <0,value>
   #                          10           ENDM
                              11
----                          12   data    SEGMENT PUBLIC 'Data'
0000 (100                     13   initial DB      100 DUP (?)
     ??
     )
0064 03                       14   top     DB      3,10
0065 0A
                              15
                              16           wombat
*** ERROR #37, LINE #16, UNDEF    IED INSTRUCTION OR ILLEGAL VARIABLE DEFINITION
                              17
0066 534251                   18   sam     DB      'SBQ'
0069 (10                      19           DW 10 DUP (1,3,5 DUP (44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
016D 6400              R      20   itop    DW      top
016F 6D01----          R      21   iitop   DD      itop
0173 07                       22           d7      87H
0174 ----              R      23   es_base DW      extra
----                          24   data            ENDS
                              25
----                          26   extra   SEGMENT
0000 (256                     27   array1  DW      ar_size DUP (?)
     ????
     )
----                          28   extra   ENDS
                              29
  000A[BX]                    30   arlbx   EQU     ES: array1 [BX+10]
                              31
----                          32   code    SEGMENT PUBLIC 'code'
                              33           ASSUME  DS:data,CS:code
0000 ----              R      34   ds_base DW      data
                              35

0002                          36   init    PROC    FAR
0002 B9F600                   37           mov     count,ar_size-10
0005 8BD9                     38           mov     BX,count
0007                          39   init_loop:
0007 909090909090             40           mov     arlbx,ival
000D E2F8                     41           LOOP    init_loop
000F CB                       42           RET
                              43   init    ENDp
                              44
0010 2E8E1E0000        R      45   start:  mov     DS, ds_base
0015 8E067401          R      46           mov     ES, es_base
0019 9A0200----        R      47           CALL    init
001E E80000            E      48           CALL    process
0021 92                       49           XCHG    AX,DX
0022 9A0000----        E      50           CALL    system
----                          51   code    ENDS
                              52
  0010                        53           END     start
```

Figure 4-2. Fields of Information in the List File

Body

The body consists of columns of information organized typically as previously described. The following is a discussion of the specifics of the information displayed.

These names identify the fields of information: LOC, the location counter; OBJ, the object code; LINE, the line number and SOURCE, the line of source code.

They appear in this format:

```
LOC  OBJ                    LINE   SOURCE
```

LOC

The locations counter is the hexadecimal number that represents the offset from the beginning of the SEGMENT being assembled. In lines that generate object code and have LABEL or PROC, the value is the one at the beginning of the line. For ORG lines, the value shown is the new value. The following line of a list file displays a line that has a PROC.

```
LOC  OBJ                    LINE   SOURCE

0002                          36   init    PROC    FAR
```

For any other line (such as the second or third line in a Dup construction or a continuation line) there is no display, as is shown in the following figure.

```
0069 (10                      19            DW 10 DUP (1,3,5 DUP (44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
```

If there is '----'in the LOC field, you have coded either an open or close SEGMENT, as in the illustration that follows.

```
----                          26   extra   SEGMENT
```

If the LOC area is blank, either a directive or a comment has been encountered by ASM86. In this case the directive ASSUME, has been coded.

```
                              33            ASSUME  DS:data,CS:code
```

OBJ

The object code is the hexadecimal number that displays the object bytes generated in the assembly. If there is '----' in this column, this indicates relocatable paragraph numbers. To the right of the OBJ field of information can be found either an R or an E or a blank area. R indicates relocatable code has been generated; E that external code had been generated. An E takes precedence over an R on lines with both kinds of code. The following figure illustates the location of the dashes and E and R.

```
0015 8E067401        R       46              mov     ES, es_base
0019 9A0200----      R       47              CALL    init
```

Object code generated by Dups constructs has a special format. Whenever a DUP field begins, a left parens appears in the left column of the object field, followed by the count in decimal numbers. The content bytes are presented left-justified on the following lines, concluded with a right paren in the leftmost column. These bytes appear reversed here since the 8086 reverses bytes. For nested Dups, the left paren number and right paren are indented one column for each nesting level, but the content bytes are never indented.

```
0069 (10                        19              DW 10 DUP (1,3,5 DUP (44H,55H),5)
     0100
     0300
      (5
     4400
     5500
      )
     0500
     )
```

EQUATE

This field is not named as such but is composed of one-half of the LOC field and one-half of the OBJ field. Basically, if the information that you are looking at is aligned with column three on your listing file, you've got EQUATE information. REG appears here if the right side of the EQU is a register.

```
LOC  OBJ                    LINE    SOURCE

  REG                          1    count   EQU     CX
```

Variable or label equates can have segment override and indexing attributes here; a colon after the number (the value) signals an override, the square brackets signals an index attribute.

```
  000A:[ ]                     30    aribx   EQU     ES: array1 [BX+10]
```

You can equate to MACRO, RECORD, RFIELD, EXTRN, SEGMENT and GROUP and they can all appear in this field. In the following example the equate field contains # to indicate the continuation of the codemacro definition beyond the first line.

```
  MACRO                         8   CODEMACRO       d7      value:D
   #                            9           r17     <0,value>
   #                           10           ENDM
```

The pound sign, #, also indicates a record definition as is noted below.

```
   #                            5   r17     RECORD  sign:1,low7:7
```

The equate field can contain a negative number as in indicated in this illustration of an equate to a negative number.

```
-0800                           2    ival    EQU     -800H
```

Line

The line number, the decimal number indicating each input line, starting from 1 and incrementing with every source line. If there is no information listed, the number increases by one anyway.

```
-0800                           2    ival    EQU     -800H
 0100                           3    ar_size EQU     100H
                                4
```

Source

A copy of the source line that you entered except for tabs and illegal non-printing characters. For ease of reading in this list file, tabs are expanded with sufficient numbers of blank spaces to place the character (that you entered) immediately after the tab to column 1 modulo 8. For those of you who still hesitate when you read a phrase like that, this means columns 9, 17, 25 etc. What is accomplished is that the source code information remains within the column noted as SOURCE.

```
----                            26   extra   SEGMENT
0000 (256                       27   array1  DW      ar_size DUP (?)
     ????
     )
----                            28   extra   ENDS
```

Errors are included in the list file in the exact order in which they occurred. They are documented by error number, line number, (pass number if other than the first pass), and error message. Explanatory text detailing recovery from error conditions is in Appendix A.

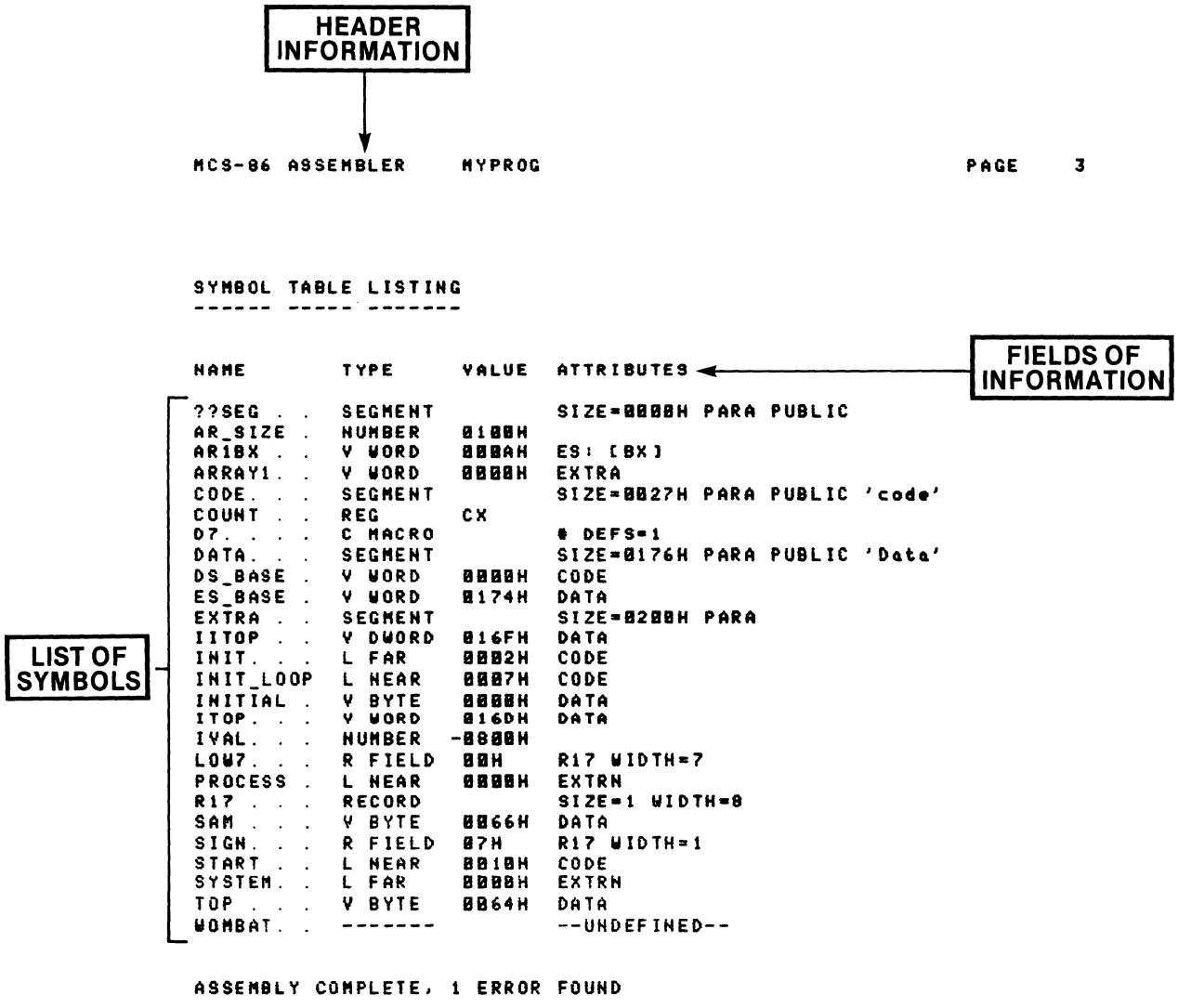
```
                                15
                                16           wombat
*** ERROR #37, LINE #16, UNDEFINED INSTRUCTION OR ILLEGAL VARIABLE DEFINITION
                                17
0066 534251                     18   san     DB      'SBQ'
```

Symbol Table

The symbol table follows the listing of the source and object codes. It is preceded by either four blank lines (NOPAGING control) or one entire blank page (if you have selected the PAGING control). Header information identifies the MCS-86 Assembler YOURPROGRAM and the page number. The listing itself is documented as the SYMBOL TABLE LISTING. Beneath that title there are columns of information; they are:

NAME TYPE VALUE ATTRIBUTES

The list of symbols is organized in alphabetic order, using the ASCII ordering of characters except for underscore which comes first. Reserved names are not included unless they were redefined in some way.




