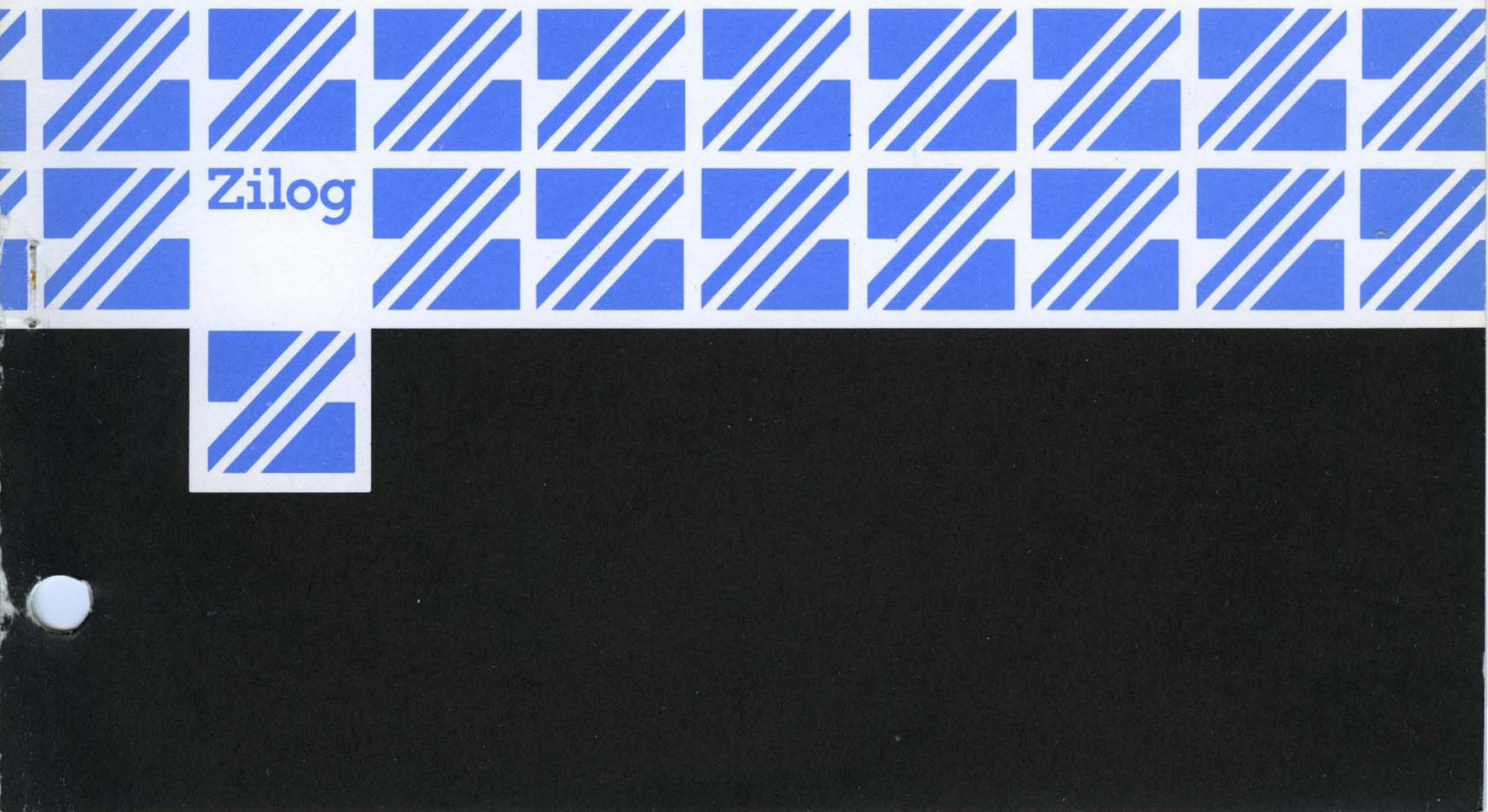
Zilog

Z80™-RIO

Relocating Assembler and Linker User's Manual



Copyright © 1978 by Zilog, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Zilog.

Zilog assumes no responsibility for the use of any circuitry other than circuitry embodied in a Zilog product. No other circuit patent licenses are implied.

Z80-RIO
RELOCATING ASSEMBLER AND LINKER
USER'S MANUAL

TABLE OF CONTENTS

OVERVIEW 2

1.0 ASSEMBLER 5

1.1 THE ASSEMBLY LANGUAGE 5

1.2 ASSEMBLY LANGUAGE CONVENTIONS 6

Delimiters 6
Comments 6
Labels 6
Opcodes and Operands 7
Numbers 7
Character Values 8
Upper/Lower Case 8

1.3 EXPRESSIONS 9

Table of Operators 9
Mode of Expressions 10
Relative Addressing 12

1.4 LISTING FORMAT 13

Cross-Reference 14

1.5 PSEUDO-OPS 16

Data Definition: DEFB, DEFW, DEFM, DEFT. . 16
Storage Definition: DEFS 18
Source Termination: END 19
Symbol Definition: EQU, DEFL, GLOBAL, EXTERNAL 20
Sample Relocatable Program 23
Reference Counter Control: ORG 25
Conditional Assembly: COND and ENDC . . . 26

1.6 MACROS 29

Macro Definition 29
Macro Calls and Macro Expansion 30
Symbol Generator 31
Recursion 31
Listing Format 32

1.7 ASSEMBLER COMMANDS 40

Eject, Heading, List, Maclist, Include . . 40

1.8 ASSEMBLER COMMAND LINE OPTIONS 42

Using the Assembler 42
Module Identification 44
Options: Macro, Symbol, Xref, Absolute, NOList, NOMaclist, NOObject, NOWarning, Date 45
I/O Routing Options 47
Logical Unit Definitions 48
Fatal Error Messages 48

2.0 LINKER 50

2.1 FUNCTION 50

Features 50

2.2 USING THE LINKER 51

Non-fatal Errors 52
Assignment of Module Origins 53
Segment Generation 53
Link-only Modules 55
Module Identification 55

2.3 LINKER COMMAND LINE OPTIONS 57

Entry, LET, Map, Name, NOMap, NOWarning, Print, RLength, STacksize, SYmbol 57
Logical Unit Definitions 60
Fatal Error Messages 60

2.4 MAP FORMAT 63

2.5 OVERLAYS 64

APPENDICES

Appendix A - Opcode Listing

Appendix B - Object Module Format

Appendix C - Assembly Error Messages

PREFACE

This manual describes the relocating assembler (ASM) and linker (LINK) programs which support the translation of Z80 assembly language source code into executable machine code on the Zilog RIO operating system.

This manual contains two chapters -- Chapter 1 covers assembly language conventions, pseudo-ops, and how to invoke the Assembler with different command options. It does not describe the Z80 instruction set or mnemonics, since this information can be found in the publication

Z80-Assembly Language Programming Manual

Chapter 2 covers the functional characteristics of the Linker and how to invoke it with different command options.

RELATED PUBLICATIONS

RIO Operating System User's Manual
Z80-CPU Technical Manual

OVERVIEW

The relocating assembler ASM accepts a source file (a symbolic representation of a portion of a program in Z80 assembly language) and translates it into an object module. It may also produce a listing file which contains the source, assembled code, and optional symbol cross-reference. The features of the relocating assembler include macros and conditional assembly, as well as the capability to produce either relocatable or absolute object code.

All object code must be processed by the linker LINK before it is ready for execution. The linker accepts several object modules and combines them into a single program in the form of an executable procedure file. It may also produce a load map describing the assigned addresses and lengths of modules. An optional symbol file may be created for use by a symbolic debug package.

Relocation

Relocation refers to the ability to bind a program module and its data to a particular memory area at a time after the assembly process. The output of the assembler is an object module which contains enough information to allow the linker to assign that module a memory area. Since many modules may be loaded together to form a complete program, a need for inter-module communication arises. For example, one module may contain a call to a routine which was assembled as part of another module and may be located in some arbitrary part of memory. Therefore, the assembler must provide information in the object module which allows the linker to link inter-module references.

There are several major advantages to using the relocating assembler as compared to the absolute assembler:

1) Assignment of modules to memory areas can be handled by the linker rather than requiring the programmer to assign fixed absolute locations via the "ORG" pseudo-op, and thus modules can be relocated without requiring re-assembly.

2) If errors are found in one module, only that one module needs to be re-assembled and then re-linked with the other modules, thus increasing software productivity.

3) Programs can be structured into independent modules, coded separately and assembled, even though the other modules may not yet exist. This separation of

functionality also enhances program maintenance and verification.

4) Libraries of commonly used modules can be built and then linked with programs without requiring their re-assembly.

5) Communications between overlay segments can be achieved through methods similar to normal (non-overlay) inter-module references.

Relocatability is specified on a per module basis; in other words, a program may contain some relocatable modules and some absolute modules, but a single module is either entirely absolute or entirely relocatable. The notion of "absolute" is that there is a one-to-one direct correspondence between the assigned address of an instruction within a module during assembly, and the memory address of that instruction during program execution. Thus an absolute module whose origin is "absolute 0" will have its first instruction located at memory location 0. Absolute modules usually contain ORG pseudo-ops to set the origin of sections of code, since the default origin is 0 (see ORG pseudo-op).

Modules are by default assembled as relocatable modules. The notion of "relocatable" is that the assigned address of an instruction within a module during assembly, is a relative offset from an origin which is assigned an absolute address by the Linker. During program execution, the instruction will be located at the memory location specified by the assigned origin plus the relative offset. Thus, a relocatable module whose first instruction (which is located at "relocatable 0") will have that instruction located at the memory location which is the assigned origin of the module as determined by the Linker.

To achieve relocation in the Z80, instructions which reference memory addresses (such as LD, CALL, JP) and data values which serve as pointers to memory locations (such as PTR: DEFW DATA where DATA is a label in the program module) must be altered at linkage time to reflect the assigned absolute addresses. However, as far as the programmer is concerned, instructions and addressing remain the same as in the absolute assembler.

Inter-module Communication

To allow instructions to refer to "names" (either data values or entry points) in other separately assembled modules, the assembler provides two pseudo-ops, GLOBAL and EXTERNAL. The syntax is the pseudo-op, followed by a list of names.

The meaning of GLOBAL is that each of the listed names is defined in this module, and the name is made available to other modules which contain an EXTERNAL declaration for that name. The complementary form then is EXTERNAL, which declares each of the listed names to be defined in some other module. Notice that the GLOBAL name may be in either an absolute module or a relocatable module. A portion of the object module contains a list of both the GLOBALs which are defined in the module, as well as the EXTERNALs which the module references. One function of the Linker is to match all the EXTERNALs with the appropriate GLOBALs so that every instruction will reference the correct address during program execution.

Programming Hints

A general approach to designing a program, then, would be to divide it into manageable portions, each of which constitutes a module. Most modules can be relocatable, that is, it does not matter where the module is placed in memory as long as it is correctly referenced by other modules through the use of GLOBAL and EXTERNAL names. Parts of the program may be coded, assembled, linked and tested. Then other modules may be added in a step-wise fashion without re-assembly of already-tested modules, thereby increasing programming productivity.

For those applications which will ultimately be implemented in a system configuration where the program code is in ROM and the data area is in RAM, it is a good idea to segregate the data into one or two modules with GLOBAL declarations for each data address. During the program development stage, these data modules might be relocated anywhere, and then assigned a particular origin depending on the memory configuration only when the program is finished.

1.0 ASSEMBLER

1.1 THE ASSEMBLY LANGUAGE

The assembly language of the Z80 is designed to minimize the number of different opcodes corresponding to the set of basic machine operations and to provide for a consistent description of instruction operands. The nomenclature has been defined with special emphasis on mnemonic value and readability. For example, the movement of data is indicated primarily by a single opcode, LD (standing for LoaD), regardless of whether the movement is between different registers or between registers and memory.

The first operand of an LD instruction is the destination of the operation, and the second operand is the source of the operation. For example,

```
LD A,B
```

indicates that the contents of the second operand, register B, are to be transferred to the first operand, register A. Similarly,

```
LD C,3FH
```

indicates that the constant 3FH is to be loaded into the register C. Enclosing an operand wholly in parentheses indicates a memory location addressed by the contents of the parentheses. For example,

```
LD HL,(1200)
```

indicates the contents of memory locations 1200 and 1201 are to be loaded into the 16-bit register pair HL. Similarly,

```
LD (IX+6),C
```

indicates the contents of the register C are to be stored into the memory location addressed by the current value of the 16-bit index register IX plus 6.

The regular formation of assembly instructions minimizes the number of mnemonics and format rules that the user must learn and manipulate. Additionally, the resulting programs are easier to interpret, which in turn reduces programming errors and improves software maintenance.

1.2 ASSEMBLY LANGUAGE CONVENTIONS

An assembly language source program consists of a sequence of statements in the order which defines the user's program. Each statement corresponds to a line of text of up to 128 characters ended by a carriage return. Each statement may contain zero or more of the following four fields which are identified primarily by their order in the statement line and by specific delimiter characters as indicated below.

```
LABEL   OPCODE     OPERAND     COMMENT
```

For example, here is a statement which contains all four fields:

```
LOOP:   LD         HL,VALUE    ;GET THE VALUE
```

a. DELIMITERS - Opcodes, operands, and labels must be separated from each other by one or more commas, spaces or tabs. Notice that anywhere a comma could be used, a space or tab will be equivalent, and vice-versa.

b. COMMENTS - Comments are not a functional part of an assembly language source program, but instead are used for program documentation to add clarity, and to facilitate software maintenance. A comment is defined as any string following a semicolon in a line, and is ignored by the assembler. Comments can begin in any column, so a statement might consist of only a comment.

c. LABELS - A label is a symbol representing up to 16 bits of information and is used to specify an address or data. If the programmer attempts to use a symbol that has been defined to have a value greater than 8 bits for an 8-bit data constant, the assembler will generate an error message.