```
MCS-86 ASSEMBLER    MYPROG                                  PAGE    3


SYMBOL TABLE LISTING
------ ----- -------


NAME          TYPE      VALUE   ATTRIBUTES
??SEG . .     SEGMENT           SIZE=0000H PARA PUBLIC
AR_SIZE .     NUMBER    0100H
ARIBX . .     V WORD    000AH   ES: [BX]
ARRAY1. .     V WORD    0000H   EXTRA
CODE. . .     SEGMENT           SIZE=0027H PARA PUBLIC 'code'
COUNT . .     REG       CX
D7. . . .     C MACRO           # DEFS=1
DATA. . .     SEGMENT           SIZE=0176H PARA PUBLIC 'Data'
DS_BASE .     V WORD    0000H   CODE
ES_BASE .     V WORD    0174H   DATA
EXTRA . .     SEGMENT           SIZE=0200H PARA
IITOP . .     V DWORD   016FH   DATA
INIT. . .     L FAR     0002H   CODE
INIT_LOOP     L NEAR    0007H   CODE
INITIAL .     V BYTE    0000H   DATA
ITOP. . .     V WORD    016DH   DATA
IVAL. . .     NUMBER    -0800H
LOW7. . .     R FIELD   00H     R17 WIDTH=7
PROCESS .     L NEAR    0000H   EXTRN
R17 . . .     RECORD            SIZE=1 WIDTH=8
SAM . . .     V BYTE    0066H   DATA
SIGN. . .     R FIELD   07H     R17 WIDTH=1
START . .     L NEAR    0010H   CODE
SYSTEM. .     L FAR     0000H   EXTRN
TOP . . .     V BYTE    0064H   DATA
WOMBAT. .     -------           --UNDEFINED--


ASSEMBLY COMPLETE, 1 ERROR FOUND
```

Name

The name of the symbol appears here as it was entered; periods and spaces are added to fill out the field if the name is too short. A name may be up to 31 characters long.

```
??SEG . .   SEGMENT          SIZE=0000H PARA PUBLIC
AR_SIZE .   NUMBER   0100H
```

Type

This is the kind of symbol that you have defined and it may be any of these:

ABS, BYTE, WORD, DWORD for variables (V), NEAR, FAR for labels (L), NUMBER for numbers, REG for registers, C MACRO for codemacros, ------- for an undefined symbol and SEGMENT, RECORD, GROUP RFIELD can appear here.

External symbols have the type that appears in the EXTRN statement. This area of information may be shifted to accomodate the length of the NAME.

```
AR1BX . .   V WORD   000AH  ES: [BX]
CODE. . .   SEGMENT          SIZE=0027H PARA PUBLIC 'code'
COUNT . .   REG      CX
D7. . . .   C MACRO          # DEFS=1
INIT. . .   L FAR    0002H   CODE
INIT_LOOP   L NEAR   0007H   CODE
INITIAL .   V BYTE   0000H   DATA
IVAL. . .   NUMBER  -0800H
LOW7. . .   R FIELD  00H     R17 WIDTH=7
R17 . . .   RECORD           SIZE=1 WIDTH=8
WOMBAT. .   -------          --UNDEFINED--
```

Value

Variables and labels have their offset written as a hexadecimal number that contains the value of the number, not the value of the offset. (The value can be negative.)

```
IITOP . .   V DWORD  016FH   DATA
INIT. . .   L FAR    0002H   CODE
```

RFIELDS have the shift count for the record field as shown on the next line.

```
LOW7. . .   R FIELD  00H     R17 WIDTH=7
```

If the VALUE is blank, you have coded one of these items: SEGMENT, GROUP, C MACRO,RECORD or an undefined symbol.

```
??SEG . .   SEGMENT          SIZE=0000H PARA PUBLIC
R17 . . .   RECORD           SIZE=1 WIDTH=8
WOMBAT. .   -------          --UNDEFINED--
```

EXTRN symbols always have the value of 0000H as is shown in the following figure.

```
PROCESS .   L NEAR   0000H   EXTRN
SYSTEM. .   L FAR    0000H   EXTRN
```

Attribute

If you have coded a SEGMENT (see the TYPE column) the ATTRIBUTE field contains a hexadecimal number for the number of bytes contained in the segment.

```
CODE. . .   SEGMENT          SIZE=0027H PARA PUBLIC 'code'
```

Following the size is an additional kind of attribute information. Alignment specification is indicated by any of these terms: PARA (for paragraph), PAGE, INPAGE, BYTE or WORD.

```
DATA. . .   SEGMENT          SIZE=0176H PARA PUBLIC 'Data'
EXTRA . .   SEGMENT          SIZE=0200H PARA
```

Relocatability distinctions follow the alignment specifications and can be any of these possibilities:

blank, PUBLIC, COMMON, ABS, MEMORY, STACK

```
CODE. . .   SEGMENT          SIZE=0027H PARA PUBLIC 'code'
DATA. . .   SEGMENT          SIZE=0176H PARA PUBLIC 'Data'
EXTRA . .   SEGMENT          SIZE=0200H PARA
```

As you may have noted in the preceding illustration a classname follows the relocation information. That classname is indicated by single quotes.

```
CODE. . .   SEGMENT          SIZE=0027H PARA PUBLIC 'code'
DATA. . .   SEGMENT          SIZE=0176H PARA PUBLIC 'Data'
```

If your symbol type is a variable or label, the Attribute field contains the name of the segment with the symbol definition.

```
INIT. . .   L FAR     0002H  CODE
INIT_LOOP   L NEAR    0007H  CODE
INITIAL .   V BYTE    0000H  DATA
ITOP. . .   V WORD    016DH  DATA
```

If your symbol type is RECORD, the Attribute field indicates the number of bytes and number of bits (the width) required for that record.

```
R17 . . .   RECORD           SIZE=1 WIDTH=8
```

The following table summarizes the information that can be found and interpreted in the symbol table.

Table 4-1. Symbol Table Information

NAME	TYPE	VALUE	ATTRIBUTES
	L NEAR L FAR V BYTE V WORD V DWORD V ABS	offset from segment in which it was defined (in hex)	1. segment name or group name 2. PUBLIC or EXTRN or blank
	NUMBER	value of nbr (in hex)	1. RELOC or blank 2. PUBLIC or blank
	SEGMENT		1. SIZE=nnnnH 2. relocatability: blank, UNDEFINED, PUBLIC, ABS, MEMORY, STACK, COMMON 3. align type: PARA=paragraph, PAGE, INPAGE, BYTE, WORD 4. classname: '-------'
	REG	any register	
	C MACRO		# DEFS=nnn(decimal)
	------		UNDEFINED
	GROUP		every segment explicitly defined or SEG:extrnname
	RECORD		1. "SIZE"=# of bytes, either 1 or 2 2. "WIDTH"=# of bits, 1-16
	RFIELD	Shift count (2 digits)	1. record name 2. "WIDTH"=n or UNDEFINED
EQUATE(You can EQUate to any of the above or to any address expression)			
	V BYTE V WORD V DWORD	offset	1. segment 2. override attribute, "xx:", where xx is the segment register 3. indexing attribute "[]": [DI],[BP], [BX],[SI],[BX + SI],[BP + SI], [BX + DI],[BP + SI]with or without [...]

The Errorprint File

If you selected ERRORPRINT as a control with assembler invocation, then all source lines containing errors, and the error messages plus an error summary are sent to a file or a device (whichever you specified).

This is how it appears

```
ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE BAD
OBJECT MODULE PLACED IN BAD.OBJ
ASSEMBLER INVOKED BY: :F1:ASM86 BAD.SRC ERRORPRINT(:F1:BAD.ERR)

LOC  OBJ             LINE  SOURCE

0000 9090909090        1           MOV   AL,100H
*** ERROR #74, LINE #1, OPERANDS DO NOT MATCH THIS INSTRUCTION

ASSEMBLY COMPLETE, 1 ERROR FOUND
```

The format of the ERRORPRINT information is identical to that of the list file.

If you selected the console device as output for ERRORPRINT the format is similar to the list file format with the exception of all the header information. Here is how it looks on the console device.

```
0000 9090909090        1           MOV   AL,100H
*** ERROR #74, LINE #1, OPERANDS DO NOT MATCH THIS INSTRUCTION

ASSEMBLY COMPLETE, 1 ERROR FOUND
```

If you selected the console device for output and no errors were detected this is the message that displays

```
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPENDIX A
ERROR MESSAGES AND RECOVERY

In keeping with the high-level nature of the MCS-86 assembly language, ASM86 features an advanced error-reporting mechanism. Over 100 English-language messages are provided. Some messages pinpoint the symbol, character, or token at which the error was detected. Error messages are inserted into the listing after the line on which they were detected. They are of the following format:

***ERROR #m, LINE #n, message

where m is the error number, n is the number of the line on which the error occurred, and "message" is the English message corresponding to the error number. If the error is detected in pass 2, the clause "(PASS 2) " precedes the message. Errors numbered less than 800 are ordinary, non-fatal errors. Assembly of the error line can usually be regarded as suspect; but subsequent lines can be assembled correctly. If an error occurs within a codemacro definition or a record definition, the definition does not take place.

Errors numbered in the 800's are assembler errors. They should be reported to Intel if they occur.

Errors numbered in the 900's are fatal errors. They are marked by the line "*** FATAL ERROR ***" preceding the message line. Assembly of the source code is halted. The remainder of the program is scanned and listed, but not acted upon.

Here is a list of the error messages provided by ASM86, ordered by error number.

*** ERROR #1 SYNTAX ERROR AT OR BEFORE "token"

ASM86 contains an internally-encoded grammar of the 8086 assembly language, and requires your program to conform to that grammar. Many times the syntax error will be at the token given in the error message; e.g.,

```
ASSUME CS
```

gives a syntax error at or before <CR>, meaning the line is missing things at the end—in this case, a colon followed by a segment name. More often, however, the assembler will not detect the error until one or more tokens later; e.g.,

```
AAA DB 0
```

gives a syntax error at "DB". The error is that AAA is already defined as an instruction (ASCII adjust for addition). The assembler interprets the line as an AAA instruction with "DB 0" as the operand field. Since the keyword "DB" is not a legal parameter, the "DB" is flagged, even though "AAA" is the user's mistake.

ASM86 treats codemacro, register, and record names as unique syntactic entities; thus, when you use these kinds of names improperly you will often get a syntax error. For example,

```
SS EQU 7
```

is a syntax error since SS is a register name and thus is syntactically distinct from an undefined symbol.

Some grammatic constructs are larger than single lines; i.e., SEGMENT-ENDS pairs, PROC-ENDP pairs, and CODEMACRO-ENDM pairs. You can thus get syntax errors for lines which by themselves are syntactically correct, but are misplaced within the program. E.g.,

```
FOO ENDS        ; with no corresponding SEGMENT statement
BAZ ENDP        ; with no corresponding PROC statement
DATA SEGMENT    ; within a codemacro
```

Note that you will get a syntax error at an END statement if you have SEGMENT or PROC statements without corresponding ENDS or ENDP statements.

ASM86 will usually discard the rest of the line when it finds a syntax error. If the error occurs within a codemacro definition, the assembler exits definition mode. This will cause your ENDM statement to produce another syntax error, which will go away when you fix the first error.

*** ERROR #2 OPERANDS DO NOT MATCH THIS INSTRUCTION

This message will occur quite often for those who are not accustomed to symbol typing in the 8086 assembly language. It usually indicates that the type of one of the operands is inappropriate for the instruction.

For example, the following sequence will generate this error:

```
BAZ DW 0
MOV BL, BAZ
```

Since BAZ is a word variable, it cannot be moved into the byte register BL. You can correct this error in several ways, depending on what your motivations were. You could change BAZ DW 0 to BAZ DB 0. You could change BL to BX. Or you could override the type of BAZ:

```
MOV BL, BYTE PTR BAZ
```

Another example: if FOO is a procedure name, the instruction

```
MOV BX,FOO
```

generates an error, since there is no MOV instruction in the built-in codemacros whose second operand is a label type. The correct version is

```
MOV BX, OFFSET FOO
```

since OFFSET FOO is typed as a number.

To become proficient at avoiding this error, and at understanding this error when it occurs, you should master the distinction between the types "variable", "label", and "number". You should then learn how to read the built-in codemacros for the instruction set. Finally, when you understand the algorithm used for matching instructions to the codemacro definitions, you will be able to predict the precise code generated for each instruction.

In some cases, this error reveals instructions not supported by the 8086 hardware. For example,

```
FOO SEGMENT
MOV ES, FOO
```

gives an error because there is no hardware instruction which moves the immediate number FOO into a segment register. You must re-code:

```
MOV AX, FOO
MOV ES, AX
```

*** ERROR #3 INSTRUCTION SIZE BIGGER THAN PASS 1 ESTIMATE

This error occurs when the instruction contains a forward reference, and the assembler guesses too optimistically about how much code the forward reference will cause the instruction to generate. There are several situations in which this happens:

a. The forward reference is a variable which requires a segment override prefix. For forward references, you must explicitly code the override:

   ```
   MOV CX, ES:FWD_REF
   ```

   Otherwise, the assembler will guess that it is not needed.

b. The forward reference is a FAR label. You must explicitly provide the type in this case:

   ```
   JUMP FAR PTR FWD_LABEL
   ```

   Otherwise, the assembler will guess NEAR.

c. You have promised SHORT, or you have used an instruction which takes only SHORT displacements. You must change your code not to use a SHORT jump.

   To minimize the chance of this error, you should avoid forward references as much as possible. Declare your variables and externals at the top of your modules; try to arrange your program so that FAR labels come first.

*** ERROR #4 INSUFFICIENT TYPE INFORMATION TO DETERMINE CORRECT INSTRUCTION

This error occurs when one of the operands to an instruction is a register expression which does not have a BYTE or WORD attribute attached to it. If one of the other operands can identify the type, then no error is issued; e.g.,

```
MOV AX, [BX]
MOV [BX], 0FFFEH
MOV BL, [DI + 500]
```

are all correct because the AX and the 0FFFEH indicate that WORD PTR [BX] is intended, and the BL indicates that BYTE PTR [DI] is intended. However,

```
INC [BX]
MOV [BX],0
```

are both flagged. The 0 does not commit [BX] to being a BYTE or a WORD memory location. You must specify BYTE PTR [BX] or WORD PTR [BX] for both instructions.

*** ERROR #5 OPERAND NOT REACHABLE FROM SEGMENT REGISTERS

This error occurs when you do not use the ASSUME statement correctly. Every time you reference a variable, the segment in which that variable occurs must be ASSUMEd to be reachable from one of the segment registers. For example, the program

```
FOO SEGMENT
BAZ DW 0
MOV AX, BAZ
FOO ENDS
END
```

will produce this error, since BAZ cannot be reached from any of the registers. The line

```
ASSUME DS:FOO
```

at the top of the program will eliminate the error.

For most programs, a single ASSUME statement at the top of the program for each of the four segment registers CS,DS,ES, and SS will suffice.

If you want more than one segment to be reachable from the same segment register at the same time, you must GROUP the segments together, and ASSUME the group to be reachable.

*** ERROR #6 CANNOT JUMP NEAR TO A LABEL WITH A DIFFERENT CS-ASSUME

This error detects the following inconsistency in your program: You demand a NEAR jump to another section of code. NEAR jumps do not change the CS register. Yet the other piece of code is expecting the CS register to have a different value than the code from which you are jumping. You must either make a FAR jump, or change your CS-assumes so they are consistent.

*** ERROR #7 NO CS-ASSUME IN EFFECT—NEAR LABEL CANNOT BE DEFINED

The assembler must store the CS-assume associated with each label It needs this in order to instruct the LINK program to generate the correct displacement for NEAR jumps between different segments of the same group. For most programs, a single ASSUME statement at the top of the code will suffice.

*** ERROR #8 NO CS-ASSUME IN EFFECT—NEAR JUMP CANNOT BE GENERATED

This is a special case of error 6: you are missing a CS-assume.

*** ERROR #9 DEFAULT SEGMENT CANNOT BE OVERRIDDEN

This is a signal that you have attempted to violate a hardware limitation in the string imperatives which involve the DI register. The hardware does not allow for any override of the default ES register; thus the assembler requires the operand to the instruction to be reachable from the ES register. The facility is implemented via the NOSEGFIX directive included in the appropriate codemacros.

*** ERROR #10 LABEL CANNOT BE USED AS A VARIABLE
(NO COLON ALLOWED)

This error occurs when you put a colon on the label to a storage initialization line; e.g.,

```
FOO: DB 3
```

The assembler assumes that you probably want FOO to be a variable in this context, and requires you to make it so by removing the colon. If you understand the difference between a variable and a label, and you still want FOO to be a label, you could make it so by placing it by itself on the line above the DB.

*** ERROR #11 ILLEGAL LABEL TO THIS DIRECTIVE
(NO COLON ALLOWED)

This error is reported when a label with a colon appears on a GROUP, PROC, RECORD, or SEGMENT directive. These directives call for a label without a colon.

*** ERROR #12 THIS DIRECTIVE REQUIRES A LABEL (WITHOUT A COLON)

This error is reported for a missing label to a GROUP, PROC, RECORD, or SEGMENT declarative.

*** ERROR #13 THIS DIRECTIVE DOES NOT ACCEPT A LABEL TO ITS LEFT

This error is called for lines on which no label is allowed: ASSUME, CODEMACRO, EXTRN, NAME, ORG, PURGE, and PUBLIC.

*** ERROR #14 LABEL IS NOT REACHABLE
FROM CS—WILL NOT BE DEFINED

This happens when you have no ASSUME for CS, or when your CS-ASSUME is for a segment other than the one you are assembling. For example, if FOO is a segment,

```
ASSUME CS:FOO
BAZ SEGMENT
GORN PROC
```

is illegal—the assembler does not know what offset to generate for the label GORN, since GORN's segment BAZ is not ASSUMEd to be in the CS register. To correct this error, you can either provide an ASSUME CS:BAZ, or group FOO and BAZ together, and ASSUME that CS contains the group, as follows:

```
FOOBAZ GROUP FOO, BAZ
ASSUME CS:FOOBAZ
BAZ SEGMENT
GORN PROC
```

*** ERROR #15 ALREADY DEFINED SYMBOL,
THIS DEFINITION IGNORED FOR "symbol"

This error is given when a symbol has an illegal multiple definition. To avoid confusion, we suggest that you usually correct this error by using a different name for one of the symbols, instead of using PURGE.

*** ERROR #16 ALREADY EQUATED SYMBOL,
THIS DEFINITION IGNORED FOR "symbol"

This is identical to case 15, except that the quoted name has appeared EQUated to a forward reference name which has not yet been resolved.

*** ERROR #17 ARITHMETIC OVERFLOW IN EXPRESSION OR LOCATION COUNTER

This error is reported whenever a 17-bit calculation takes place whose answer is not in the bounds −65535 to 65535. Notable particular instances of this include:

a. User expressions with large answers or intermediate values
b. Division by zero
c. Oversize constants
d. Overflow of the location counter

*** ERROR #18 ILLEGAL CHARACTER IN NUMERIC CONSTANT

Numeric constants begin with decimal digits, and are delimited by the first non-token character (not alpha, numeric, '/', '@', or '_'). The set of legal characters for a constant is determined by the base:

a. Base 2: 0,1, and the concluding 'B'.
b. Base 8: 0-7, and the concluding 'O' or 'Q'.
c. Base 10: 0-9, and the optional concluding 'D'.
d. Base 16: 0-9, A-F, and the concluding 'H'.

*** ERROR #19 ABSOLUTE,
NON-FORWARD-REFERENCE NUMBER REQUIRED

This error is reported in cases where the absolute number expected cannot be completely computed at pass 1 assembly time. Note that this excludes relocatable numbers. The situations where this is required include:

a. A SEGMENT directive with an AT.
b. A DUP count.
c. Widths and defaults in a RECORD definition.
d. Range specifiers in a CODEMACRO definition.
e. Initialization values in a CODEMACRO definition.

*** ERROR #20 ADDRESS EXPRESSION REQUIRED AS OPERAND TO THIS OPERATOR

Some expression operators don't make any sense if their operands are not address expressions (see the MCS-86 Assembly Language Reference Manual for a discussion of address expressions). These operators include segment override, OFFSET, bracket combination, subtraction with non-absolute minuend, SEG, TYPE, LENGTH, and SIZE of a non-record-name.

*** ERROR #21 ILLEGAL OPERANDS TO ADDITION OR COMBINATION OPERATION

One of the operands to an addition or combination operation has to be either an absolute number or an absolute register expression. Note that this error may occur if the operation is subtraction; since if the right-hand operator is an absolute number it is negated and then added.