A label is composed of a string of one or more characters, of which the first six characters must be unique. For example, the labels LONGNAME and LONGNAMEALSO will be considered to be the same label. The first character must be alphabetic and any following characters must be either alphanumeric (A...Z or 0...9), the question-mark character (?) or the underbar character (_). Any other characters within a label will cause an error message. A label can start in any column if immediately followed by a colon. It does not require a colon if started in column one.

Labels are entered into a symbol table along with the value assigned by the assembler. The symbol table normally resides in RAM, but automatically overflows to a disk file if necessary, so there is virtually no limit to the number of labels that can occur in a single program.

Labels are normally assigned the value of the reference counter for the statement in which they occur. The reference counter corresponds to the CPU program counter and is used to assign and calculate machine-language addresses within an object module. The symbol $ is used to represent the value of the reference counter of the first byte of the current instruction, and can be used in general expressions (see Expressions).

d. OPCODES AND OPERANDS

Z80 machine instructions are represented by specific mnemonics, usually consisting of a single opcode (such as LD, CALL, etc.) and zero or more operands (such as a register designator like HL or a condition code like Z). See the Appendix for a complete listing of all opcode/operand combinations.

The Assembler also recognizes certain pseudo-ops (sometimes called directives) which occur in the opcode/operand positions (see Pseudo-ops). Certain opcodes allow a particular operand to be the value of an arbitrary expression using arithmetic and logical operators (such as addition or multiplication) and operands (such as labels, numbers or character values).

Numbers

The Assembler will accept numbers in the following different bases: including binary, octal, decimal and hexadecimal. Numbers must always start with a digit (leading zeros are sufficient), and may be followed immediately by a single letter which signifies the base of the numbers ('B' for binary, 'O' or 'Q' for octal, 'D' for decimal, and 'H' for hexadecimal). If no base is specififed, decimal is assumed. For example, here is the same number represented in each of the four bases:

1011100B, 134Q, 134O, 92, 92D, 05CH

Character Values

The Assembler will accept as operands ASCII characters bounded by a single quote mark with the corresponding ASCII value (e.g., 'A' has value 41H). Because all operands are evaluated as 16-bit quantities, strings of characters longer than two will be truncated in value to the first two characters, so that 'AB' and 'ABCDE' will have the same value. There is a method to define data storage which is initialized to a string of up to 63 characters (see DEFM, DEFT pseudo-ops). The quote character itself is represented by two successive single quote marks within a string (e.g., LD A, '''' causes the A register to be loaded with the value 27H, which is the ASCII value for the single quote character). A special string called the null string (which has the value of 0) can be represented by just two successive single quotes as in ''.

Upper/Lower Case

The Assembler processes source text which contains both upper and lower case alphabetic characters in the following manner. All opcodes and keywords, such as register names or condition codes, must be either all capitals or all lower case. Label names may consist of any permutation of upper and lower case; however, two names which differ in case will be treated as two different names. Thus LABEL, label, and LaBeL will be considered as three different names. Notice that one could use a mixture of case to allow definition of labels or macros which look similar to opcodes, such as Push, or LdiR, without redefining the meaning of the opcode. All assembler commands, such as *List or *Include (see Assembler Commands), can be in any combination of upper or lower case, as can the arithmetic operators such as .NOT., .AND., or .EQ., and numbers can be any mixture of case, such as 0ffffh, 0AbCdH, or 01100lb.

1.3 EXPRESSIONS

The Assembler will accept a wide range of expressions involving arithmetic and logical operations. All expressions are evaluated left to right, except that unary operators are performed first, exponentiation next, multiplication, division, modulo, and shifts next, followed by addition and subtraction, then logical operations and comparisons. The following table is arranged in a hierarchy of descending precedence.

Table of Operators

OPERATOR	FUNCTION	PRECEDENCE
+	UNARY PLUS	1
-	UNARY MINUS	1
.NOT. or \	LOGICAL NOT	1
.RES.	RESULT	1
**	EXPONENTIATION	2
*	MULTIPLICATON	3
/	DIVISION	3
.MOD.	MODULO	3
.SHR.	LOGICAL SHIFT RIGHT	3
.SHL.	LOGICAL SHIFT LEFT	3
+	ADDITION	4
-	SUBTRACTION	4
.AND. or &	LOGICAL AND	5
.OR. or ^	LOGICAL OR	6
.XOR.	LOGICAL XOR	6
.EQ. or =	EQUALS	7
.GT. or >	GREATER THAN	7
.LT. or <	LESS THAN	7
.UGT.	UNSIGNED GREATER THAN	7
.ULT.	UNSIGNED LESS THAN	7

Parentheses can be used to ensure correct expression evaluation. Note, however, that enclosing an expression wholly in parentheses indicates a reference to a memory address.

Delimiters such as spaces or commas are not allowed within an expression since they serve to separate the expression from other portions of the statement.

16-bit signed integer arithmetic is used throughout.

Notice that anywhere a number or character value can be used in an expression, a symbol (label) could be used as well. The value of the symbol is always a 16-bit quantity.

The negative of an expression can be formed by a preceding minus sign "-".

For example:

```
LD A,-2 is equivalent to LD A,0FEH,
since 0FEH is the 8-bit two's complement
representation of negative 2.
```

The five comparison operators (.EQ., .GT., .LT., .UGT., and .ULT.) evaluate to a logical True (all ones) if the comparison is true and a logical False (zero) otherwise. The operators .GT. and .LT. deal with signed numbers whereas .UGT. and .ULT. assume unsigned arguments.

For example:

```
1.EQ.2 is equivalent to 0,
(2+2).LT.5 is equivalent to 0FFFFH
and '0'=0 is equivalent to 0
```

The Result operator (.RES.) causes overflow to be suppressed during evaluation of its argument, thus overflow is not flagged with an error message.

For example:

```
LD BC,7FFFH+1 would cause an error message,
whereas LD BC,.RES.(7FFFH+1) would not.
```

The Modulo operator (.MOD.) is defined as:

```
X.MOD.Y = X-Y*(X/Y)
where the division (X/Y) is integer division.
```

For example:

```
8.MOD.3 is 2 and 0ABH.MOD.0BH is 6
```

The Shift operators (.SHR.,.SHL.) shift their first argument right or left by the number of positions given in their second argument. Zeros are shifted into the high-order or low-order bits, respectively.

Mode of Expressions

Arithmetic expressions have a mode associated with them: absolute, relocatable or external as defined below. A symbol in an expression is absolute if its value is a constant (either a number, character value, or a symbol which has been EQUated to a constant expression). A symbol in an expression is relocatable if its value is subject to relocation after assembly (either a label or the reference counter symbol '$'). Simple relocation means that only a

single relocation factor needs to be added to the relocatable value at load time. GLOBAL names within a relocatable module have a relocatable mode. GLOBAL names within an absolute module have an absolute mode. A symbol is external if it is declared in an EXTERNAL pseudo-op.

In the following, A stands for an absolute symbol or expression, R stands for a relocatable symbol or expression, and X stands for an external symbol or expression. <operator> is one of the standard arithmetic symbols such as +, -, *, .OR., .AND., etc. <comparison> is either .EQ., .LT., .GT., .UGT., .ULT. -- the resulting value is either True (all ones) or False (all zeros).

A is defined as either:

1) absolute symbol
2) A <operator> A
3) +A or -A or .NOT.A or .RES.A (unary operators)
4) R - R
5) A <comparison> A
6) R <comparison> R

R is defined as either:

1) relocatable symbol
2) R + A
3) R - A
4) A + R
5) +R (unary plus)
6) .RES.R (ignores overflow in a relocatable expression)

X is defined as either:

1) external symbol
2) X + A
3) X - A
4) A + X
5) +X (unary plus)
6) .RES.X (ignores overflow in an external expression)

Invalid expressions include:

```
R + R           ; this is not simple relocatable
                  since the relocation factor would
                  have to be added twice
R * A
A - R
X + R           ; cannot add external and relocatable
R = A           ; cannot compare absolute with
                  relocatable
X = X           ; cannot compare externals
```

Relative Addressing

In specifying relative addressing with either the JR (Jump Relative) or DJNZ (Decrement and Jump if Not Zero) instructions, the Assembler automatically subtracts the value of the next instruction's reference counter from the value given in the operand field to form the relative address for the jump instruction. For example:

```
JR      C,LOOP
```

will jump to the instruction labeled LOOP if the Carry flag is set. The limits on the range of a relative address is 128 bytes in either direction from the reference counter of the next instruction (notice that JR and DJNZ are two-byte instructions). Thus LOOP must be a label within the range <-126, +129> from the JR instruction. An error message will be generated if this range is exceeded.

Notice that the mode of a label and the reference counter must match so that the result of the implied subtraction is absolute (both must be relocatable or both must be absolute). If not, an error message will be generated.

An expression which evaluates to a displacement in the range <-126, +129> can be added to the reference counter to form a relative address. For example:

```
JR      C,$+5
```

will jump to the instruction which is 5 bytes beyond the current instruction.

1.4 LISTING FORMAT

The Assembler produces a paginated listing of the source program along with the generated object code, plus an optional cross-reference listing of all symbols. A description of the various fields in the listing format follows. Refer also to the sample listing.

Heading	Each page has a heading which contains the name of the source file, a page number, column headings as explained below, and the Assembler version number. In addition, the heading may contain a user-specified heading (see *Heading Assembler command) and a user-specified string which is usually the date of assembly (see Date option).
LOC	Location: This column contains the value of the reference counter for statements which generate object code. It is blank otherwise.
OBJ CODE	Object code: This column contains the value of generated object code, up to a maximum of four bytes of code. It is blank if a statement does not generate object code.
M	Mode: This single character column indicates the mode of an instruction. An 'R' indicates the instruction contains a reference to a relocatable address, an 'X' indicates the instruction contains a reference to an external address, and a blank indicates the instruction is absolute. Notice that an instruction which contains either a relocatable or external reference will be modified by the Linker; therefore, the value in the object code listing does not reflect the true value of the instruction during program execution.
STMT	Statement number: This column contains the sequence number of each source statement.

SOURCE STATEMENT	The remainder of the line contains the source statement text. Listing lines longer than 132 characters will be truncated.

Cross-reference

If the cross-reference option is specified, then an alphabetical list of all the symbols in the source program, along with the statement number of each line which contains a reference to that symbol, will be appended to the listing file. The format is a symbol, followed by its 16-bit value, a mode indicator, the statement line on which the symbol was defined, and a list of the statement lines which refer to the symbol.

The mode of a symbol follows its value, with blank=absolute, 'R'=relocatable, 'G'=GLOBAL, and 'X'=external. Names which are declared EXTERNAL are considered to be defined by the EXTERNAL pseudo-op statement. Names which are declared GLOBAL are considered to be defined where they appear as a label, and considered to be referenced by the GLOBAL statement.

```
                                        multiply        6/15/77     PAGE   1
 LOC    OBJ CODE M STMT SOURCE STATEMENT                                ASM 4.0
                     1          GLOBAL ENTRY,MLTPLY
                     2          EXTERNAL OUTNUM
                     3
                     4 ; TEST OF MULTIPLY ROUTINE
0000    ED5B0D00 R   5 ENTRY:   LD   DE,(DATA)
0004    0E02         6          LD   C,TWO
0006    CD0F00   R   7          CALL MLTPLY     ;DO MULTIPLICATION
0009    CD0000   X   8          CALL OUTNUM     ;OUTPUT VALUE IN HL
000C    76           9          HALT
                    10
000D    AA00        11 DATA:    DEFW 0AAH
                    12
                    13 TWO      EQU 2
                    14
                    15           ;8-BIT BINARY MULTIPLY ROUTINE
                    16           ; MULTIPLIER IN C
                    17           ; MULTIPLICAND IN DE
                    18           ; PARTIAL SUM IN HL
000F    210000      19 MLTPLY:  LD   HL,0
0012    0608        20          LD   B,8
                    21
0014    CB39        22 MLOOP:   SRL  C
0016    3001        23          JR   NC,NOADD
0018    19          24          ADD  HL,DE
                    25
0019    CB23        26 NOADD:   SLA  E
001B    CB12        27          RL   D
001D    10F5        28          DJNZ MLOOP
                    29
001F    C9          30          RET
                    31          END

CROSS REFERENCE                         multiply        6/15/77     PAGE  2
SYMBOL   VAL M DEFN REFS
DATA     000D R   11    5
ENTRY    0000 G    5    1
MLOOP    0014 R   22   28
MLTPLY   000F G   19    1    7
NOADD    0019 R   26   23
OUTNUM   0000 X    2    8
TWO      0002     13    6
```

1.5 PSEUDO-OPS

In addition to normal opcodes which generate executable machine instructions, the Assembler recognizes several pseudo-ops which are used to control the generation of object code. Pseudo-ops have the same basic format as instructions, with the pseudo-op located in the opcode field and any operands following. Labels and comments generally may be used subject to the restrictions listed with the individual pseudo-op descriptions.

Data Definition

DEFB -- Define Byte

The DEFB pseudo-op takes one operand and generates a single byte of object code located at the current value of the reference counter.

Label	Opcode	Operand
optional:	DEFB	expression

If a label is present, it will be assigned the current value of the reference counter, and thus can be used to refer to the data value. The expression must evaluate to a quantity which can be represented in a single byte (8 bits). An error message will be generated if the value is not in the range -128 to 255. Because the Assembler treats all relocatable and external expressions as 16-bit addresses, the mode of an expression used in a DEFB must be absolute, since a 16-bit value will generally not fit into a single byte.