*** ERROR #22 NEGATIVE NUMBER NOT ALLOWED IN THIS CONTEXT

Certain contexts disallow negative numbers. They include:

a. SEGMENT declaratives with AT
b. DUP counts

*** ERRORS #23,#24 ILLEGAL USE OF REGISTER NAME OUTSIDE OF BRACKETS

Inside of square brackets, a register can undergo arithmetic; the operations are performed on the memory address represented by the bracketed expression. Outside of the brackets, the arithmetic makes no sense, and is flagged. The example is:

```
JMP BX + 3
```

is illegal; write JMP [BX — 3] instead.

*** ERROR #25 SHORT JUMP DISPLACEMENT DOES NOT FIT IN A BYTE

This error occurs in situations where a codemacro is matched, but the parameter fails to fit when the RelB directive is encountered. Note that this can never happen in the built-in instruction set, since all RelB directives are for parameters specified Cb; so the codemacro match would never have been made.

*** ERRORS #26,#27 TWO BASE OR TWO INDEX REGISTERS BEING COMBINED

The hardware does not support the following sorts of instructions:

```
MOV AX, [BX + BP]
MOV AX, FOO[SI][DI]
MOV AX, [BX + BX]
```

i.e., at most one base register and at most one indexing register can appear in an indexing expression.

*** ERRORS #28,#29,#30 BAD OPERANDS FOR RELATIONAL OR SUBTRACTION OPERATION

Subtraction and relational operations are legal only if the right side is an absolute number; or if both sides match in all relocation types and attributes. If neither of these conditions hold, this error is reported.

*** ERROR #31 ILLEGAL CHARACTER: "char"

The quoted character is printable, but it has no function in the 8086 assembly language.

*** ERROR #32 INSTRUCTION OPERAND DOES NOT HAVE A LEGAL TYPE

The only case is which this error should occur is if you use a record or a record field name by itself as an operand to an instruction.

*** ERROR #33 MORE ERRORS DETECTED, NOT REPORTED

After the ninth error on a given source line, this message is given and no more errors are reported for the line. Normal reporting resumes on the next source line.

*** ERROR #34 FORWARD-REFERENCE EQUATE CHAIN MAY NOT RESOLVE TO A REGISTER OR CODEMACRO

Forward references to codemacros and registers are illegal. This is one situation in which the error is reported.

*** ERROR #35 CANNOT EQUATE TO EXPRESSIONS INVOLVING FORWARD REFERENCES

You may equate to simple forward-reference names, or you may equate to expressions without forward references, but you cannot do both. E.g.,

```
FOO EQU BAZ + 1
BAZ EQU 5
```

is not allowed.

*** ERROR #37 UNDEFINED INSTRUCTION OR ILLEGAL VARIABLE DEFINITION

This error is reported when you give an undefined label, without a colon, at the beginning of a line, in a context where it cannot be taken as a variable definition. Usually this is just a misspelled instruction.

*** ERROR #38 UNDEFINED SYMBOL, ZERO USED

This error is reported when an undefined symbol occurs in an expression context. The absolute number zero which is used in its place may cause other errors to occur.

*** ERROR #39 VALUE WILL NOT FIT IN A BYTE

This error is issued for DB lines in which the absolute operand is not in the range −256 to 255.

*** ERROR #40 CANNOT HAVE A VARIABLE OR A LABEL IN A DB

This is another case where a symbol is of the wrong type for the context. Although conversion to the offset number automatically occurs for DW, it does not occur for DB—you must explicitly provide the OFFSET operator, and you must be sure that the resulting number is absolute and small enough.

*** ERROR #41 RELOCATABLE VALUE DOES NOT FIT IN ONE BYTE

The only relocatable numbers acceptable as operands to DB (alone or within codemacros) are numbers to which HIGH or LOW have been applied.

*** ERROR #42 STORAGE INITIALIZATION EXPRESSION IS OF THE WRONG TYPE

The only kinds of expressions allowed in initialization lists (i.e., as operands to DB, DW, DD) are variables, labels, strings, formals, and numbers. Other types will produce this error.

*** ERROR #43 STRING TERMINATED BY END-OF-LINE

All strings must be completely contained on one line. The ampersand continuation feature does not work in the middle of a string. The assembler will treat the string as if you had inserted a quote mark as the last character of your line.

*** ERROR #44 STRING LONGER THAN 2 CHARACTERS ALLOWED ONLY IN DB

Outside of the DB context, all strings are treated as absolute numbers; hence, strings of 3 or more characters are overflow quantities. You probably should be using DB.

*** ERROR #45 STRING CONSTANT CANNOT EXCEED 40 CHARACTERS

The assembler issues this message and uses the first 40 characters if the string is too long.

*** ERROR #46 DUP NESTING ALLOWED ONLY TO A DEPTH OF 8

No reasonable program will ever run into this limitation. The kind of line that would cause it is:

```
DW 2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(2 DUP(3 DUP(1234H)))))))))
```

*** ERROR #47 PARENTHESIS NESTING ALLOWED
ONLY TO A DEPTH OF 8

An example of this error would be:

```
DW 1 + (1 + (1 + (1 + (1 + (1 + (1 + (1 + (1 + (1 + 2))))))))
```

It is not likely that you will run into this limitation in any practical application.

*** ERROR #48 ABSOLUTE OPERAND REQUIRED IN THIS EXPRESSION

Most expression operators require their operands to be absolute numbers. These operators include unary minus, divide, multiply, AND, MOD, NEG, OR, SHL, SHR, and XOR.

*** ERROR #49 CANNOT TAKE HIGH OR LOW OF A PARAGRAPH NUMBER

The only kind of relocatable number which can undergo HIGH or LOW is the offset. The address of a segment does not accept HIGH or LOW. We recommend that HIGH and LOW be used only in programs translated from 8080 programs; and segment addresses do not occur in such programs in this context.

*** ERROR #50 OPERAND TO HIGH OR LOW MUST BE A VARIABLE,
LABEL, OR NUMBER

Other types of operands (e.g., segment names or record names) are disallowed.

*** ERROR #51 ILLEGAL USE OF A GROUP AS A SEGMENT OVERRIDE

This error should occur only if you attempt to provide a segment override which is a group name to an expression which already has a segment override which is a group name. For example,

```
FOO GROUP A,B,C
BAZ GROUP D,E,F
GORN DW 0
DW FOO: BAZ: GORN
```

*** ERROR #52 SEGMENT OVERRIDE MAY BE APPLIED
ONLY TO AN ADDRESS EXPRESSION

For example, the expression DS:0 is illegal. You must convert the number 0 into an address expression. This can be accomplished via the PTR operator, e.g., DS: BYTE PTR 0.

*** ERROR #53 LEFT OPERAND TO SEGMENT OVERRIDE
HAS AN ILLEGAL TYPE

The left operand to the segment override (colon) operator must be either a segment register, a segment name, a group name, or SEG of a variable or label.

*** ERROR #54 LABEL CANNOT HAVE INDEXING REGISTERS

If the left operand to PTR is NEAR or FAR, then the right operand may not have any indexing registers. The 8086 does not support labels with indexing registers (e.g., NEAR PTR [BX]).

*** ERROR #55 INVALID EXPRESSION IN SQUARE BRACKETS

The only kind of expression allowed in square brackets is an expression involving registers and/or numbers. Address expressions and other constructs (e.g., record names) are not allowed.

*** ERROR #56 VARIABLE AND SUBSCRIPT
MAY NOT BOTH BE RELOCATABLE

Example: if FOO and BAZ are both relocatable numbers, the expressions [BX + FOO] and BAZ [BX] are both legal; the expression BAZ [BX + FOO] is not, since it requires the addition of two relocatable quantities.

*** ERROR #57 OPERAND OF WIDTH MUST BE A RECORD
OR RECORD FIELD NAME

WIDTH of anything else has no meaning.

*** ERROR #58 OPERAND OF MASK MUST BE A RECORD FIELD NAME

MASK of anything else has no meaning.

*** ERROR #59 OPERAND TO OFFSET MUST BE A VARIABLE OR LABEL

OFFSET is an operator provided to allow you to convert variables or labels to numbers. If you get this error message, you probably already have a number.

*** ERROR #60 OPERAND TO LENGTH CANNOT BE A LABEL

LENGTH is intended to give the number of units initialized at a variable definition. Since labels are associated with instructions and not with storage initializations, it makes no sense to speak of the LENGTH of a label.

*** ERROR #61 OPERAND TO SIZE CANNOT BE A LABEL

SIZE is intended to give the number of bytes initialized at a variable definition. Since labels are associated with instructions and not with storage initializations, it makes no sense to speak of the SIZE of a label.

*** ERROR #62 LEFT OPERAND TO PTR CANNOT BE ZERO

Besides the usual keywords BYTE, WORD, DWORD, NEAR, and FAR, you can also give a numeric value as a left operand to PTR; e.g., 3 PTR 0. This creates a variable whose constituent unit size (i.e., TYPE) is the left operand. However, 0 PTR 4 is illegal, since 0 as a constituent unit size makes no sense.

*** ERROR #63 LEFT OPERAND TO PTR IS OF INVALID TYPE

The only valid left operands to PTR are absolute numbers and the keywords BYTE, WORD, DWORD, NEAR and FAR (which are synonyms for 1, 2, 4, −1, and −2, respectively).

*** ERROR #64 ILLEGAL NEGATIVE TYPE TO PTR, NEAR USED INSTEAD

The only negative numbers allowed as the left operand to PTR are −1 and −2, which are synonyms for NEAR and FAR. Other negative numbers are converted to NEAR, and this message is issued.

*** ERROR #65 INVALID RIGHT OPERAND TO PTR

Only variables, labels, numbers, and index-register expressions may appear to the right of PTR.

*** ERROR #66 CANNOT MAKE A SEGMENT REGISTER OVERRIDDEN VARIABLE INTO A LABEL

This error occurs when you have a variable with a segment register override as the right operand to PTR, and NEAR or FAR as the left operand. The resulting combination is illegal, since labels cannot be overridden. For example,

```
FOO DW 0
JMP NEAR PTR (ES:FOO)
```

is illegal: jumps always use the CS register, so the ES override cannot be honored.

*** ERROR #67 CANNOT OVERRIDE A LABEL WITH A SEGMENT REGISTER

This, like error #66, is an attempt to create a label with a segment register override. In this case, the attempt is made via the override operator; e.g.,

```
LABL: JMP DS:LABL
```

*** ERROR #68 ILLEGAL OPERAND TO SEG OPERATOR

The operand to SEG as it appears in a GROUP or ASSUME statement must be a variable or a label; i.e., it must have a segment associated with it.