Examples:

Object Code	Label	Opcode	Operand
BD	TRUE:	DEFB	0BDH
04		DEFB	2+2
30	CHAR:	DEFB	'0'

DEFW -- Define Word

The DEFW pseudo-op takes one operand and generates a single word (16 bits or 2 bytes) of object code located at the current value of the reference counter.

Label	Opcode	Operand
optional:	DEFW	expression

The least significant byte is located at the current value of the reference counter, while the most significant byte is located at the next higher location. The order for storing low and then high order bytes is the format for addresses stored in memory; thus the DEFW can be used to store values which "point" to a particular memory address. The mode of the expression can be either absolute, relocatable or external. If a label is present, it will be assigned the current value of the reference counter, and thus can be used to refer to the data value.

The expression may consist of a character value, but notice that only the first two characters in a string are used. The object code will contain the characters in the same order as they appear in the string, not in reverse order.

Examples:

Suppose that PROC1 is a label on an instruction which was located at the reference counter value of 40BDH.

```
Object Code     Label   Opcode  Operand

  BD40          PTR:     DEFW    PROC1
  0400                   DEFW    2+2
  3031          STRNG:   DEFW    '01'
```

DEFM -- Define Message

The DEFM pseudo-op takes one string operand and generates a sequence of bytes in the object code which represents the 7-bit ASCII code for each character in the string.

Label	Opcode	Operand
optional:	DEFM	string

A string is represented by a sequence of characters bounded by single quote marks. The number of characters must be in the range 0 to 63. The single quote character itself is represented by two successive single quote marks within the string. If a label is present, it will be assigned the current value of the reference counter, and thus can be used to refer to the first character in the string.

Notice that only the length byte and the first three characters of a long string will be listed in the object code field in the listing.

Examples:

```
Object Code     Label   Opcode  Operand

48454C4C        MSG:     DEFM   'HELLO THERE'
612762                   DEFM   'a''b'
```

DEFT -- Define Text

The DEFT pseudo-op takes one string operand and generates a sequence of bytes in the object code which represents the 7-bit ASCII code for each character in the string. In addition, it inserts a byte which contains the length of the string as the first byte in the object code.

```
Label           Opcode          Operand

optional:        DEFT           string
```

A string is represented by a sequence of characters bounded by single quote marks. The number of characters must be in the range of 0 to 63. The single quote character itself is represented by two successive single quote marks within the string. If a label is present, it will be assigned the current value of the reference counter and thus can be used to refer to the length byte of the string.

Notice that only the length byte and the first three characters of a long string will be listed in the object code field in the listing.

Examples:

```
Object Code     Label   Opcode  Operand

0B48454C        MSG:     DEFT   'HELLO THERE'
03612762                 DEFT   'a''b'
```

Storage Definition

DEFS -- Define Storage

The DEFS pseudo-op takes one operand which specifies a number of bytes to be reserved for data storage starting at the current value of the reference counter.

```
Label           Opcode          Operand

optional:        DEFS           expression
```

If a label is present, it will be assigned the current value of the reference counter, and thus can be used to refer to the first byte of data storage. The expression can evaluate to any 16-bit quantity; however, the mode must be absolute. Any symbol appearing in the expression must have been defined before the Assembler encounters the expression.

The DEFS pseudo-op reserves storage by incrementing the reference counter by the value of the expression. If the result exceeds a 16-bit quantity, an error is generated and the reference counter is not changed. Since no object code is generated into storage area, the contents of the storage during initial program execution are unpredictable.

Examples:

Notice that the symbol LINELN in the following example must have been previously defined.

```
Label           Opcode          Operand

BUFFER:          DEFS           128
                 DEFS           LINELN+1
```

Source Termination

END -- End of Source

The END pseudo-op signifies the end of the source program, and thus any subsequent text will be ignored.

```
Label           Opcode          Operand

optional:        END
```

If a label is present, it will be assigned the current value of the reference counter. Operands are ignored. If a source program does not contain an END pseudo-op, then the end-of-file mark in the last source file in the assembler command line will signify the end of the program.

Symbol Definition

Labels on instructions are automatically assigned the current value of the reference counter. The pseudo-ops EQU and DEFL can be used to assign arbitrary values to symbols. In addition, to facilitate inter-module communication, certain symbols can be declared to be either GLOBAL or EXTERNAL to a particular module.

EQU -- Equate

The EQU pseudo-op is used to assign the value of an expression to the symbol in the label field.

```
Label           Opcode          Operand

name:            EQU            expression
```

The name in the label field is required and must follow the rules for forming a label (remember the colon is optional if the name starts in column one). The name must not be redefined as a label anywhere else in the source program, or a multiple definition error will occur.

The value of the expression can be any 16-bit value, but any symbol appearing in the expression must have been defined before the Assembler encounters the expression. The symbol being EQUated inherits the mode of the expression, that is, relocatable or absolute. The mode of the expression must not be external. A name which is EQUated to a GLOBAL name in a module will be relocatable if the GLOBAL name is relocatable, otherwise it is absolute; however, the name will not be accessible to other separately assembled modules.

Examples:

Suppose the symbol TWO has the value 2.

```
Label   Opcode  Operand

ASCICR   EQU    0DH             ; ASCII carriage return
FOUR     EQU    TWO+2
SEMICO   EQU    ';'             ; ASCII semicolon
```

DEFL -- Define Label

The DEFL pseudo-op is used to assign the value of an expression to the symbol in the label field. This symbol may be redefined by subsequent DEFL pseudo-ops.

Label	Opcode	Operand
name:	DEFL	expression

The name in the label field is required and must follow the rules for forming a label (remember the colon is optional if the name starts in column one). The DEFL pseudo-op is identical to the EQU pseudo-op except that "name" may appear in multiple DEFL pseudo-ops in the same program. Notice that both the value and the mode can be changed by each occurrence of the DEFL.

Example:

```
Object Code     Label   Opcode  Operand

                FLAG:   DEFL    0
 3E00                   LD      A,FLAG
                FLAG:   DEFL    0FFH
 3EFF                   LD      A,FLAG
```

Programming Hints

Using symbolic names instead of numbers for constant values enhances the readability of a program and tends to make the code self-documenting. For instance, the symbol BUFLEN is a more descriptive name for a value than, say, 72. Furthermore, by using EQUates, if a value which is used throughout a program needs to be changed, only the EQU statement needs to be modified rather than finding all occurrences of the number 72.

It is generally preferable to use EQU for symbol definition, since the Assembler will generate error messages for multiply-defined symbols which may indicate spelling errors or some other oversight by the user. DEFL should be reserved for special cases where it is necessary to reuse the same symbol, for instance, in conjunction with conditional assembly.

A common use of EQUates is to symbolically represent offsets in a data structure. The following example demonstrates how EQUates might be used to reference certain fields in an I/O parameter vector as offsets from the IY index register:

```
IOLUN   EQU   0          ; logical unit designator
IOREQ   EQU   1          ; request code
IODTA   EQU   2          ; data transfer address
IODL    EQU   4          ; data length
OPEN    EQU   04H        ; open I/O device request code

LD (IY+IOREQ),OPEN       ; open the device
CALL SYSTEM
```

GLOBAL -- Global Symbol Declaration

The GLOBAL pseudo-op is used to declare that each of its operands are symbols defined in the module, and the name and value are made available to other modules which contain an EXTERNAL declaration for that name.

Label	Opcode	Operands
	GLOBAL	name1, name2,...

There must be no label. There may be zero or more names which are separated by delimiters and refer to labels which are defined elsewhere in the module. GLOBAL pseudo-ops may occur anywhere within the source text. Notice that the mode of a GLOBAL depends on how the label is defined, which, for labels on instructions or data values, depends on whether the module is relocatable or absolute.

A name may not be declared as both GLOBAL and EXTERNAL. If the name is never actually defined in the module (i.e., there is no corresponding label definition), an error message will be generated at the end of the listing.

Examples:

Label	Opcode	Operand
	GLOBAL	ENTRY
	GLOBAL	FEE,FI,FO,FUM

*Implementation Note

In order to distinguish between GLOBAL symbols which have been defined and those which have not, the Assembler maintains undefined GLOBAL symbols in the symbol table with the value OFFFFH. This implies that no GLOBAL symbol should be given this value, or it will be considered to be undefined.

EXTERNAL -- External Symbol Declaration

The EXTERNAL pseudo-op is used to declare that each of its operands are symbols defined in some other module but referenced in this module.

```
Label           Opcode          Operands

                EXTERNAL        name1,name2,...
```

There must be no label. There may be zero or more names which are separated by delimiters and refer to names which are defined in some other module. EXTERNAL pseudo-ops may occur anywhere within the source text. The EXTERNAL pseudo-op assigns each name an external mode, which allows the name to be used in arbitrary expressions elsewhere in the module, subject to the rules for external expressions.

A name may not be declared as both EXTERNAL and GLOBAL. If the name also appears as a label (i.e., the name is assigned a value within this module), an error message will be generated.

Examples:

```
Label           Opcode          Operands

                EXTERNAL        TTY
                EXTERNAL        SYSTEM,MEMTOP,MEMBOT
```

Sample Relocatable Program

This simple program is an incomplete specification of three separate modules -- a main routine and an input and output routine -- where each is maintained in a different source file and is assembled separately from the others. In addition, each source file contains an *INCLUDE of the same file of constants (EQUates) to maintain consistency throughout the program and avoid repetitious typing and editing.

File MAIN.S contains:

```
      GLOBAL  MAIN,BUFFER
      EXTERNAL  INPUT,OUTPUT
*INCLUDE EQUATES
FULL:   DEFB FALSE                ; LOCAL FLAG BYTE
BUFFER: DEFS 80                   ; GLOBAL DATA BUFFER
MAIN:
      LD A,(FULL)                 ; TEST BUFFER FLAG
      CP FALSE
      CALL Z,INPUT                ; FILL BUFFER IF EMPTY

      LD A,(FULL)                 ; TEST BUFFER FLAG
      CP TRUE
      CALL Z,OUTPUT               ; OUTPUT BUFFER IF FULL

      JP MAIN                     ; INFINITE LOOP
      END
```

File INDRIVER.S contains:

```
      GLOBAL  INPUT
      EXTERNAL  BUFFER
*INCLUDE EQUATES
INPUT:                            ; DRIVER ENTRY POINT
      LD  HL,BUFFER               ; ACCESS GLOBAL DATA BUFFER

      RET
      END
```

File OUTDRIVER.S contains:

```
      GLOBAL  OUTPUT
      EXTERNAL  BUFFER
*INCLUDE EQUATES
OUTPUT:                           ; DRIVER ENTRY POINT
      LD  HL,BUFFER               ; ACCESS GLOBAL DATA BUFFER

      RET
      END
```

File EQUATES.S contains:

```
TRUE  EQU     OFFH
FALSE EQU     0
```

Reference Counter Control

ORG -- Origin

The ORG pseudo-op is used to set the reference counter to the value of an expression. The reference counter serves the same function for the Assembler as the program counter does for the CPU in specifying where object code is located.

Label	Opcode	Operand
optional:	ORG	expression

The reference counter is set to the 16-bit value of the expression, so that the next machine instruction or data item will be located at the specified address. Any symbol appearing in the expression must have been defined before the Assembler encounters the expression. The reference counter is initially set to zero, so if no ORG statement precedes the first instruction or data byte in the module, that byte will be assembled at location zero. If a label is present, it will be assigned the same value as the expression. A module may contain any number of ORG statements.

The mode of the expression in an ORG pseudo-op must not be external and depends on the relocatability of the module. If a module is absolute, the ORG pseudo-op serves to assign an absolute address to both the reference counter and the label. The expression will be treated as relocatable in a relocatable module, since the effect is to change the relocatable reference counter. Thus the label on an ORG statement in a relocatable module will have a relocatable mode. For example, the effect of the statement

Label	Opcode	Operand
LAB:	ORG	100

within a relocatable module would be to set the reference counter to "relocatable 100", assign the label LAB the value 100, and give it a relocatable mode.

Relocatable modules do not generally contain ORG statements, since the pseudo-op is useful only to reserve space within the module (in a manner similar to the DEFS pseudo-op), and not to assign an absolute location to a section of code since this would defeat the purpose of relocation. Since modules are by default relocatable, a warning message will be printed if an ORG pseudo-op is encountered in a relocatable module to indicate to the user that he may have forgotten to specify the Absolute option

in the Assembler command line. This warning message may be suppressed by the NOW option.

Some applications require that a section of code must start on a particular address boundary, for instance, at an address which is some multiple of a memory page size (say, 256 bytes per page). A useful formula for incrementing the reference counter to the next page boundary is N*(($+N-1)/N) where N is the page size. For example, suppose the page size is 8 and it is required that an interrupt vector table INTVEC be located at the next page boundary following the current value of the reference counter. Then the following statement achieves this effect regardless of the current value of the reference counter.

```
Label           Opcode          Operand

INTVEC:          ORG            8*(($+7)/8)
```

Notice that this formula is useful only in an absolute module, since multiplication and division are undefined for relocatable operands (the reference counter symbol $ is relocatable in a relocatable module).

*Implementation Note

In order to assist the Linker in assigning origins to relocatable modules, the Assembler computes the size of a module so that other modules can be relocated immediately following it. This length is computed as the value of the reference counter at the end of assembly minus the origin of the module. The origin of a relocatable module is relocatable zero, while the origin of an absolute module is the lowest address in the module which contains object code. In either case, if a program contains an ORG statement which sets the reference counter to a value less than it was previously, unexpected results may occur. In particular, an absolute module whose reference counter at the end of assembly is less than the lowest address which contains object code will cause an erroneous length to be computed. The programmer is advised that, in general, the reference counter should be incremented in a monotonic fashion so that addresses are in ascending order.