*** ERROR #69 OPERAND TO SEG HAS NO SEGMENT

The operand to SEG as it appears in an expression must be a variable or a label—if not, it has no segment associated with it; and SEG therefore has no meaning.

*** ERROR #70 RELOCATION OF LABEL TOO COMPLICATED

In practical programs, you should never see this error. An example of what it takes to produce it is:

```
JMP GROUPNAME:SEGNAME:FOO
```

where FOO is a label in a segment whose offsets require relocation.

*** ERROR #71 SOURCE LINE CANNOT EXCEED 128 CHARACTERS

The only effect of this mistake is that the excess characters are not listed—the line is otherwise processed correctly.

*** ERROR #72 ATTEMPT TO SHIFT A RELOCATABLE VALUE

This error results when a relocatable value is passed as an operand to an instruction whose codemacro shifts the operand before outputting it. It does not make sense to shift a relocatable value.

*** ERROR #73 CANNOT PUT A RELOCATABLE VALUE INTO A RECORD OR MODRM FIELD

This error results when a relocatable value is passed as an operand to an instruction whose codemacro squeezes the operand into a record field or a MODRM field. It does not make sense to extract fields from relocatable values.

*** ERROR #74 STARTING ADDRESS MUST BE A LABEL

The starting address of the program, given as an optional operand to the END statement, is the point to which the loader of the program will jump. As such, it must be a label (and not, for example, a variable or a number).

*** ERROR #75 UNDEFINED RIGHT SIDE OF EQU

The left side will in this case remain undefined.

*** ERROR #76 RIGHT SIDE OF EQU IS OF ILLEGAL TYPE

Only simple names and expressions are allowed on the right side of EQU. An example of a wrong type is: FOO EQU 'STRING'

*** ERROR #77 CANNOT EQU SYMBOL TO ITSELF

The example FOO EQU FOO is illegal.

*** ERROR #78 CIRCULAR CHAIN OF EQUATES

An example is:

```
FOO EQU BAZ
BAZ EQU FOO
```

*** ERROR #79 LEFT SIDE OF EQU ALREADY DEFINED, THIS EQU IGNORED

Only previously undefined or purged names can appear to the left of EQU.

*** ERROR #80 SYMBOL NOT IN USER SYMBOL TABLE, CANNOT BE PURGED

The user symbol table contains not only user symbols, but also the instruction set codemacros, the registers, and the built-in segment ??SEG. Any of these names can be purged. Assembler keywords (e.g., DB, EXTRN, BYTE, PUBLIC, AT, SEG, RELW, DUP, etc.) appear in another table and cannot be purged. If you get this message, your symbol is either an assembler keyword, was never defined, or was already purged.

*** ERROR #81 OPERAND TO ORG NOT IN THIS SEGMENT

The operand to ORG can be either an absolute number or a relocatable number. If it is relocatable, it must be offset-relocatable from the segment currently being assembled. Such a number is usually had by applying OFFSET to a variable or label in the current segment; for example,

```
ORG OFFSET $ + 2
```

*** ERROR #82 ILLEGAL FORWARD REFERENCE OF A REGISTER

The only time this can happen is if you use EQU to give an alternate name to a register, but use the alternate name somewhere above the EQU statement. This is not allowed. You should always put EQUs to registers at the top of your program; in fact, we recommend that you put all your EQUs at the top of your program.

*** ERROR #83 ALIGN-TYPE DOES NOT MATCH ORIGINAL SEGMENT DEFINITION

If you have more than one SEGMENT-ENDS pair for the same segment in your program, they must have the same align-type. For example, you cannot specify one to be BYTE and the other to be PARA. Note that if you leave the align-type off the first SEGMENT declaration, that segment has align-type PARA. Therefore, all subsequent declarations of that segment must have either no align-type or align-type PARA. It is always acceptable to leave the align-type blank for subsequent SEGMENT declaratives—the align-type given in the first declarative is used.

*** ERROR #84 COMBINE-TYPE DOES NOT MATCH ORIGINAL SEGMENT DEFINITION

If you have more than one SEGMENT-ENDS pair for the same segment in your program, they must have the same combine-type. For example, you cannot specify the first one to be no combine-type (private), and a subsequent one to be PUBLIC. It is always acceptable to leave the combine-type blank for subsequent SEGMENT declaratives—the combine-type given in the first declarative is used.

*** ERROR #85 CLASS DOES NOT MATCH ORIGINAL SEGMENT DEFINITION

If you have more than one SEGMENT-ENDS pair for the same segment in your program, they cannot have differing classes. For example,

```
FOO SEGMENT 'CODE'
ENDS
FOO SEGMENT 'DATA'
ENDS
```

is illegal. Note that it is always acceptable to give the class for the first SEGMENT declarative for a segment, and then leave the CLASS blank for all subsequent declaratives.

*** ERROR #86 MISMATCHED LABEL ON ENDS OR ENDP

ENDS and ENDP require a label which matches the corresponding SEGMENT and PROC declaratives. If this error occurs, one of several things could be wrong: You could have a typographical error. You could have a missing ENDS or ENDP for a nested SEGMENT or PROC. You could have an error in the corresponding SEGMENT or PROC line; in which case this error will go away when the other is fixed.

*** ERROR #87 CANNOT HAVE MORE THAN ONE NAME DECLARATIVE

The first NAME declarative is honored and this one is ignored.

*** ERROR #88 TEXT FOUND BEYOND END STATEMENT—IGNORED

This is a warning—there are no ill effects. The extra text appears in the listing but is not assembled.

*** ERROR #89 PREMATURE END OF FILE (NO END STATEMENT)

There are no ill effects from omitting the END statement, other than this message. Note that if your program is missing an ENDM, ENDS, or ENDP statement, the END statement is syntactically invalid and is thus not recognized. This error message will follow the syntax error message.

*** ERROR #90 RECORD FIELD WIDTH MUST BE BETWEEN 1 AND 16 BITS

Zero-width record fields are disallowed. Widths greater than 16 make no sense, since the containing record cannot exceed 16 bits.

*** ERROR #91 RECORD WIDTH MAY NOT EXCEED 16 BITS

The record is not defined when this happens.

*** ERROR #92 DEFAULT VALUE DOES NOT FIT INTO RECORD FIELD

The default value for the record field is too large: the number of bits needed to represent the number is greater than the width of the field.

*** ERROR #93 LEFT OPERAND TO DOT OPERATOR
MUST BE A FORMAL PARAMETER

*** ERROR #94 RIGHT OPERAND TO DOT OPERATOR
MUST BE A RECORD FIELD

The dot operator is a special operator allowed in only one context: in codemacros, with a formal to the left and a record field to the right. Any other usage is an error.

*** ERROR #95 RECORD INITIALIZATION ILLEGAL OUTSIDE OF A
CODEMACRO

This error occurs when the first name on a line is a record name. You could be trying an initialization, as the message indicates or you could be trying to redefine the name, not realizing that it is a record name.

*** ERROR #96 CODEMACRO NAME ALREADY DEFINED AS SOMETHING OTHER THAN A CODEMACRO

It is legal to have multiple definitions of a codemacro. In that case, however, all definitions of the symbol must be codemacro definitions. If the symbol has been defined as anything else, it cannot be redefined as a codemacro, unless it is first purged.

*** ERROR #97 TWO FORMALS WITH THE SAME NAME

Within a given codemacro definition, all formals must have a different name.

*** ERROR #98 CANNOT HAVE MORE THAN 7 FORMALS TO A CODEMACRO

This limitation is imposed by the internal codemacro coding formats.

*** ERROR #99 ILLEGAL SPECIFIER LETTER TO A CODEMACRO FORMAL

The only specifier letters allowed are A, C, D, E, M, R, S, and X.

*** ERROR #100 ILLEGAL MODIFIER LETTER TO A CODEMACRO FORMAL

The only modifier letters allowed are B, D, W, and nothing.

*** ERROR #101 ILLEGAL EXTRA CHARACTERS AFTER SPECIFIER AND MODIFIER

You have either made a typographical error, or have mistaken the syntax of CODEMACRO lines.

*** ERROR #102 ONLY A,D,R,S SPECIFIERS CAN TAKE A RANGE

Range checking for codemacro matching is done only for parameters which are numbers or registers.

*** ERROR #103 FORMAL PARAMETER EXPECTED BUT NOT SEEN

In certain contexts in codemacros (i.e., RELB, RELW, SEGFIX, NOSEGFIX, and MODRM), the only construct allowed is a formal parameter. If it is not seen, this error is given.

*** ERROR #104 UNDEFINED OR FORWARD REFERENCE ILLEGAL IN CODEMACRO

All numbers provided in a codemacro definition must be determined in pass 1.

*** ERROR #105 ILLEGAL STORAGE INITIALIZATION CONSTRUCT FOR A CODEMACRO

This error occurs when an operand to a storage initialization (DB, DW, DD, or record initialization) is of illegal type; e.g. a record name by itself as an operand would produce this error.

*** ERROR #106 INSTRUCTIONS NOT ALLOWED IN CODEMACROS, USE INITIALIZATIONS INSTEAD

This error results when you place an instruction (a codemacro call) within a codemacro definition. For example,

```
CODEMACRO NOP
XCHG AX,AX
ENDM
```

is an error. You must hand-expand the codemacro with the appropriate storage initialization:

```
CODEMACRO NOP
DB 90H
ENDM
```

*** ERROR #107 NESTED ANGLE BRACKETS NOT ALLOWED

For example, the construct <<0,1>,2> is flagged by this message.

*** ERROR #108 A NULL ENTRY IS LEGAL ONLY WITHIN ANGLE BRACKETS

The line RECNAME <0,,1> is legal within a codemacro—the default value is used for the second field. However, outside of a record initialization context: DB 0,,1 the null entry makes no sense.

*** ERROR #109 DEFINITION TOO BIG FOR INTERNAL BUFFER

The internal storage limit for groups, records, and codemacros is 128 bytes. For groups, this is a limit of 40 segments. For records, the limit cannot be reached (you will run into the width limit before this one). The limit for codemacros is not easy to define; a rough guess is that a codemacro which generates 60 bytes of object code is near the limit.

*** ERROR #110 RECORD INITIALIZATION TOO COMPLICATED FOR CODEMACRO ENCODING

The internal codemacro storage formats disallow a record initialization to produce more than 15 bytes of internal code. What this means externally is complicated to describe; but if none of your records has more than 7 fields, you should never run into this limit.

*** ERROR #111 MISMATCHED LABEL ON ENDM

The label on the ENDM directive is optional; if it is given, it must match the corresponding CODEMACRO name.

*** ERROR #112 TYPE IS ILLEGAL FOR PUBLIC SYMBOL "symbol"

Only variables, labels, and numbers may be declared public. No subscripting or overrides are allowed.

*** ERROR #113 NO DEFINITION FOR PUBLIC SYMBOL "symbol"

A public symbol must be defined within the program.

*** ERROR #114 CANNOT ASSUME AN UNDEFINED SEGMENT

If a symbol is ASSUMEd into a segment register and is a forward reference, the assembler always guesses that it is a segment. If the symbol is never defined, it is an undefined segment. Although this usage of an undefined segment is illegal for ASSUMEs, it is legal for group definitions.

*** ERROR #115 DUP COUNT MUST BE GREATER THAN 0, 1 USED

The repetition count of a DUP must be greater than 0. It is not unusual for this error to immediately follow error 22.

*** ERROR #800 UNRECOGNIZED ERROR #MESSAGE NUMBER
*** ERROR #801 SOURCE FILE READING UNSYNCHRONIZED
*** ERROR #802 INTERMEDIATE FILE READING UNSYNCHRONIZED
*** ERROR #803 BAD OPERAND STACK RECORD
*** ERROR #804 BAD OPERAND STACK READ REQUEST
*** ERROR #805 BAD OPERAND STACK POP REQUEST
*** ERROR #806 PARSE STACK UNDERFLOW
*** ERROR #807 AUXILIARY STACK UNDERFLOW
*** ERROR #808 BAD AUXILIARY STACK READ REQUEST
*** ERROR #809 BAD OPERAND STACK TYPE IN EXPRESSION
*** ERROR #810 BAD STORAGE INITIALIZATION RECORD
*** ERRORS #812,#813 INSTRUCTION OPERAND HAS IMPOSSIBLE TYPE

Error messages in the 800's should never occur. If you get one of these error messages, please notify Intel Corporation via the Software Problem Report included with this manual.

*** ERROR #900 USER SYMBOL TABLE SPACE EXHAUSTED

You must either eliminate some symbols from your program, or break your program into smaller modules.

*** ERROR #901 PARSE STACK OVERFLOW

This error will be given only for grammatical entities far beyond the complication seen in normal programs.

*** ERROR #902 OVERFLOW IN OPERAND STACK—TOO MANY ELEMENTS

This error typically occurs when a list of storage initialization elements is too long—about 20 elements, depending on the complication of the last elements. You can correct this by breaking your initialization up into several lines.

*** ERROR #903 OVERFLOW IN OPERAND STACK—ELEMENTS TOO COMPLICATED

This error is similar to error 902. You should break your list of elements into several lines.

*** ERROR #904 AUXILIARY STACK OVERFLOW

This error indicates that one of ASM86's minor stacks has overflowed. This can come about through excessively complicated storage initialization operands; or by excessively deep nesting of SEGMENTs and PROCs.

*** ERROR #905 INTERMEDIATE FILE BUFFER OVERFLOW

This error indicates that a single source line has generated an excessive amount of information for pass 2 processing. In practical programs, the limit should be reached only for lines with a gigantic number of errors—correcting the other errors should make this one go away.

APPENDIX B
LINKING MCS-86 ASSEMBLY LANGUAGE AND PL/M-86 PROGRAMS

This appendix is directed to the person who is already familiar with PL/M-86 and the documents related to it. In particular, Chapter 9 in the *PL/M-86 Compiler Operator's Manual* provides linking information. If you require a more broad background of information, turn to the preface for a complete list of those documents and their order numbers.

The purpose of this appendix is to describe and show how modules coded in ASM86 can communicate with modules in PL/M-86. This means how data may be passed between such modules to provide parameters for processing and to return the results of that processing.

The conventions for passing data back and forth are determined by the PL/M-86 language compiler. These conventions, explained in the next pages, include the stack, the BP register, and the general purpose registers used in specific ways.

PL/M-86 generates object code for three distinct environments called SMALL, MEDIUM, and LARGE models of computation; this fact places additional constraints on the assembly language programmer. The constraints that are unique to these three environments are described AFTER the conventions which are common to all. The examples at the end of this appendix illustrate all the models of computation.

Conditions and Conventions Common To All Models of Computation

1. The parameters are all passed on the stack.
2. When there are parameters, they must be pushed onto the stack prior to the call instruction, in the left-to-right order named in the PL/M-86 procedure declaration. For example, if the MCS-86 assembly language program is calling a PL/M-86 program as follows:

   ```
   P:  Procedure (Parm1, Parm2, Parm3) PUBLIC;
   ```

   then Parm1 must be pushed first, Parm2 second, Parm3 third. This is the order PL/M-86 expects to find them on the stack, when this procedure is CALLed. This is also the order PL/M-86 supplies them on the stack when it executes a CALL to any procedure.

   Therefore when a PL/M-86 program CALLs an ASM86 procedure, the left-to-right order of the operands (parameters) in the PL/M-86 CALL must correspond to the first-to-last order expected by the ASM86 procedure.

   Word parameters are pushed as words. The convention for passing bytes is to put the byte value in the low byte of the word pushed onto the stack. This is what your ASM86 procedure must expect for byte parameters from PL/M-86 calls, and also what it must supply if it passes bytes to a PL/M-86 procedure.

   When doubleword pointers are passed (MEDIUM or LARGE models only), the segment word is pushed onto the stack first, followed by the offset word.
3. PL/M-86 expects the stack to look the same after a procedure returns as it looked before the parameters were pushed. Therefore an ASM86 program CALLing a PL/M-86 procedure should expect the stack upon return to no longer have the parameters available, because the PL/M-86 procedure adjusted SP. Furthermore, a CALLed ASM86 procedure may return by using the statement RET N, where N is the number of BYTES occupied by the parmeters

passed. This will restore the stack to its condition prior to the CALL, by incrementing SP. Note that in the SMALL case, N is always twice the number of parameters. In the MEDIUM and LARGE cases, however, N must be the sum of four times the number of pointers passed, plus twice the number of non-pointers, because pointers in those environments are 4 bytes instead of 2.

4. PL/M-86 uses the BP register to address the stack. A CALLed procedure in ASM86 must be sure to save this value if BP will be used in the procedure, and to restore that value prior to returning control to the PL/M-86 program that CALLed it.
5. Except for functions (see rule 6), PL/M-86 considers all general purpose registers except SP and BP to be volatile, i.e., their contents need not be saved and restored. Consequently, a CALLed procedure in ASM86 is free to use such registers without considering their prior contents. An ASM86 program CALLing a PL/M-86 procedure cannot expect the contents of these registers to be preserved. Instead, it must save what it needs prior to CALLing the procedure.
6. PL/M-86 expects to receive and provide return values (function results) in certain registers depending on the TYPE of procedure, as follows:

Procedure Type	Result Returned In
BYTE	AL
WORD	AX
INTEGER	AX
POINTER (SMALL)	BX
POINTER (MEDIUM,LARGE)	ES and BX
REAL	top of RMU* stack

*RMU, the real math unit is documented in the PL/M-86 Compiler Operator's Manual

7. In all models, if an ASM86 procedure expects to be called by a PL/M-86 program and the procedure needs to alter any segment registers, except ES, it must save the segment register(s) upon entry and restore them prior to the return. ES is exempt from this rule.

Conditions and Conventions Specific to Each Model of Computation

Small Model

For the SMALL model of computation, PL/M-86 creates two groups named CGROUP, containing the CODE segment, and DGROUP containing the DATA, STACK, CONST, and MEMORY segments. The CS register contains the base address of the CGROUP and the DS and SS registers contain the base address of the DGROUP.

To communicate successfully, your ASM86 procedure must be in a segment named CODE which is PUBLIC and has classname 'CODE',

```
CODE SEGMENT PUBLIC 'CODE'
```

This declaration causes your code to be combined with the output of PL/M-86 such that CALLs and RETurns are NEAR. One consequence of this is the return address occupies only one word on the stack for SMALL. Another reason for this declaration is that, in the SMALL model, PL/M-86 declares all external procedures to be in the CODE segment.

Medium Model and Large Model

In these models, all CALLs and RETurns to external procedures are "FAR", using 2 words for the return address on the stack. Therefore you may name the segment containing your procedure anything you like. PL/M-86 pointer variables are 32-bit quantities under MEDIUM and LARGE. Pointer parameters in these two models are stored on the stack with the segment pushed first. The return address is stored on the stack this same way, segment first, offset last. PL/M-86 procedures in the LARGE models will save the DS register and restore it upon Procedure exit. A PUBLIC PL/M-86 procedure will have type FAR and will do a "long" RETurn, i.e., restoring both the Instruction Pointer and the CS register.

Static Data

Static data means data which is not passed as parameters,but which is around all the time, that is, local or external.

Local Data

Small and Medium Model of Computation. Data declared in the ASM86 module must be in the 'DATA' segment. The data segment must be declared as

```
DATA SEGMENT PUBLIC 'DATA'
```

;local variable declarations

```
DATA ENDS
```

Moreover, PL/M-86 requires that the data segment be contained in the group, "DGROUP". Dgroup should be declared as

```
DGROUP GROUP DATA
```

You need not declare other segments in the group you do not use (e.g., STACK, CONST,MEMORY).

Large Model of Computation. The data segments for PL/M-86 are non- combinable segments, i.e., NOT PUBLIC. Therefore, the data can be in a segment with any name you wish. However, you must be sure to save the segment registers, DS in particular, before you load them with the address of local segments, since they must be restored before the procedure returns.

External Data

Small and Medium Models of Computation. Variables declared in PL/M-86 modules will be in either the "CONST" or in the "DATA" segment. A variable will be in the "CONST" segment if it was initialized using the DATA attribute, e.g., DECLARE A BYTE DATA (3);. A variable will be in the "DATA" segment if it is not initialized, or it is initialized using the INITIAL attribute, e.g., DECLARE B WORD INITIAL (0FFFFH);.

If you know which segment the external data is declared in, then you may wish to declare that same segment in the ASM86 module. For example, if both A and B are in the CONST segment, this is how it might appear:

```
CONST SEGMENT PUBLIC 'CONST'
EXTRN A: BYTE,  B:WORD
CONST ENDS
DGROUP GROUP CONST
ASSUME DS:DGROUP, SS:DGROUP
```

However, if you don't know what segment the variable is in then you can simply say:

```
EXTRN A: BYTE,   B:   WORD
DGROUP   GROUP   SEG A, SEG B
ASSUME   DS:DGROUP, SS:DGROUP
```

If A and B are both in the CONST segment, then this last example is identical to the one above it.