Conditional Assembly

Conditional assembly allows the programmer to inhibit the assembly of portions of the source text provided certain conditions are satisfied. Conditional assembly is particularly useful when a program needs to contain similar code sequences for slightly different applications. Rather

than generating a multitude of programs to handle each application, the application-dependent sections of code can be enclosed by the conditional pseudo-ops within a single program. Then by changing the values of several symbols used to control the conditional assembly, the user can generate different object modules from subsequent assemblies of the same source.

Another common use of conditional assembly is in conjunction with macros to control generation of code dependent on the value of parameters (see Macros).

COND and ENDC -- Conditional Assembly

The pseudo-op COND is used to test the value of an operand and, depending on the result, inhibit assembly of subsequent statements until an ENDC pseudo-op is encountered.

Label	Opcode	Operand
optional:	COND	expression
optional:	ENDC	

The Assembler evaluates the expression in the operand field of the COND pseudo-op. If the 16-bit result is true (non-zero), the COND pseudo-op is ignored and subsequent statements are assembled normally. If the result is false (zero), the assembly of subsequent statements is inhibited until an ENDC pseudo-op is encountered. The mode of the expression can be either relocatable or absolute; however, it may not be external. There is no operand for the ENDC pseudo-op.

Notice that the definition of symbols within a conditional assembly block may be inhibited, and thus references to these symbols elsewhere in the module may cause undefined symbol errors. In particular, the label on an ENDC pseudo-op will not be defined if the assembly is inhibited when the pseudo-op is encountered.

Conditional assembly blocks cannot be nested, i.e., the occurrence of an ENDC causes resumption of assembly of subsequent statements. Therefore, each COND does not require a matching ENDC.

The user must be cautious of nesting conditional blocks within macro definitions (see Macros) and vice-versa. If a macro definition is only partially completed because a surrounding conditional assembly block has inhibited the definition, or conversely, if a macro contains an unmatched COND/ENDC, unexpected results may occur.

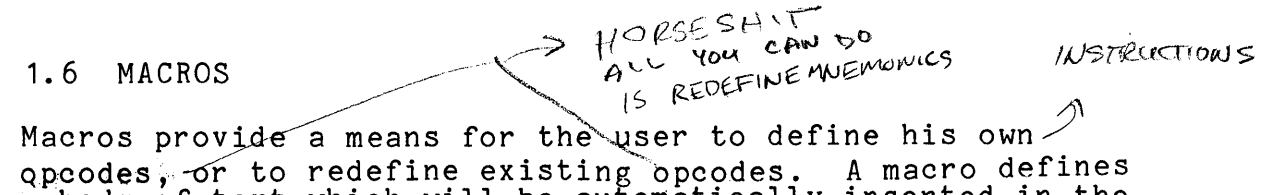
Examples:

```
Label       Opcode      Operand

            COND        FLAG
                        ; these statements are
                        ; assembled only if the
                        ; value of FLAG is not
                        ; equal to zero
            ENDC
```

(see Macros for further examples)

1.6 MACROS



Macros provide a means for the user to define his own opcodes, or to redefine existing opcodes. A macro defines a body of text which will be automatically inserted in the source stream at each occurrence of a macro call. In addition, parameters provide a capability for making limited changes in the macro at each call. If a macro is used to redefine an existing opcode, a warning message is generated to indicate that future use of that opcode will always be processed as a macro call. If a program uses macros, then the assembly option M must be specified (described later).

Macro Definition

The body of text to be used as a macro is given in the macro definition. Each definition begins with a Macro pseudo-op and ends with an ENDM pseudo-op. The general forms are:

Label	Opcode	Operands
name:	MACRO	#P0,#P1,...,#Pn
optional:	ENDM	

The name is required, and must obey all the usual rules for forming labels (remember the colon is optional if the name starts in column one).

There can be any number of parameters (including none at all), each starting with the symbol #. The rest of the parameter name can be any string not containing a delimiter (blank, comma, tab, semicolon) or the symbol #. However, parameters will be scanned left to right for a match, so the user is cautioned not to use parameter names which are prefix substrings of later parameter names. Parameter names are not entered in the symbol table.

The label on an ENDM is optional, but if one is given it must obey all the usual rules for forming labels.

Each statement between the MACRO and ENDM statements is entered into a temporary macro file. The only restriction on these statements is that they do not include another macro definition (nested definitions are not allowed). They may include macro calls. The depth of nested calls is limited only by the available buffer space in the macro parameter stack. The amount of buffer space required by a sequence of nested calls is dependent on the number and length of parameters, which in general allows calls nested to a depth of about 15.

The statements of the macro body are not assembled at definition time, so they will not define labels, generate code, or cause errors. Exceptions are the assembler commands such as *List, which are executed wherever they occur. Within the macro body text, the formal parameter names may occur anywhere that an expansion-time substitution is desired. This includes within comments and quoted strings. The symbol # may not occur except as the first symbol of a parameter name.

Macros must be defined before they are called.

Macro Calls and Macro Expansion

A macro is called by using its name as an opcode at any point after the definition. The general form is:

```
label:  name 'S1', 'S2',...'Sn'
```

The label is optional and will be assigned to the current value of the reference counter, while the name must be a previously-defined macro. There may be any number of argument strings, Si, separated by any number of blanks, tabs, or commas. Commas do not serve as parameter place holders, only as string delimiters. If there are too few parameters, the missing ones are assumed to be null. If there are too many, the extras are ignored. The position of each string in the list corresponds with the position of the macro parameter name it is to replace. Thus the third string in a macro call statement will be substituted for each occurrence of the third parameter name.

The strings may be of any length and may contain any characters. The outer level quotes around the string are generally optional, but are required if the string contains delimiters or the quote character itself. The quote character is represented by two successive quote marks at the inner level. The outer level quotes, if present, will not occur in the substitution. The null string, represented by two successive quote marks at the outer level, may be used in any parameter position.

After processing the macro call statement, the Assembler switches its input from the source file to the macro file. Each statement of the macro body is scanned for occurrences of parameter names, and for each occurrence found, the corresponding string from the macro call statement is substituted. After substitution, the statement is assembled normally.

Symbol Generator

Every macro definition has an implicit parameter named #$YM. This may be referenced by the user in the macro body, but should not explicitly appear in the MACRO statement. At expansion time, each occurrence of #$YM in the definition is replaced by a string representing a 4-digit hexadecimal constant.

This string is constant over a given level of macro expansion, but increases by one for each new macro call. The most common use of #$YM is to provide unique labels for different expansions of the same macro. Otherwise, a macro containing a label would cause multiple definition errors if it were called more than once. Notice that a generated label must start with an alphabetic character just like all other labels.

The following is a simple macro which fills a buffer specified by the parameter #BUFFER with the number of blanks specified by #COUNT. The use of #$YM is necessary to create a unique label for each call to the macro.

```
FILLBL    MACRO #BUFFER #COUNT
          LD    B,#COUNT          ; number of bytes
          LD   HL,#BUFFER         ; address of buffer
FL#$YM    LD   (HL), ' '          ; insert blank
          INC HL                  ; advance pointer
          DJNZ FL#$YM             ; loop until done
          ENDM
```

If this is the first macro call, it would generate the following expansion when called with parameters INBUF and 50:

```
          FILLBL INBUF 50
          LD B,50
          LD HL,INBUF
FL0000:   LD (HL),' '
          INC HL
          DJNZ FL0000
          ENDM
```

The next call would cause the label FL0001 to be generated, and so on.

Recursion

Macros may include calls to other macros, including themselves. A macro which directly calls itself (or indirectly by calling a second macro which calls the first

macro) is said to be recursive. Each recursive call causes a new expansion of the macro, possibly with different parameters. In order to prevent the macro being called endlessly, conditional assembly is used to inhibit a recursive call when certain conditions are satisfied (see the following examples). A macro which calls itself more times than the macro parameter stack can accommodate will generate an error. This often indicates that the recursion halting conditions would never be satisfied and the macro would be endlessly calling itself.

Listing Format

By default, each expanded statement is listed with a blank STMT field to differentiate macro expansions from normal source statements. If the Maclist flag is turned off by the NOM option or *M OFF, then only the macro call is listed.

Examples:

The following example, MOVE, is a macro which is used to move (copy) a block of memory. The source address, destination address, and number of bytes are all specified by parameters to the macro call.

The macro also provides the option to save the registers before the move and restore them upon completion. This option is implemented through the use of conditional assembly and the fact that any missing parameters are assumed to be null. Thus, if a non-null parameter is substituted for #SAVE?, the true condition of the COND statement is met and the code to save and restore the registers is assembled. If no parameter is passed, the #SAVE? parameter will be equal to the null string and the code will be ignored.

```
 1 ; MOVE MACRO--BLOCK MOVE OF #LENGTH BYTES
 2 ;              FROM #SOURCE TO #DEST
 3 ;              IF 4TH PARAMETER IS PRESENT,
 4 ;              THEN REGISTERS ARE PRESERVED
 5 ;
 6 MOVE  MACRO #DEST #SOURCE #LENGTH #SAVE?
 7       COND .NOT.('#SAVE?'='')
 8       PUSH BC
 9       PUSH DE
10       PUSH HL
11       ENDC
12       LD   BC,#LENGTH  ;MOVE #LENGTH BYTES
13       LD   DE,#DEST
14       LD   HL,#SOURCE
15       LDIR
16       COND .NOT.('#SAVE?'='')
17       POP  HL
18       POP  DE
19       POP  BC
20       ENDC
21       ENDM
```

If the MOVE macro is called as follows:

```
MOVE OUTBUF INBUF 25 S
```

code would be generated to save the registers BC, DE and HL, to move 25 bytes from INBUF to OUTBUF and then to restore registers BC, DE and HL.

```
                22 ; A SAMPLE USE OF 'MOVE' WHICH PRESERVES
                23 ; REGISTERS
                24   MOVE OUTBUF INBUF 25 S
                          COND .NOT.('S'='')
0000   C5                 PUSH BC
0001   D5                 PUSH DE
0002   E5                 PUSH HL
                          ENDC
0003   011900             LD   BC,25  ;MOVE 25 BYTES
0006   111C00             LD   DE,OUTBUF
0009   213500             LD   HL,INBUF
000C   EDB0               LDIR
                          COND .NOT.('S'='')
000E   E1                 POP  HL
000F   D1                 POP  DE
0010   C1                 POP  BC
                          ENDC
```

If the last parameter were omitted and the call was made as follows:

```
MOVE OUTBUF INBUF 25
```

the code to preserve the registers would not be generated because the #SAVE? parameter is the null string.

```
                    25 ; THIS 'MOVE' DOES NOT SAVE REGISTER
                    26 ; CONTENTS
                    27   MOVE OUTBUF INBUF 25
                              COND .NOT.(''='')
                              PUSH BC
                              PUSH DE
                              PUSH HL
                              ENDC
0011    011900               LD   BC,25  ;MOVE 25 BYTES
0014    111C00               LD   DE,OUTBUF
0017    213500               LD   HL,INBUF
001A    EDB0                 LDIR
                             COND .NOT.(''='')
                             POP  HL
                             POP  DE
                             POP  BC
                             ENDC
                    28
001C                29 OUTBUF: DEFS 25
0035                30 INBUF:  DEFS 25
```

The next example demonstrates the use of recursion, as well as the use of the DEFL pseudo-op to redefine the value of a label to be used as a counter to control the recursion.

When DUP is called, it initializes the label to the value passed to #CNT and then calls DUP1 with the parameter that was passed to #OP.

The DUP1 macro decrements the value of the label and recursively calls itself until the label has a value of zero.

The second call to DUP demonstrates the use of the *M OFF and *M ON assember commands to suppress the listing of the expansion of the macro (see Assembler commands description).

```
                          1 ; DUPLICATE MACRO -- DUPLICATES #OP FOR #CNT
                          2 ; LINES USES DLAB LABEL AS RECURSION FACTOR
                          3 DUP  MACRO #OP #CNT
                          4 DLAB DEFL #CNT
                          5      DUP1 '#OP'
                          6      ENDM
                          7
                          8 ; DUP1 RECURSIVE MACRO -- OUTPUTS #OP EACH TI
                          9 ; RECURS ON THE VALUE OF THE LABEL DLAB
                         10 DUP1 MACRO #OP
                         11      COND DLAB>0
                         12 DLAB DEFL DLAB-1
                         13      DUP1 '#OP'
                         14      #OP
                         15      ENDC
                         16      ENDM
                         17
                         18 ; ROTATE RIGHT ACCUMULATOR 3 TIMES
                         19           DUP RRA 3
                            DLAB DEFL 3
                                 DUP1 'RRA'
                                 COND DLAB>0
                            DLAB DEFL DLAB-1
                                 DUP1 'RRA'
                                 COND DLAB>0
                            DLAB DEFL DLAB-1
                                 DUP1 'RRA'
                                 COND DLAB>0
                            DLAB DEFL DLAB-1
                                 DUP1 'RRA'
                                 COND DLAB>0
                            DLAB DEFL DLAB-1
                                 DUP1 'RRA'
                                 RRA
                                 ENDC
0000    1F                       RRA
                                 ENDC
0001    1F                       RRA
                                 ENDC
0002    1F                       RRA
                                 ENDC
                         20 *M OFF
0003                     21           DUP 'INC HL' 4 ;INCREMENT HL BY 4
                         22 *M ON
```

The last example, DB, is a macro which accepts up to 10 parameters, and issues a DEFB pseudo-op for each value.

The macro issues a DEFB for its first parameter and then calls itself recursively with the same parameters it received minus the first. Thus with each call the parameters are shifted to the left by one, and any missing parameters become the null string. After all of the values have been processed, the call to DB will pass the null string as parameter #1 and the condition of the COND statement will not be met. Notice how the outer level of single quotes is stripped from the parameter strings.

```
                        25 ; DB MACRO -- ACCEPTS UP TO 10 VALUES ON SAME L
                        26 ;  OUTPUTS A DEFB FOR EACH ONE -- SELF-RECURSIV
                        27 DB     MACRO #0 #1 #2 #3 #4 #5 #6 #7 #8 #9
                        28        DEFB #0
                        29        COND '#1'
                        30     DB '#1' '#2' '#3' '#4' '#5' '#6' '#7' '#8' '
                        31        ENDC
                        32        ENDM
                        33
                        34             DB 1,2,3,4,5,6,7,8,9,10
0007    01                        DEFB 1
                                  COND '2'
                               DB '2' '3' '4' '5' '6' '7' '8' '9' '10'
0008    02                        DEFB 2
                                  COND '3'
                               DB '3' '4' '5' '6' '7' '8' '9' '10' ''
0009    03                        DEFB 3
                                  COND '4'
                               DB '4' '5' '6' '7' '8' '9' '10' '' ''
000A    04                        DEFB 4
                                  COND '5'
                               DB '5' '6' '7' '8' '9' '10' '' '' ''
000B    05                        DEFB 5
                                  COND '6'
                               DB '6' '7' '8' '9' '10' '' '' '' ''
000C    06                        DEFB 6
                                  COND '7'
                               DB '7' '8' '9' '10' '' '' '' '' ''
000D    07                        DEFB 7
                                  COND '8'
                               DB '8' '9' '10' '' '' '' '' '' ''
000E    08                        DEFB 8
                                  COND '9'
                               DB '9' '10' '' '' '' '' '' '' ''
000F    09                        DEFB 9
                                  COND '10'
                               DB '10' '' '' '' '' '' '' '' ''
0010    0A                        DEFB 10
                                  COND ''
                               DB '' '' '' '' '' '' '' '' ''
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
                                  ENDC
```

```
                    36              DB LAB,LAB*3+1,TRUE,FALSE,OFFH
0011    02                  DEFB LAB
                            COND 'LAB*3+1'
                        DB 'LAB*3+1' 'TRUE' 'FALSE' 'OFFH' '' '' '' '' ''
0012    07                  DEFB LAB*3+1
                            COND 'TRUE'
                        DB 'TRUE' 'FALSE' 'OFFH' '' '' '' '' '' ''
0013    FF                  DEFB TRUE
                            COND 'FALSE'
                        DB 'FALSE' 'OFFH' '' '' '' '' '' '' ''
0014    00                  DEFB FALSE
                            COND 'OFFH'
                        DB 'OFFH' '' '' '' '' '' '' '' ''
0015    FF                  DEFB OFFH
                            COND ''
                        DB '' '' '' '' '' '' '' '' ''
                            ENDC
                            ENDC
                            ENDC
                            ENDC
                            ENDC
                    37 *M OFF
0016                38              DB 0 0 0 0 0 0 0 0 0
                    39 *M ON
                    40 LAB:  EQU 2
                    41 TRUE: EQU OFFH
                    42 FALSE:EQU 0
```

1.7 ASSEMBLER COMMANDS

The Assembler recognizes several commands to modify the listing format, and one command to allow other source files to be included at that point within the source program. An assembler command is a line of the source file beginning with an * in column one. The character in column two identifies the type of command. Arguments, if any, are separated from the command by any number of blanks, tabs, or commas. The following commands are recognized by the Assembler:

*Eject — Causes the listing to advance to a new page starting with this line. This is accomplished by outputting an ASCII form feed character followed by a carriage return character.

*Heading s — Causes string s to be taken as a heading to be printed at the top of each new page. String s may be any string of 1 to 28 characters not containing leading delimiters (notice that the string is not bounded by quote marks). This command does an automatic Eject.

*List OFF — Causes listing and printing to be suspended starting with this line.

*List ON — Causes listing and printing to resume, starting with this line.

*Maclist OFF — Causes listing and printing of macro expansions to be suspended, starting with this line. Only macro calls will be in the listing.

*Maclist ON — Causes listing and printing of macro expansion to resume, starting with this line.

*Include filename

Causes the source file "filename" to be included in the source stream following the command statement.

The expected use of *Include is for files of macro definitions, lists of EXTERNAL declarations, lists of EQUates, or commonly used subroutines, although it can be used anywhere in a program where the other commands would be legal. The filename must follow the normal convention for specifying source filenames as indicated in the

Assembler command line options description. The included file may also contain an *Include command, up to a nested level of four.

If the *Include command appears within a macro definition, the Assembler will always try to shoe-horn the file inside the macro definition, and although the *Include statement will appear in a macro expansion, the file will not be included again at the point of expansion. *Include works in the expected manner in conjunction with conditional assembly. For example,

```
            COND    exp
*Include    FILE1
            ENDC
;FILE1 is included only if the value of exp is non-zero.
```

1.8 ASSEMBLER COMMAND LINE OPTIONS

The Assembler interacts with the RIO operating system to permit the user complete control over the routing of all input and output (I/O) during assembly. The Assembler makes use of both the console device and disk files for I/O. The reader may wish to refer to the RIO Operating System User's Manual mentioned in the preface for a more complete description of files and I/O handling.

Using the Assembler

The Assembler is invoked from the RIO Executive by a command in the general form:

```
ASM  file_name*  [(options)]
```

where * means that one or more file names may be specified in the standard RIO format, and the square brackets mean that the options are optional. However, if present, they must be enclosed by parentheses. The Assembler processes the given source files in the order specified to produce a single object module and a listing file. It is not necessary to specify options to control assembly since defaults exist, but when included they must be enclosed by parentheses. The command is terminated by either a return or a semicolon.

When invoked, the Assembler first identifies itself by printing on the console its name and version number, where n.n is the current version specification:

```
ASM n.n
```

The Assembler processes the source program in two passes. The first pass identifies instructions, tests for correctness of operand combinations, defines the value of symbols and writes an intermediate file to the disk. At the conclusion of this process, a message is printed on the console:

```
PASS 1 COMPLETE
```

During the second assembly pass, listing and object files are created as output. The listing file contains the source language statements in their original format as well as the corresponding machine language and memory addresses. The listing file may optionally include a symbol cross-reference. The listing file is an ASCII type file with 128 bytes per record.

The object module consists of a binary type file with 128 bytes per record which is meaningful only to the Linker. It consists of four basic parts: a Header, the External Symbol Dictionary (ESD), Code blocks, and an optional Internal Symbol Dictionary (ISD). The Header contains information describing the module such as the number of ESD entries and the number of bytes of code. The ESD consists of all the GLOBAL and EXTERNAL names and possibly some indication of where the names are defined in the following Code blocks. The Code blocks contain both the actual machine code, as indicated in the listing, as well as a bit-map which indicates which instructions are to be relocated or linked to EXTERNAL names. The ISD is created only if the Symbol option in the command line is specified. The ISD contains all the symbols which are internal to the module (i.e., not GLOBAL or EXTERNAL) and is used by the Linker to create a symbol table file for use by a symbolic debug package. See the Appendix for a complete description of an object module.

Both files are created by default on the same disk drive with the first source file. If files already exist with the same name, they are erased and replaced with the newly generated files.

Errors in the source program detected during assembly are indicated by appropriate messages following the specific line both on the console and in the listing file (see the Appendix). In addition, the number of errors is indicated on the console at the end of the second pass by the following message where n is a decimal number:

```
n ASSEMBLY ERRORS
```

Normally, the assembly is complete after two passes. However, if either the cross-reference or the symbol option are specified, the Assembler starts a third pass by printing the following message:

```
PASS 2 COMPLETE
```

At the conclusion of the assembly, a message is printed on the console:

```
ASSEMBLY COMPLETE
```

Both the second and third pass are implemented as overlays to a portion of the first pass. The overlays are in procedure type files named ASM2 and ASM3, respectively. If these files cannot be found when required, the Assembler will abort.

At any point during the assembly, entering a '?' character will cause the Assembler to pause until another '?' character is entered. This allows the user to scan a portion of output while it is generated. In addition, if the ESCape character (ASCII 1BH) is entered, the Assembler will close all files immediately and abort the assembly. Notice that if the assembly is aborted, output files may contain incomplete information.

Several temporary scratch files are created by the Assembler, but are erased upon completion of an assembly. They include, depending on the options specified, the intermediate file, macro file, cross-reference file, and a symbol table overflow file. The user can control the assignment of these files if necessary (see below).

The Assembler uses a buffered I/O technique for handling the assembly language source file, listing file, object file and temporary files. The Assembler automatically determines the available work space and allocates the buffer sizes accordingly. Hence there are no constraints on the size of the assembly language source file that can be assembled, provided there is adequate space for the files.

Module Identification

The Assembler follows the general principle that anywhere a file name may be specified, that file name may be completely or partially qualified by either a device driver name or a drive specification.

For example:

```
1) ASM PROG1.S
2) ASM $MYDOS:2/multiply
```

The first example demonstrates the use of an unqualified file name, PROG1.S, which can be found on the master device. If that device is ZDOS, then the standard disk drive search sequence of drive 1,2,...0 will be used. The second example demonstrates a fully qualified file name, multiply, which can be found on drive 2 of the device MYDOS.

Each source file must be ASCII type with 128 bytes per record.

When the period character is used to separate a file name into several parts, those parts are referred to as file name "extensions". For instance, the file name PROG1.S in the above example has the extension .S. The notion of file name extensions is a useful convention for the user who

wishes to categorize certain files by their names. To assist in this categorization, and to provide a uniform method for naming generated files such as the associated object and listing files, the Assembler insists that all source file names must end in the extension .S (for source). Notice there may be several extensions, as long as the last one is .S, and either upper or lower case is acceptable. Furthermore, the extension does not have to be typed in the command line, so that if the last two characters are not .S, the Assembler will automatically append the extension before attempting to open the file.

By default, the object module and list file will be given the name of the first source file which comprises the module, as indicated in the command line, but with the extensions .OBJ and .L, respectively. The case of the extension is the same as the first character of the file name.

For example:

1) ASM YOURPROG
   looks for source YOURPROG.S, creates YOURPROG.OBJ and YOURPROG.L

2) ASM myprog1.s myprog2 TTY.INT
   looks for sources myprog1.s, myprog2.s, and TTY.INT.S, creates myprog1.obj and myprog1.l

It is possible to override the name of either the object or listing file by appropriate options in the command line as described below.

Options

Several options can be used to modify the ASM command; however, the list of options must be enclosed in parentheses following the source file names. The options may be specified in any order, with capital letters in the following list signifying the minimum allowable abbreviation. Notice that either upper or lower case can be used.

In order to maximize the effectiveness of allocation of file buffers, and thus reduce I/O activity, certain processing must be explicitly requested:

Macro	Enables macro processing. If not specified, macro definitions and calls will be flagged as errors.

Symbol	Causes a third pass to be executed, producing a binary representation of the internal symbols (i.e., not GLOBAL or EXTERNAL) to be appended to the object module. This information may be used by a symbolic debug package.
Xref	Causes a third pass to be executed, producing a sorted cross-reference table at the end of the list file.

The following options will override default settings:

Absolute	Causes the entire module to be assembled with absolute addresses. Default is relocatable.
NOList	Suppresses the creation of the listing file.
NOMaclist	Suppresses the listing and printing of macro expansions.
NOObject	Suppresses the creation of the object file.
NOWarning	Suppresses the generation of opcode redefinition warning messages, and relocatable ORG warning messages.

The following option aids documentation:

D=string	Places up to an 18-character string, not including blanks, tabs, commas, or semicolons, in the heading of each page of the listing (normally used to specify a date).

Note the following interactions:

NOL overrides *List and *Maclist commands in the text.

*Maclist commands in the text override NOM.

X will override NOL for the third pass only. That is, (NOL X) will produce a list file containing only the cross-reference listing.

NOO overrides S.

I/O Routing Options

The Assembler uses several logical I/O units, some of which depend on the requirements of the module being assembled. The user can control the definition of these units either through interaction with the RIO Executive before giving the ASM command, or by specifying certain options in the command line. Pass completion notices and all fatal error messages which cause the Assembler to abort (see Fatal Error Messages) are always routed to the console (logical unit CONOUT). Normally, errors in the source statements will also be routed to the console, unless overridden by the Print option.

Print	Causes the listing output as it is generated to be routed to the logical unit SYSLST, in addition to being directed to a listing file. This is particularly useful if SYSLST is defined to be a line printer driver, for instance. The device driver must recognize the WRITE ASCII request code to operate properly. The Print option inhibits source statement errors from being sent to the console to avoid duplicate messages when SYSLST and CONOUT are both assigned to the same device.

The following options will override the default definition of output and temporary files used by the Assembler.

file_name can be a completely or partially qualified file name.

I=file_name	defines intermediate file
L=file_name	defines listing file
M=file_name	defines macro file
O=file_name	defines object file
T=file_name	defines symbol table overflow file
X=file_name	defines cross-reference file

The symbol table overflow file is created automatically but only if necessary. M= and X= imply the Macro and Xref options, respectively.

Notice that giving a complete file name to a file which would normally be a temporary scratch file will cause that file to remain after assembly. For instance, I=3 causes the intermediate file to be assigned to a scratch file (zero-length name) on drive 3 of the master device, whereas I=3/I.TMP causes the intermediate file to remain after assembly as the file I.TMP on drive 3 of the master device.