The DGROUP contains four segments: CONST, DATA, STACK and MEMORY. You need only declare the DGROUP with those segments you explicitly reference.

Large Model of Computation. Variables defined with the DATA attribute are placed in the code segment. The code and data segments cannot be named explicitly. The reason for this is that PL/M-86 prefixes "DATA" and "CODE" with the module name, and the two are separated by a dot, e.g., MYPROG.DATA is a large model data segment. ASM86 does not allow the dot to be a character in the identifier. The data and code segments are non-combinable, NOT PUBLIC. Therefore, in order to reference external data, variables declared in PL/M-86 modules, you will have to use "SEG VAR" either as a group member or directly in the assume statement:

```
ASSUME DS: SEG A
```

```
PL/M-86 COMPILER    PLM86CALLINGSEQUENCES


ISIS-II PL/M-86      COMPILATION OF MODULE PLM86CALLINGSEQUENCES
OBJECT MODULE PLACED IN :F1:exampl.OBJ
COMPILER INVOKED BY:  :f3:plm86 :F1:exampl.p86 PRINT(SMALL)

                                                    SMALL MODEL OF COMPUTATION
             $CODE
   1         plm86$calling$sequences: DO;

   2    1    DECLARE i BYTE;
   3    1    DECLARE j WORD;                    PL/M-86 DECLARATION
   4    1    DECLARE k INTEGER;                 OF THE ASM86 PROCEDURE
   5    1    DECLARE l POINTER;

   6    1    p: PROCEDURE (bvalue, wvalue, intvalue, pvalue) EXTERNAL;
   7    2         DECLARE  bvalue                            BYTE,
                         wvalue                             WORD,
                               intvalue                     INTEGER,
                                     pvalue                 POINTER;
   8    2    END p;

   9    1    i = 0FFH;
                                                  ; STATEMENT # 9
             0002  FA              CLI
             0003  2E8E160000      MOV     SS,CS:@@STACK$FRAME
             0008  BC0A00          MOV     SP,@@STACK$OFFSET
             000B  8BEC            MOV     BP,SP
             000D  16              PUSH    SS
             000E  1F              POP     DS
             000F  FB              STI                          THE CALLING SEQUENCE
             0010  C606060000FF    MOV     I,0FFH
  10    1    j = 0FFFFH;
                                                  ; STATEMENT # 10
             0015  C7060000FFFF    MOV     J,0FFFFH
  11    1    k = 31415;
                                                  ; STATEMENT # 11
             001B  C7060200B77A    MOV     K,7AB7H
  12    1    l = @k;
                                                  ; STATEMENT # 12

             0021  C7060400020D    MOV     L,OFFSET(K)
  13    1    CALL p(i, j, k, l);
                                                  ; STATEMENT # 13
             0027  B0FF            MOV     AL,0FFH
             0029  50              PUSH    AX       ; 1  PUSH i
             002A  B8FFFF          MOV     AX,0FFFFH
             002D  50              PUSH    AX       ; 2  PUSH j
             002E  B8B77A          MOV     AX,7AB7H
             0031  50              PUSH    AX       ; 3  PUSH k
             0032  B80200          MOV     AX,OFFSET(K)
             0035  50              PUSH    AX       ; 4  PUSH l
             0036  E80000          CALL    P

  14    1    END plm86$calling$sequences;
                                                  ; STATEMENT # 14
             0039  FB              STI
             003A  F4              HLT



PL/M-86 COMPILER    PLM86CALLINGSEQUENCES




MODULE INFORMATION:

     CODE AREA SIZE      = 003BH      59D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 0007H       7D
     MAXIMUM STACK SIZE  = 000AH      10D
     24 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER     PLM86CALLINGSEQUENCES

ISIS-II PL/M-86      COMPILATION OF MODULE PLM86CALLINGSEQUENCES
OBJECT MODULE PLACED IN :F1:exampl.OBJ
COMPILER INVOKED BY:  :f3:plm86 :F1:exampl.p86 PRINT(MEDIUM) MEDIUM
```

MEDIUM MODEL OF COMPUTATION

```
                $CODE
   1            plm86$calling$sequences: DO;

   2    1       DECLARE i BYTE;
   3    1       DECLARE j WORD;
   4    1       DECLARE k INTEGER;
   5    1       DECLARE l POINTER;

   6    1       p: PROCEDURE (bvalue, wvalue, intvalue, pvalue) EXTERNAL;
   7    2            DECLARE  bvalue                            BYTE,
                                 wvalue                        WORD,
                                       intvalue                INTEGER,
                                             pvalue            POINTER;
   8    2       END p;

   9    1       i = 0FFH;                                      ; STATEMENT # 9
                0002  F6
                0003  2E
                0008  B
                000B  8BEC
                000D  16
                000E  1F
                000F  FB
                0010  C606
  10    1       j = 0FFFFH;                                    ; STATEMENT # 10
                0015  C7060000FFFF    MOV     J,0FFFFH
  11    1       k = 31415;                                     ; STATEMENT # 11
                001B  C7060200B77A    MOV     K,7AB7H
  12    1       l = @k;                                        ; STATEMENT # 12
                0021  8D060200        LEA     AX,K
                0025  89060400        MOV     L,AX
                0029  8C1E0600        MOV     L+2H,DS
  13    1       CALL p(i, j, k, l);                            ; STATEMENT # 13
                002D  B1FF            MOV     CL,0FFH
                002F  51              PUSH    CX      ; 1     PUSH i
                0030  B9FFFF          MOV     CX,0FFFFH
                0033  51              PUSH    CX      ; 2     PUSH j
                0034  B9B77A          MOV     CX,7AB7H
                0037  51              PUSH    CX      ; 3     PUSH k
                0038  1E              PUSH    DS      ; 4     PUSH l
                0039  50              PUSH    AX      ; 5
                003A  9A00000000      CALL    P

  14    1       END plm86$calling$sequences;
                                                             ; STATEMENT # 14
```

CALLING SEQUENCE

NOTE: (1) 2 WORDS ARE PUSHED FOR THE POINTER, 1, FIRST DS WHERE DS=SEGMENT (4) AND AX WHERE AX=OFFSET (5). AX WAS LOADED IN THE LINE 12 THAT IS CIRCLED

(2) A FAR CALL IS ILLUSTRATED

SEE PART (1) OF NOTE

```
PL/M-86 COMPILER     PLM86CALLINGSEQUENCES

                003F  FB              STI
                0040  F4              HLT


MODULE INFORMATION:

     CODE AREA SIZE      = 0041H      65D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 0009H       9D
     MAXIMUM STACK SIZE  = 000EH      14D
     24 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER     PLM86CALLINGSEQUENCES


ISIS-II PL/M-86      COMPILATION OF MODULE PLM86CALLINGSEQUENCES
OBJECT MODULE PLACED IN :F1:exampl.OBJ
COMPILER INVOKED BY:  :f3:plm86 :F1:exampl.p86 PRINT(LARGE)  LARGE

                                                    LARGE MODEL OF COMPUTATION
             $CODE
   1         plm86$calling$sequences: DO;

   2    1    DECLARE i BYTE;                          THE CALLING SEQUENCE
   3    1    DECLARE j WORD;                          THE SAME AS FOR MEDIUM
   4    1    DECLARE k INTEGER;
   5    1    DECLARE l POINTER;

   6    1    p: PROCEDURE (bvalue, wvalue, intvalue, pvalue) EXTERNAL;
   7    2         DECLARE  bvalue                              BYTE,
                           wvalue                              WORD,
                                  intvalue                     INTEGER,
                                           pvalue              POINTER;
   8    2    END p;

   9    1    i = 0FFH;
                                                    ; STATEMENT # 9
             0004  FA              CLI
             0005  2E8E160000      MOV     SS,CS:@@STACK$FRAME
             000A  BC1000          MOV     SP,@@STACK$OFFSET
             000D  8BEC            MOV     BP,SP
             000F  2E8E1E0200      MOV     DS,CS:@@DATA$FRAME
             0014  FB              STI
             0015  C606080FF       MOV     I,0FFH
  10    1    j = 0FFFFH;
                                                    ; STATEMENT # 10
             001A  C7060000FFFF    MOV     J,0FFFFH
  11    1    k = 31415;
                                                    ; STATEMENT # 11
             0020  C706020B77A     MOV     K,7AB7H
  12    1    l = @k;
                                                    ; STATEMENT # 12

             0026  8D060200        LEA     AX,K
             002A  890604          MOV     L,AX
             002E  8C1E0600        MOV     L+2H,DS
  13    1    CALL p(i, j, k, l);
                                                    ; STATEMENT # 13
             0032  B1FF            MOV     CL,0FFH
             0034  51              PUSH    CX      ; 1   PUSH i
             0035  B9FFFF          MOV     CX,0FFFFH
             0038  51              PUSH    CX      ; 2   PUSH j
             0039  B9B77A          MOV     CX,7AB7H
             003C  51              PUSH    CX      ; 3   PUSH k
             003D  1E              PUSH    DS      ; 4 ┐
             003E  50              PUSH    AX      ; 5 ┘ PUSH l
             003F  9A00000000      CALL    P

  14    1    END plm86$calling$sequences;
                                                    ; STATEMENT # 14
             0044  FB              STI


PL/M-86 COMPILER     PLM86CALLINGSEQUENCES

             0045  F4              HLT


MODULE INFORMATION:

     CODE AREA SIZE      = 0046H      70D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 0009H       9D
     MAXIMUM STACK SIZE  = 0010H      16D
     24 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
MCS-86 ASSEMBLER    EXMPLE                                                  PAGE    1


ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE EXMPLE
OBJECT MODULE PLACED IN :F5:EXMPLE.OBJ
ASSEMBLER INVOKED BY: A :F5:EXMPLE.A86

LOC  OBJ                  LINE   SOURCE

                             1   ;Example for linking small programs to PL/M-86
                             2
                             3   ;The small model consists of two groups:  CGROUP and DGROUP.  The ASM86
                             4   ;program must contain a declaration of the CGROUP.  This group must be
                             5   ;ASSUMEd in to the CS register before the beginning of the procedure.
                             6   ;If the procedure does not reference any external or local data, then
                             7   ;DGROUP does not have to be included in the program.  If, however,
                             8   ;there are references to local or external data then DGROUP must also
                             9   ;be declared and ASSUMEd in to the DS register.  At your option it
                            10   ;may be ASSUMEd into the SS register
                            11
                            12   ;The CGROUP contains one segment, CODE.  Typically, there is
                            13   ;no need to declare a CONST segment, unless your program has constant
                            14   ;values. (They might end up in ROM.)  The DGROUP contains four segments:
                            15   ;CONST, DATA, STACK, and MEMORY.  If local data storage is used, then the DATA
                            16   ;segment should be used for storage.  Include the STACK segment only if
                            17   ;you intend to use the stack
                            18   ;
                            19
                            20   CGROUP   GROUP  CODE
                            21   DGROUP   GROUP   DATA
                            22   ASSUME CS: CGROUP, DS:' DGROUP
                            23
----                        24   DATA SEGMENT PUBLIC 'DATA'   ;MUST BE PUBLIC AND SHOULD HAVE CLASS
0000 ??                     25   local_j DB ?                 ;-NAME "DATA"
----                        26   DATA ENDS                    ;USED TO HOLD THE LOCAL -J VALUE
                            27
                            28   PUBLIC P
                            29
----                        30   CODE SEGMENT PUBLIC 'CODE'   ;PUBLIC AND  CODE' ARE REQUIRED
                            31
                            32   ;P DOES NOTHING EXCEPT REFERENCE THE 4 PARAMETERS.  NOTICE THAT IT
                            33   ;SAVES THE BP AND THE COPIES SP INTO IT.  THIS ALLOWS THE PARAMETERS
                            34   ;TO BE OBTAINED CONVENIENTLY VIA THE BP REGISTER.
                            35   ;
0000                        36   P  PROC NEAR                 ;NEAR PROCEDURE SINCE IT IS BEING
                            37                                ;-CALLED BY 'SMALL' PL/M-86 PROGRAM
0000 55                     38      PUSH BP                   ;SAVE THE BP REGISTER
0001 8BEC                   39      MOV  BP,SP                ;POINT BP AT THE TOP OF THE STACK
0003 8B4604                 40      MOV  AX,[BP+4]            ;GET L
0006 8B5E06                 41      MOV  BX,[BP+6]            ;GET K
0009 8B4E08                 42      MOV  CX,[BP+8]            ;MOVE J INTO LOCAL STORAGE
000C 880E0000         R     43      MOV local_j, CL
0010 8A560A                 44      MOV  DL, [BP+10]          ;GET I RECALL THAT I IS A BYTE
0013 C20800                 45      RET   8                   ;RETURN POPPING THE PARAMETERS
                            46
                            47   P  ENDP
----                        48   CODE ENDS
                            49        END


SYMBOL TABLE LISTING                                                        PAGE    2
------ ----- -------


NAME      TYPE      VALUE  ATTRIBUTES

??SEG     SEGMENT          SIZE=0000H PARA PUBLIC
CGROUP.   GROUP            CODE
CODE      SEGMENT          SIZE=0016H PARA PUBLIC 'CODE'
DATA      SEGMENT          SIZE=0001H PARA PUBLIC 'DATA'
DGROUP    GROUP            DATA
LOCAL_J   V BYTE   0000H   DATA
P         L NEAR   0000H   CODE PUBLIC


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPENDIX C
SAMPLE PROGRAM: SENDING CHARACTERS TO THE CRT

This appendix contains a sample program that is commonly coded. It is intended as an (immediately usable) illustration of the MCS-86 assembly language.