Examples:

1) ASM PROG1
   produces PROG1.OBJ and PROG1.L on the master device and same drive as the source program, PROG1.S, will use the same drive for both the intermediate file and the symbol table overflow file if necessary, and will not print the listing on SYSLST.

2) ASM PROG1 (O=$MYDOS:0/A.OUT S X=2 P NOL)
   produces an object file with attached symbol table named A.OUT on drive 0 of device $MYDOS, prints the listing with a cross-reference on SYSLST, uses drive 2 of the master device for the temporary cross-reference file, and produces no listing file. The intermediate file and symbol table overflow file will be created as scratch files on the same device and drive as the source, PROG1.S.

Logical Unit Definitions

The Assembler uses the following logical I/O units:

2	CONOUT	Error messages
3	SYSLST	Print option output
4		List file
5		Object file
6		Intermediate file
7		Macro file
8		Xref file
9		Symbol table overflow file
10-13		Source files (multiple units used to support nested *Include commands)

Fatal Error Messages

There are several conditions that may occur which will cause the Assembler to terminate further processing, close all files, and abort by returning to the RIO Executive. The following messages may appear on the console:

INVALID OPTION: s

Indicates that the string s was not recognized as a valid command line option. The cause may be, for instance, a misspelled option specifier or invalid use of delimiters within options.

MEMORY TOO SMALL

Indicates that the amount of available buffer space is not adequate to permit assembly. Buffers are allocated during each pass depending on the various options requested in the command line.

LINE TOO LONG: filename

Indicates that a source text line in "filename" was encountered that was longer than 128 characters. Due to buffer allocation, longer lines are not acceptable. This error may indicate that the contents of the source file have been damaged.

FILE NOT FOUND: filename

Indicates that the source "filename" could not be found in a device directory. Remember that the source file name must end in the extension .S.

INVALID ATTRIBUTES: filename

Indicates that the source "filename" had attributes other than type ASCII (subtype is ignored) and a record length of 128 bytes.

INVALID FILE: filename

Indicates that the string "filename" is not a proper file name specifier. This may include illegal characters within the name or an invalid device or drive designator (see RIO Operating System User's Manual for further details). The string "filename" is printed as it appeared in the command line.

I/O ERROR e ON UNIT u

All other fatal errors involve I/O errors (hex value e) on a particular logical unit (decimal value u). Refer to the Logical Unit Definitions for the Assembler and the RIO Operating System User's Manual for an explanation of specific errors.

2.0 LINKER

2.1 FUNCTION

The function of the Linker is to process one or more object modules created by the Assembler (version 4.0 or greater), or any other language translator which generates relocatable object modules, and output a single program in the form of an executable procedure file. The Linker provides relocation of modules and resolves inter-module references to allow linking of separately assembled modules. In addition, a load map file and a binary symbol table file may be created.

The Linker interacts with the RIO operating system to permit the user complete control over the routing of all input and output (I/O) during linkage. The Linker makes use of both the console device and disk files for I/O. The reader may wish to refer to the RIO Operating System User's Manual mentioned in the preface for a more complete description of files and I/O handling.

Features

1) Creates a segmented procedure type file of either 128, 256, 512, or 1024 bytes per record. Code may be assigned to any location in memory, and up to 16 different segments of code can be generated which allows for discontiguous sections of memory which are not overlaid when the procedure file is loaded.

2) Provides ability to link both absolute and relocatable modules together.

3) Allows optional assignment of some or all relocatable modules to specific absolute origins.

4) Permits specification of the execution starting address as either a hexadecimal number or a GLOBAL symbol.

5) Checks for multiply-defined GLOBALs and unresolved EXTERNALs.

6) Provides the capability to specify "link-only" modules, that is, modules which are used to control assignment of relocatable module origins and resolve EXTERNAL references, but are not included as part of the generated procedure file.

7) Creates a load map with the assigned origin and length of each module, followed by an alphabetically sorted list of all GLOBALs, their assigned addresses and the module which contains them. The map file also contains a list of any error messages generated during linkage.

8) Allows optional creation of a binary symbol table file, structured by module, for use by a symbolic debug package.

9) Provides user control over the routing of all input and output during linkage.

2.2 USING THE LINKER

The linker is invoked from the RIO Executive by issuing the command LINK followed by a list of module names and options as described below. The general form is:

```
LINK  file_name*  [(options)]
```

where * means that one or more file names may be specified in the standard RIO format, and the square brackets mean that the options are optional; however, if present they must be enclosed by parentheses. The command is terminated by either a return or a semicolon.

When invoked, the Linker first identifies itself by printing on the console its name and version number, where n.n is the current version specification:

```
LINK n.n
```

There are two distinct passes to the Linker. During the first pass the Linker must build what is called the Link Directory, which allows it to assign absolute addresses to modules and GLOBAL names. EXTERNAL names must be partially resolved -- i.e., assigned the absolute address of the corresponding GLOBAL name. Modules are normally assigned ascending addresses in the order they are specified.

During the second pass, the object code of each module is processed, local references are relocated, EXTERNAL references are completely resolved, and both a procedure file and a load map file are generated. An optional third pass will occur if a symbol file is to be generated. At the end of linkage, if there were any non-fatal errors, an error count is printed on the console:

```
n ERRORS
```

where n is the total number of multiply-defined GLOBALs and unresolved EXTERNALs. Other messages are considered only warning messages.

The final message on the console is:

```
LINK COMPLETE
```

At any time during linkage, the '?' character can be entered, and the Linker will pause until another '?' character is entered. If the ESCape character is entered, the Linker will close all files immediately and abort by returning to the RIO Executive. If the Linker is aborted, output files may contain incomplete information.

Non-fatal Errors

There are two non-fatal error conditions which can occur during the first phase of linking. Multiply-defined GLOBAL names will be listed and should be considered a significant error. Unresolved EXTERNALs will be listed but may not indicate an error, since the module may not have actually used the EXTERNAL name in a reference, for instance, during the initial checkout of a program. These errors will cause the Linker to suppress creation of a procedure file unless the LET option is specified (see Options).

If either of these errors exist, an appropriate message is output to both the console and map file, followed by a list of the symbol names and the appropriate object module which caused the error:

```
MULTIPLY-DEFINED GLOBAL IN MODULE:
symbol_name    module_name
```

This message indicates that symbol_name was defined as a GLOBAL in two different modules, where module_name contains the second definition. If a load map is generated, the user can determine the other module from the alphabetic list of GLOBALs. References to a multiply-defined GLOBAL will always refer to the first definition.

```
UNRESOLVED EXTERNAL IN MODULE:
symbol_name    module_name
```

This message indicates that symbol_name which was declared to be EXTERNAL to module_name could not be matched to a corresponding GLOBAL name. If a procedure file is created with unresolved EXTERNAL references, those references will contain unpredictable addresses.

Assignment of Module Origins

The Linker maintains a reference counter which is similar to the CPU program counter and is used to assign sections of machine code to memory locations. By default, relocatable modules are located in contiguous memory areas automatically by assigning the current value of the reference counter to be the module's origin and then incrementing the reference counter by the module's length. The reference counter is always initialized to zero.

In order to control the assignment of relocatable modules to certain absolute addresses, it is possible to intersperse assignments to the reference counter between module names. The form for these assignments is either:

1) $=X
   X is a hexadecimal number (no trailing 'H')

2) $=$+X

where the first allows direct assignment to the reference counter which causes the next relocatable module to have that address as its origin. The second form allows the reference counter to be incremented by an appropriate amount, say to allocate some buffer space between modules. For example, to link three modules together with an origin of 4000 and a data buffer module at 9000:

```
LINK   $=4000   MODA   MODB   MODC   $=9000   DATA
```

MODA, MODB and MODC will follow immediately after each other in memory starting at location 4000.

If the reference counter assignment is poorly formed (embedded delimiters or an invalid hexadecimal number) an invalid option message will be issued and the Linker will abort. Notice that absolute modules may be linked to relocatable modules or to each other. Absolute modules contain their own address information, so it makes little sense to use $ when linking absolute modules. The reference counter is reset to the lowest address of the absolute module which contains object code and then incremented by the length of the module. Both the lowest address and the length of the module are calculated by the language translator which created the object module.

Segment Generation

Often the need arises in a program to separate it into several non-contiguous segments of machine code due either

to certain hardware requirements--such as the application memory configuration or peripheral device interface restrictions--or certain software requirements--such as the need to implement an overlay scheme. The Linker provides an automatic facility for generating up to 16 non-contiguous memory segments, each of which consists of an arbitrary number of bytes located at an arbitrary starting point. Each segment contains a number of bytes which is always an integral multiple of the procedure file's record length (default of 128 bytes--see Record Length option).

When the resulting procedure file is loaded, only those areas of memory occupied by the segments will be modified. The Linker uses two strategies in determining when to start generating a new segment:

1) A new segment is started for machine code which is to be located at an address which is less than the previous value of the reference counter. This may occur as the result of an ORG statement in assembly language or from the assignment of the Linker's reference counter using the $=X construct. The Linker will output the following message:

   POSSIBLE CODE OVERLAY AT n IN module_name

   which indicates that machine code may be overlaid at memory location n when the procedure file is subsequently loaded. The module_name indicates the object module which causes the new segment to be generated. If the maximum allowable number of segments (16) is exhausted, the Linker will abort.

   Example:

   ```
   LINK $=5000  MODA $=4000  MODB

   module MODB will start a new segment
   ```

2) A new segment is started for machine code which is to be located at an address which is greater than the procedure file's record length beyond the current reference counter value, that is, the address of the last generated byte of machine code. This avoids entire records of meaningless data in the procedure file which might be caused by eiher a large DEFS statement in assembly language or an assignment to the Linker's reference counter using the $=X construct. If the maximum allowable number of segments is exhausted, the Linker is forced to generate all remaining machine code in the last segment.

Example:

```
LINK $=1000 MODA    $=2000 MODB
```

module MODB will start a new segment unless MODA is larger than C00 bytes, assuming a record length of 400.

Link-only Modules

A standard technique for implementing a program which is larger than the available memory space is to "overlay" one portion on top of anoher which is not longer needed. The various portions can be maintained as separate procedure files which are loaded by program interaction with the RIO Executive (see Overlays for further details). In order to facilitate the construction of overlay programs, the Linker provides a method to specify that certain modules are "link-only"; that is, the object modules will be used to help assign module starting addresses and to resolve EXTERNAL references, but will not cause generation of machine code in the resulting procedure files.

Any module name that is immediately preceded by the minus character '-' (ASCII 2DH) will be considered to be a link-only module. A link-only module will appear in the load map along with its GLOBALs, but it will not be loaded as a part of the procedure file.

Example:

```
LINK  $=2000  RESIDENT  -OVERLAY.2
```

would cause the creation of a procedure file containing only the machine code of the module RESIDENT and not that of OVERLAY.2. However, any EXTERNAL references within RESIDENT which are satisfied by GLOBALs in OVERLAY.2 will be correctly resolved.

Module Identification

The Linker follows the general principle that anywhere a file name may be specified, it may be completely or partially qualified by either a device driver name or a drive specification.

For example:

1) LINK MODA MODB.OBJ
2) LINK $MYDOS:2/multiply

The first example demonstrates the use of unqualified file names, MODA and MODB.OBJ, which can be found on the master device. If that device is ZDOS, then the standard disk drive search sequence of drive 1,2,...,0 will be used. The second example demonstrates a fully qualified file name, multiply, which can be found on drive 2 of the device MYDOS.

Each object module must be a binary type file with 128 bytes per record. In addition, an object module contains information within it which identifies it, and the Linker will abort if a module does not contain the correct identification.

Each object module file name must end in the extension .OBJ, and either upper or lower case is acceptable. Furthermore, the extension does not have to be typed in the command line, so that if the last four characters are not .OBJ, the Linker will automatically append the extension before attempting to open the file.

By default, the resulting procedure file will have the same name as the first object file which comprises the program as indicated in the command line, but without the .OBJ extension. The map file and symbol file will have the same name as the procedure file, but with the extensions .MAP and .SYM, respectively. The case of the extension is the same as the first character of the file name.

For example:

1) LINK YOURPROG
   looks for object YOURPROG.OBJ; creates YOURPROG, YOURPROG.MAP, and YOURPROG.SYM

2) LINK myprog1.obj myprog2 TTY.INT
   looks for object modules myprog1.obj, myprog2.obj and TTY.INT.OBJ; creates myprog1, myprog1.map and myprog1.sym

It is possible to override the name of either the procedure file, map file or symbol file by appropriate options in the command line as described below.

2.3 LINKER COMMAND LINE OPTIONS

As the name implies, options are optional and default values will be assumed as indicated. If options are desired, they must be enclosed in parentheses following the object module file names. The format for options is a keyword--usually just the first letter as indicated by the capitals in the following list--sometimes followed by an equal sign (=), and an appropriate value or name as specified for each option. Options can be specified in any order and the last occurrence of a particular option will override previous ones.

Notice that several of the command options are used to control the definition of the logical I/O units used by the Linker. Thus, through interaction with the RIO Executive before giving the LINK command and by using the options which specify a file name (and the Print option), the user can direct the routing of all I/O during linkage. Notice that "filename" in the following descriptions means a standard RIO file name which may be completely or partially qualified.

All fatal error messages which cause the Linker to abort (see Fatal Error Messages) are always routed to the console (logical unit CONOUT). Normally, non-fatal errors and warning messages are also routed to the console, unless overridden by the Print option, in which case they are routed to SYSLST.

Entry=n

Used to specify the starting execution address for the procedure file. n may be either a hex value (must start with a numeric digit, a leading zero is sufficient) or a GLOBAL name. Default--the starting address is taken to be the assigned origin of the first object module. If a GLOBAL name is specified but not found, a warning message will be output and the default origin will be used.

LET

Used to force the second phase of linking even if there were errors such as unresolved EXTERNALs or multiply-defined GLOBALs. Default--errors inhibit the generation of a procedure file.

Map=filename

Used to cause generation of a load map which indicates module starting addresses and lengths. In addition, all GLOBALs are listed (alphabetically sorted) with their assigned addresses and the module which contains them. The filename specifies a file where the load map and any non-fatal error messages are to be output. Default--a load map (ASCII type with 128 bytes per record) is generated and output to a file with extension .MAP with the same name as the procedure file.

Name=filename

Used to specify the name of the procedure file to be created. The name defaults to that of the first module name minus the .OBJ extension. The name will also be given to the map and symbol files (unless overridden by specific options) with the appropriate extensions .MAP and .SYM.

NOMap

Suppresses creation of the load map file. Default--a load map is output to a file.

NOWarning

Used to inhibit warning messages about multiply-defined GLOBALs or unresolved EXTERNALs. Suppresses warnings that possible overlay of code may occur when the reference counter is decremented. Also inhibits the checking for multiply-defined GLOBALs and thus speeds up the first phase of linking. Default--warning messages are output and multiply-defined GLOBALs are checked for.

Print

Causes the output of the load map and error messages to be routed to SYSLST. Default--only error messages are output to CONOUT and nothing is output to SYSLST.

RLength=n

Used to specify the record length of the procedure file. n is a hexadecimal number equal to either 80H, 100H, 200H, or 400H (no trailing 'H'). Default--record length is 80H (128 bytes decimal).

STacksize=n

Used to specify the size of the stack space required by the procedure file when executed. n is a hexadecimal number of bytes. Default--a stack size of 80H bytes is specified.

SYmbol

SYmbol=filename

Used to create a binary symbol file (128 bytes per record) combining the absolutized GLOBALs and the ISD (Internal Symbol Dictionary) of each object module for use by a symbolic debug package. Notice that some modules may have no ISD. The file name, if not specified, defaults to the same name as the procedure file, with extension .SYM. Default--no symbol file is created.

Note the following interactions:

NOM overrides Map=filename.

NOW is an implied LET since errors will be suppressed.

If a procedure file is not successfully generated, there will be no symbol file and the map file will only contain error messages.

If the user specifies a scratch file (i.e., a file name with a zero length name) for an output file, that file will disappear after linkage.

Examples:

1) LINK $=1000 PROG1 PROG2
   produces a procedure file PROG1 (with entry point 1000) and a map file PROG1.MAP on the master device and same drive as the object module, PROG1.OBJ, and will not print the load map on SYSLST.

2) LINK PROG1 (N=$MYDOS:2/PROG.RUN S NOM P E=MAIN)
   produces a procedure file PROG.RUN (with entry point equal to the assigned address of GLOBAL MAIN) and a symbol file PROG.RUN.SYM on drive 2 of the device MYDOS, will not create a map file but will print the load map and any error messages on SYSLST.

Logical Unit Definitions

The Linker uses the following logical I/O units:

2	CONOUT	Error messages
3	SYSLST	Print option output
4		Object files
5		Procedure file
6		Map file
7		Symbol file

Fatal Error Messages

There are several conditions that may occur which cause the Linker to terminate further processing, close all files, and abort by returning to the RIO Executive. The following messages may appear on the console:

INVALID OPTION: s

Indicates that the string s was not recognized as a valid command line option. The cause may be, for instance, a misspelled option specifier or invalid use of delimiters within options. Notice that all numbers must be specified in hexadecimal notation (but no trailing 'H').

LINK DIRECTORY OVERFLOW

Indicates that there is insufficient memory space to permit linkage. The Link Directory is basically a composite list of GLOBALs and EXTERNALs for each object module which the Linker uses to relocate and resolve object code references. File buffers are allocated in proportion to available space; however, there must be room for both the Link Directory and the minimum size buffer for the various files. The size of the Link Directory may be kept small by reducing the number and length of GLOBAL symbols in a program and by not including EXTERNAL declarations for symbols never referenced in a module. This message will also occur if the user attempts to link more than 127 modules together.

PROGRAM TOO BIG

Indicates that the reference counter was incremented past FFFFH, which is beyond the address space of the CPU. The program size or origin should be reduced.

TOO MANY SEGMENTS

Indicates that more than the maximum number (16) of procedure file segments was created. This may occur when the reference counter is decremented too many times to allow a new segment to be generated. The reference counter is decremented by either an assignment to $ in the Linker command line or by an ORG statement in assembly language within an object module. This problem can be avoided by rearranging modules or sections of code so that the reference counter is incremented in ascending order.

FILE NOT FOUND: filename

Indicates that the object "filename" could not be found in a device directory. Remember that the object file name must end in the extension .OBJ.

INVALID FILE: filename

Indicates that the string "filename" is not a proper file name specifier. This may include illegal characters within the name or an invalid device or drive designator (see RIO Operating System User's Manual for further details). The string "filename" is printed as it appeared in the command line.

INVALID FORMAT: filename

Indicates that the object filename had attributes other than type binary (subtype is ignored) and a record length of 128 bytes. This message will also appear if the file does not contain a header of data which is consistent with an object module. This may indicate that the file was damaged and should be regenerated by reassembling the module.

INVALID DATA: filename

Indicates that the object filename contains invalid data, such as a checksum error, or a symbol name whose length is either zero or greater than the maximum allowed (currently 127). This usually occurs as a result of a damaged file which should be regenerated by reassembling the module.

I/O ERROR e ON UNIT u

All other fatal errors involve I/O errors (hex value e) on a particular logical unit (decimal value u). Refer to the Logical Unit Definitions for the Linker and the RIO Operating System User's Manual for an explanation of specific errors.

2.4 MAP FORMAT

The Linker produces a load map which indicates the assigned origin and length of each module, and an alphabetic list of all GLOBALs with their assigned addresses and the modules that contain them. In addition, the map contains the program's name, its length in bytes, and its entry point. Refer to the following example which is, hopefully, self-explanatory.

```
LINK  $=1000 MODA MODB $=2000 COS.MATH SIN.MATH (N=COMPUTE
E=MAIN)

      LINK 1.0

      LOAD MAP
      MODULE           ORIGIN  LENGTH

      MODA               1000  0B32
      MODB               1B32  01AA
      COS.MATH           2000  00B0
      SIN.MATH           20B0  00B6

      GLOBAL          ADDRESS  MODULE

      COS                2000  COS.MATH
      MAIN               10CC  MODA
      SIN                20B0  SIN.MATH
      SQRT               1B32  MODB
      VECTOR             1669  MODA

      PROGRAM COMPUTE -- 1166 BYTES
      ENTRY:  10CC
```

The total number of bytes for a program is computed as the difference between the lowest and highest addresses which actually contain machine code in the procedure file. Thus the number of bytes may not accurately reflect the actual amount of memory a program occupies.

2.5 OVERLAYS

This section provides some guidelines for the user whose program must be implemented as an overlay structure due to memory space constraints. By utilizing certain facilities of the Linker, such as assignment of relocatable module origins and link-only modules, the user may build an effective overlay scheme. Notice, however, that overlay management must be built into the user's program in the sense that the program must explicitly initiate the loading of an overlay segment before any references are made to that segment.

An overlay structure normally has a resident segment which contains data and routines which are common to the various overlay segments. In particular, the routine which initiates the loading of the other segments would normally be in the resident segment. The other segments will normally be maintained as separate procedure files which have been created through a series of linkages as described below.

The method for structuring an overlay program is best illustrated by a simple example. Consider a program (EXAMPLE) which gathers a large amount of data from a particular source (READ), reorganizes and calculates new data (PROCESS), and then outputs this data to a particular destination (WRITE). Assume that this sequence only needs to be done once. Presumably, there is only room for one of these segments to be resident at a time, along with the data and a few common routines, but by simply over-laying the previous segment, the program can accomplish its desired goal.

By properly declaring data and routines as GLOBAL and EXTERNAL in each of the four modules which comprise the program, the user can then combine them in separate linkages as follows. Since READ can fit in memory with the resident segment, EXAMPLE, the two modules are linked together into a single procedure file called EXAMPLE. Notice that because the resident module contains references to both PROCESS and WRITE, these modules are included as link-only modules to allow the proper references to be resolved.

```
1) LINK  EXAMPLE $=5000 READ $=5000 -PROCESS
                        $=5000 -WRITE
```

Then the two overlay segments are created by linking the resident segment with the appropriate module, but specifying the resident as a link-only module.

```
2) LINK -EXAMPLE $=5000 PROCESS  (N=PROCESS)
3) LINK -EXAMPLE $=5000 WRITE    (N=WRITE)
```

The program can be executed by issuing the command EXAMPLE, and provided the resident portion initiates the loading of the two overlay segments at the proper time, the entire program will run to completion.

Overlay procedure files may be loaded in several ways. A program can contain code to open the correct file and read the contents into memory at the right location. Alternatively, a command string which loads a file without executing it can be passed to the RIO operating system by making an I/O request to logical unit 0 with a pointer to a section of memory which contains a string such as 'PROCESS,' followed by a carriage return. See the RIO Operating System User's Manual for further details.

Much more complicated overlay structures can be designed in the same general fashion. For instance, an overlay segment may cause a portion of itself to be overlaid. Or an overlay segment may be loaded several times, instead of just once, and if it contains data that must be saved, it must be written to a temporary file before it is overlaid. By careful planning and programming, the user may utilize the facilities of the Linker to create an overlay schema suitable to the application.