```
MCS-86 ASSEMBLER    SDK86P                                                    PAGE    1


ISIS-II MCS-86 ASSEMBLER      ASSEMBLY OF MODULE SDK86P
OBJECT MODULE PLACED IN :F4:SDK86P.OBJ
ASSEMBLER INVOKED BY: :F4:A :F4:SDK86P.A86

LOC  OBJ                  LINE   SOURCE

                             1   ;THIS SDK86 PROGRAM ECHOS CHARACTERS FROM A KEYBOARD TO A CRT,
                             2   ;AND GENERATES A FREQUENCY DISTRIBUTION OF CHARACTER OCCURENCES.
                             3
----                         4   RAMSEG  SEGMENT AT 30H                    ;PLACE RAM SEGMENT AT 300H
0000 (128                    5   FREQNCY DB      128 DUP(0)                ;INITIALISE OCCURENCE COUNT ARRAY
     00
     )
                             6                                             ;TO ZERO
0080 (16                     7           DW      16 DUP(?)                 ;RESERVE AN AREA FOR THE STACK
     ????
     )
0094                         8   STKTOP  LABEL   WORD                      ;INTIAL STACK POINTER POSITION
----                         9   RAMSEG ENDS
                            10
                            11
----                        12   ROMSEG  SEGMENT AT 200H                   ;PLACE ROM SEGMENT AT 200H
                            13           ASSUME CS:ROMSEG,DS:RAMSEG,SS:RAMSEG,ES:NOTHING
                            14
  0FF0                      15   USARTDATA       EQU     0FF0H            ;8251A DATA PORT ON SDK86
  0FF2                      16   USARTSTAT       EQU     0FF2H            ;8251A STATUS PORT
0000 3000                   17   SETSEG  DW      RAMSEG                   ;SEGMENT ADDRESS OF BEGINNING OF RAMSEG
                            18
0002 2E8E1E0000             19   START:  MOV     DS,CS:SETSEG             ;SET UP DATA SEGMENT AS IN ASSUME
0007 2E8E160000             20           MOV     SS,SETSEG                ;SET UP STACK SEGMENT
000C BC9400                 21           MOV     SP,OFFSET STKTOP         ;SET INITIAL STACK POINTER VALUE
                            22
000F E81900                 23   LOOP1:  CALL    CI                       ;READ CHARACTER TO AL
0012 8AE0                   24           MOV     AH,AL
0014 E80500                 25           CALL    CO
0017 E81E00                 26           CALL    COUNTIT                  ;COUNT OCCURENCE OF CHARACTER IN AL
001A EBF3                   27           JMP     LOOP1
                            28
001C 8AF20F                 29   CO:     MOV     DX,USARTSTAT             ;IF USART NOT READY FOR CHARACTER
001F EC                     30           IN      AL,DX                    ;INPUT A BYTE INTO AL WITH PORT #
                            31                                            ;IN DX
0020 2401                   32           AND     AL,1
0022 74F8                   33           JZ      CO                       ;THEN WAIT
0024 8AF00F                 34           MOV     DX,USARTDATA             ;ELSE OUTPUT CHARACTER
0027 8AC4                   35           MOV     AL,AH
0029 EE                     36           OUT     DX,AL
002A C3                     37           RET
                            38
002B 8AF20F                 39   CI:     MOV     DX,USARTSTAT             ;IF CHARACTER NOT READY
002E EC                     40           IN      AL,DX
002F 2402                   41           AND     AL,2
0031 74F8                   42           JZ      CI                       ;THEN WAIT
0033 8AF00F                 43           MOV     DX,USARTDATA             ;ELSE BRING IN CHARACTER
0036 EC                     44           IN      AL,DX
0037 C3                     45           RET


MCS-86 ASSEMBLER    SDK86P                                                    PAGE    2

LOC  OBJ                  LINE   SOURCE

                            46
0038                        47   COUNTIT PROC    NEAR                     ;EXPECT CHARACTER IN AL
0038 32E4                   48           XOR     AH,AH                    ;ZERO AH
003A 8BF0                   49           MOV     SI,AX                    ;16 BIT INDEX INTO FREQUENCY TABLE
                            50                                            ;IN SI
003C FE84                   51           INC     FREQNCY[SI]              ;INCREMENT ARRAY INDEXED BY SI
003E C3                     52           RET
                            53   COUNTIT ENDP
                            54
----                        55   ROMSEG  ENDS
                            56
  0002                      57           END     START
```

APPENDIX D
RULES FOR SHORTENING CONTROLS

Any of the controls mentioned in this book have a legal short form. This appendix contains these rules. The rules can be used to shorten most of the controls found in Intel languages. Here are the rules:

- if the control is a one syllable word, use the first two characters;
- if the control is a polysyllabic word, but not a compound word, use the first character from the first two syllables;
- if the control is a compound word, use the first character from each of the compounding words; however,
- if the control begins with NO, NO cannot be shortened.

GLOSSARY

This glossary contains terms that may have specific Intel 'flavor'; it is intended to be used as an indicator of the specific usage.

Assembler—a computer program that translates assembly source code into object code or machine code.

Assembler Language—a source language for the MCS-86 assembler that includes symbolic machine language statements in which there is generally a direct correspondence with the instruction format and data format of the computer.

Assemble—to prepare a machine language program from a symbolic language program by substituting absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

Code—a set of machine instructions giving a set of unambiguous rules specifying the way in which data is to be manipulated.

Console device—that part of the computer used for communication between the operator or engineer and the computer.

Data—a representation of facts, concepts or instructions in a formalized manner.

Default—the choice among exclusive alternatives made by the system when no explicit choice is specified by the user.

Directive—a command to the assembler that does not generate object code.

Expression—a source language combination of one or more operations and operands often represented by a combination of terms possibly within paired parentheses.

File—a collection of related records treated as a unit.

Instruction—a statement that specifies an operation and the values or locations of its operands.

Listing file—a file that can be printed out, that lists the source language statements and translations and errors.

LINK86—a program that prepares the output of a language translator for execution. It combines separately produced object modules, resolves symbolic references among them, and produces code ready for LOC-86.

LOC86—a program that produces executable code.

Models of Computation—a set of expectations as to the location of data, the location of instruction sequences and the use of registers; various methods of developing programs specifying arrangment of code, data, stack, constant and memory in the available memory address space.

Object file—output from the assembler which is itself executable machine code or ready for further processing to produce executable machine code.

Overlay—the technique of repeatedly using the same internal storage during different stages of a program.

Pass—one cycle of processing a body of data; ASM86 is a two-pass assembler (this means that it reads the source code twice).

Source file—a file that constitutes input to a language translator.

Symbol—a manner of referring to a resource of the computer; a representation of something by reason of relationship, association or convention.

Syntax—the structure of expressions in a language; acceptable input to a program; can be statements, commands or directives.

Segment—any contiguous block of memory up to 64K (physical segment); a unit of data and/or code in your assembly program, also, contents of a segment register (logical segment); the basic unit of relocation and linkage.

Table—a collection of data in which each item is uniquely identified by a label, or by its position relative to the other items or by some other means.

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

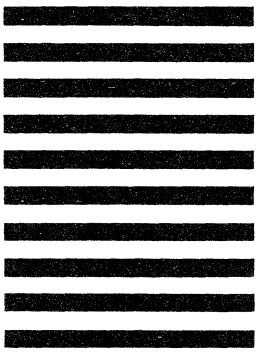
NAME _____ DATE _____
TITLE _____
COMPANY NAME/DEPARTMENT _____
ADDRESS _____
CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications
3065 Bowers Avenue
Santa Clara, CA 95051