APPENDIX A

OPCODE LISTING

```
LOC   OBJ CODE  M STMT SOURCE STATEMENT

0000  00          1    nop             ; 00
0001  018605    R 2    ld bc,nn        ; 01
0004  02          3    ld (bc),a       ; 02
0005  03          4    inc bc          ; 03
0006  04          5    inc b           ; 04
0007  05          6    dec b           ; 05
0008  0620        7    ld b,n          ; 06
000A  07          8    rlca            ; 07
000B  08          9    ex af,af'       ; 08
000C  09         10    add hl,bc       ; 09
000D  0A         11    ld a,(bc)       ; 0a
000E  0B         12    dec bc          ; 0b
000F  0C         13    inc c           ; 0c
0010  0D         14    dec c           ; 0d
0011  0E20       15    ld c,n          ; 0e
0013  0F         16    rrca            ; 0f
0014  102E       17    djnz $+dis      ; 10
0016  118605    R 18    ld de,nn        ; 11
0019  12         19    ld (de),a       ; 12
001A  13         20    inc de          ; 13
001B  14         21    inc d           ; 14
001C  15         22    dec d           ; 15
001D  1620       23    ld d,n          ; 16
001F  17         24    rla             ; 17
0020  182E       25    jr $+dis        ; 18
0022  19         26    add hl,de       ; 19
0023  1A         27    ld a,(de)       ; 1a
0024  1B         28    dec de          ; 1b
0025  1C         29    inc e           ; 1c
0026  1D         30    dec e           ; 1d
0027  1E20       31    ld e,n          ; 1e
0029  1F         32    rra             ; 1f
002A  202E       33    jr nz,$+dis     ; 20
002C  218605    R 34    ld hl,nn        ; 21
002F  228605    R 35    ld (nn),hl      ; 22
0032  23         36    inc hl          ; 23
0033  24         37    inc h           ; 24
0034  25         38    dec h           ; 25
0035  2620       39    ld h,n          ; 26
0037  27         40    daa             ; 27
0038  282E       41    jr z,$+dis      ; 28
003A  29         42    add hl,hl       ; 29
003B  2A8605    R 43    ld hl,(nn)      ; 2a
003E  2B         44    dec hl          ; 2b
003F  2C         45    inc l           ; 2c
0040  2D         46    dec l           ; 2d
0041  2E20       47    ld l,n          ; 2e
0043  2F         48    cpl             ; 2f
0044  302E       49    jr nc,$+dis     ; 30
0046  318605    R 50    ld sp,nn        ; 31
0049  328605    R 51    ld (nn),a       ; 32
004C  33         52    inc sp          ; 33
004D  34         53    inc (hl)        ; 34
004E  35         54    dec (hl)        ; 35
004F  3620       55    ld (hl),n       ; 36
0051  37         56    scf             ; 37
0052  382E       57    jr c,$+dis      ; 38
0054  39         58    add hl,sp       ; 39
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

0055  3A8605    R   59    ld a,(nn)       ; 3a
0058  3B            60    dec sp          ; 3b
0059  3C            61    inc a           ; 3c
005A  3D            62    dec a           ; 3d
005B  3E20          63    ld a,n          ; 3e
005D  3F            64    ccf             ; 3f
005E  40            65    ld b,b          ; 40
005F  41            66    ld b,c          ; 41
0060  42            67    ld b,d          ; 42
0061  43            68    ld b,e          ; 43
0062  44            69    ld b,h          ; 44
0063  45            70    ld b,l          ; 45
0064  46            71    ld b,(hl)       ; 46
0065  47            72    ld b,a          ; 47
0066  48            73    ld c,b          ; 48
0067  49            74    ld c,c          ; 49
0068  4A            75    ld c,d          ; 4a
0069  4B            76    ld c,e          ; 4b
006A  4C            77    ld c,h          ; 4c
006B  4D            78    ld c,l          ; 4d
006C  4E            79    ld c,(hl)       ; 4e
006D  4F            80    ld c,a          ; 4f
006E  50            81    ld d,b          ; 50
006F  51            82    ld d,c          ; 51
0070  52            83    ld d,d          ; 52
0071  53            84    ld d,e          ; 53
0072  54            85    ld d,h          ; 54
0073  55            86    ld d,l          ; 55
0074  56            87    ld d,(hl)       ; 56
0075  57            88    ld d,a          ; 57
0076  58            89    ld e,b          ; 58
0077  59            90    ld e,c          ; 59
0078  5A            91    ld e,d          ; 5a
0079  5B            92    ld e,e          ; 5b
007A  5C            93    ld e,h          ; 5c
007B  5D            94    ld e,l          ; 5d
007C  5E            95    ld e,(hl)       ; 5e
007D  5F            96    ld e,a          ; 5f
007E  60            97    ld h,b          ; 60
007F  61            98    ld h,c          ; 61
0080  62            99    ld h,d          ; 62
0081  63           100    ld h,e          ; 63
0082  64           101    ld h,h          ; 64
0083  65           102    ld h,l          ; 65
0084  66           103    ld h,(hl)       ; 66
0085  67           104    ld h,a          ; 67
0086  68           105    ld l,b          ; 68
0087  69           106    ld l,c          ; 69
0088  6A           107    ld l,d          ; 6a
0089  6B           108    ld l,e          ; 6b
008A  6C           109    ld l,h          ; 6c
008B  6D           110    ld l,l          ; 6d
008C  6E           111    ld l,(hl)       ; 6e
008D  6F           112    ld l,a          ; 6f
008E  70           113    ld (hl),b       ; 70
008F  71           114    ld (hl),c       ; 71
0090  72           115    ld (hl),d       ; 72
0091  73           116    ld (hl),e       ; 73
```

```
0092   74            117    ld (hl),h       ; 74
0093   75            118    ld (hl),l       ; 75
0094   76            119    halt            ; 76
0095   77            120    ld (hl),a       ; 77
0096   78            121    ld a,b          ; 78
0097   79            122    ld a,c          ; 79
0098   7A            123    ld a,d          ; 7a
0099   7B            124    ld a,e          ; 7b
009A   7C            125    ld a,h          ; 7c
009B   7D            126    ld a,l          ; 7d
009C   7E            127    ld a,(hl)       ; 7e
009D   7F            128    ld a,a          ; 7f
009E   80            129    add a,b         ; 80
009F   81            130    add a,c         ; 81
00A0   82            131    add a,d         ; 82
00A1   83            132    add a,e         ; 83
00A2   84            133    add a,h         ; 84
00A3   85            134    add a,l         ; 85
00A4   86            135    add a,(hl)      ; 86
00A5   87            136    add a,a         ; 87
00A6   88            137    adc a,b         ; 88
00A7   89            138    adc a,c         ; 89
00A8   8A            139    adc a,d         ; 8a
00A9   8B            140    adc a,e         ; 8b
00AA   8C            141    adc a,h         ; 8c
00AB   8D            142    adc a,l         ; 8d
00AC   8E            143    adc a,(hl)      ; 8e
00AD   8F            144    adc a,a         ; 8f
00AE   90            145    sub b           ; 90
00AF   91            146    sub c           ; 91
00B0   92            147    sub d           ; 92
00B1   93            148    sub e           ; 93
00B2   94            149    sub h           ; 94
00B3   95            150    sub l           ; 95
00B4   96            151    sub (hl)        ; 96
00B5   97            152    sub a           ; 97
00B6   98            153    sbc a,b         ; 98
00B7   99            154    sbc a,c         ; 99
00B8   9A            155    sbc a,d         ; 9a
00B9   9B            156    sbc a,e         ; 9b
00BA   9C            157    sbc a,h         ; 9c
00BB   9D            158    sbc a,l         ; 9d
00BC   9E            159    sbc a,(hl)      ; 9e
00BD   9F            160    sbc a,a         ; 9f
00BE   A0            161    and b           ; a0
00BF   A1            162    and c           ; a1
00C0   A2            163    and d           ; a2
00C1   A3            164    and e           ; a3
00C2   A4            165    and h           ; a4
00C3   A5            166    and l           ; a5
00C4   A6            167    and (hl)        ; a6
00C5   A7            168    and a           ; a7
00C6   A8            169    xor b           ; a8
00C7   A9            170    xor c           ; a9
00C8   AA            171    xor d           ; aa
00C9   AB            172    xor e           ; ab
00CA   AC            173    xor h           ; ac
00CB   AD            174    xor l           ; ad
```

```
LOC   OBJ CODE  M STMT SOURCE STATEMENT

00CC  AE          175    xor (hl)        ; ae
00CD  AF          176    xor a           ; af
00CE  B0          177    or b            ; b0
00CF  B1          178    or c            ; b1
00D0  B2          179    or d            ; b2
00D1  B3          180    or e            ; b3
00D2  B4          181    or h            ; b4
00D3  B5          182    or l            ; b5
00D4  B6          183    or (hl)         ; b6
00D5  B7          184    or a            ; b7
00D6  B8          185    cp b            ; b8
00D7  B9          186    cp c            ; b9
00D8  BA          187    cp d            ; ba
00D9  BB          188    cp e            ; bb
00DA  BC          189    cp h            ; bc
00DB  BD          190    cp l            ; bd
00DC  BE          191    cp (hl)         ; be
00DD  BF          192    cp a            ; bf
00DE  C0          193    ret nz          ; c0
00DF  C1          194    pop bc          ; c1
00E0  C28605    R 195    jp nz,nn        ; c2
00E3  C38605    R 196    jp nn           ; c3
00E6  C48605    R 197    call nz,nn      ; c4
00E9  C5          198    push bc         ; c5
00EA  C620        199    add a,n         ; c6
00EC  C7          200    rst 0           ; c7
00ED  C8          201    ret z           ; c8
00EE  C9          202    ret             ; c9
00EF  CA8605    R 203    jp z,nn         ; ca
00F2  CC8605    R 204    call z,nn       ; cc
00F5  CD8605    R 205    call nn         ; cd
00F8  CE20        206    adc a,n         ; c
00FA  CF          207    rst 8           ; cf
00FB  D0          208    ret nc          ; d0
00FC  D1          209    pop de          ; d1
00FD  D28605    R 210    jp nc,nn        ; d2
0100  D320        211    out (n),a       ; d3
0102  D48605    R 212    call nc,nn      ; d4
0105  D5          213    push de         ; d5
0106  D620        214    sub n           ; d6
0108  D7          215    rst 10h         ; d7
0109  D8          216    ret c           ; d8
010A  D9          217    exx             ; d9
010B  DA8605    R 218    jp c,nn         ; da
010E  DB20        219    in a,(n)        ; db
0110  DC8605    R 220    call c,nn       ; dc
0113  DE20        221    sbc a,n         ; de
0115  DF          222    rst 18h         ; df
0116  E0          223    ret po          ; e0
0117  E1          224    pop hl          ; e1
0118  E28605    R 225    jp po,nn        ; e2
011B  E3          226    ex (sp),hl      ; e3
011C  E48605    R 227    call po,nn      ; e4
011F  E5          228    push hl         ; e5
0120  E620        229    and n           ; e6
0122  E7          230    rst 20h         ; e7
0123  E8          231    ret pe          ; e8
0124  E9          232    jp (hl)         ; e9
```

```
0125  EA8605    R  233     jp pe,nn        ; ea
0128  EB            234     ex de,hl        ; eb
0129  EC8605    R  235     call pe,nn      ; ec
012C  EE20          236     xor n           ; ee
012E  EF            237     rst 28h         ; ef
012F  F0            238     ret p           ; f0
0130  F1            239     pop af          ; f1
0131  F28605    R  240     jp p,nn         ; f2
0134  F3            241     di              ; f3
0135  F48605    R  242     call p,nn       ; f4
0138  F5            243     push af         ; f5
0139  F620          244     or n            ; f6
013B  F7            245     rst 30h         ; f7
013C  F8            246     ret m           ; f8
013D  F9            247     ld sp,hl        ; f9
013E  FA8605    R  248     jp m,nn         ; fa
0141  FB            249     ei              ; fb
0142  FC8605    R  250     call m,nn       ; fc
0145  FE20          251     cp n            ; fe
0147  FF            252     rst 38h         ; ff
0148  CB00          253     rlc b           ; 00
014A  CB01          254     rlc c           ; 01
014C  CB02          255     rlc d           ; 02
014E  CB03          256     rlc e           ; 03
0150  CB04          257     rlc h           ; 04
0152  CB05          258     rlc l           ; 05
0154  CB06          259     rlc (hl)        ; 06
0156  CB07          260     rlc a           ; 07
0158  CB08          261     rrc b           ; 08
015A  CB09          262     rrc c           ; 09
015C  CB0A          263     rrc d           ; 0a
015E  CB0B          264     rrc e           ; 0b
0160  CB0C          265     rrc h           ; 0c
0162  CB0D          266     rrc l           ; 0d
0164  CB0E          267     rrc (hl)        ; 0e
0166  CB0F          268     rrc a           ; 0f
0168  CB10          269     rl b            ; 10
016A  CB11          270     rl c            ; 11
016C  CB12          271     rl d            ; 12
016E  CB13          272     rl e            ; 13
0170  CB14          273     rl h            ; 14
0172  CB15          274     rl l            ; 15
0174  CB16          275     rl (hl)         ; 16
0176  CB17          276     rl a            ; 17
0178  CB18          277     rr b            ; 18
017A  CB19          278     rr c            ; 19
017C  CB1A          279     rr d            ; 1a
017E  CB1B          280     rr e            ; 1b
0180  CB1C          281     rr h            ; 1c
0182  CB1D          282     rr l            ; 1d
0184  CB1E          283     rr (hl)         ; 1e
0186  CB1F          284     rr a            ; 1f
0188  CB20          285     sla b           ; 20
018A  CB21          286     sla c           ; 21
018C  CB22          287     sla d           ; 22
018E  CB23          288     sla e           ; 23
0190  CB24          289     sla h           ; 24
0192  CB25          290     sla l           ; 25
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

0194  CB26        291    sla (hl)        ; 26
0196  CB27        292    sla a           ; 27
0198  CB28        293    sra b           ; 28
019A  CB29        294    sra c           ; 29
019C  CB2A        295    sra d           ; 2a
019E  CB2B        296    sra e           ; 2b
01A0  CB2C        297    sra h           ; 2c
01A2  CB2D        298    sra l           ; 2d
01A4  CB2E        299    sra (hl)        ; 2e
01A6  CB2F        300    sra a           ; 2f
01A8  CB38        301    srl b           ; 38
01AA  CB39        302    srl c           ; 39
01AC  CB3A        303    srl d           ; 3a
01AE  CB3B        304    srl e           ; 3b
01B0  CB3C        305    srl h           ; 3c
01B2  CB3D        306    srl l           ; 3d
01B4  CB3E        307    srl (hl)        ; 3e
01B6  CB3F        308    srl a           ; 3f
01B8  CB40        309    bit 0,b         ; 40
01BA  CB41        310    bit 0,c         ; 41
01BC  CB42        311    bit 0,d         ; 42
01BE  CB43        312    bit 0,e         ; 43
01C0  CB44        313    bit 0,h         ; 44
01C2  CB45        314    bit 0,l         ; 45
01C4  CB46        315    bit 0,(hl)      ; 46
01C6  CB47        316    bit 0,a         ; 47
01C8  CB48        317    bit 1,b         ; 48
01CA  CB49        318    bit 1,c         ; 49
01CC  CB4A        319    bit 1,d         ; 4a
01CE  CB4B        320    bit 1,e         ; 4b
01D0  CB4C        321    bit 1,h         ; 4c
01D2  CB4D        322    bit 1,l         ; 4d
01D4  CB4E        323    bit 1,(hl)      ; 4e
01D6  CB4F        324    bit 1,a         ; 4f
01D8  CB50        325    bit 2,b         ; 50
01DA  CB51        326    bit 2,c         ; 51
01DC  CB52        327    bit 2,d         ; 52
01DE  CB53        328    bit 2,e         ; 53
01E0  CB54        329    bit 2,h         ; 54
01E2  CB55        330    bit 2,l         ; 55
01E4  CB57        331    bit 2,a         ; 57
01E6  CB56        332    bit 2,(hl)      ; 56
01E8  CB58        333    bit 3,b         ; 58
01EA  CB59        334    bit 3,c         ; 59
01EC  CB5B        335    bit 3,e         ; 5a
01EE  CB5B        336    bit 3,e         ; 5b
01F0  CB5C        337    bit 3,h         ; 5c
01F2  CB5D        338    bit 3,l         ; 5d
01F4  CB5E        339    bit 3,(hl)      ; 5e
01F6  CB5F        340    bit 3,a         ; 5f
01F8  CB60        341    bit 4,b         ; 60
01FA  CB61        342    bit 4,c         ; 61
01FC  CB62        343    bit 4,d         ; 62
01FE  CB63        344    bit 4,e         ; 63
0200  CB64        345    bit 4,h         ; 64
0202  CB65        346    bit 4,l         ; 65
0204  CB66        347    bit 4,(hl)      ; 66
0206  CB67        348    bit 4,a         ; 67
```

```
0208    CB68        349     bit 5,b         ; 68
020A    CB69        350     bit 5,c         ; 69
020C    CB6A        351     bit 5,d         ; 6a
020E    CB6B        352     bit 5,e         ; 6b
0210    CB6C        353     bit 5,h         ; 6c
0212    CB6D        354     bit 5,l         ; 6d
0214    CB6E        355     bit 5,(hl)      ; 6e
0216    CB6F        356     bit 5,a         ; 6f
0218    CB70        357     bit 6,b         ; 70
021A    CB71        358     bit 6,c         ; 71
021C    CB72        359     bit 6,d         ; 72
021E    CB73        360     bit 6,e         ; 73
0220    CB74        361     bit 6,h         ; 74
0222    CB75        362     bit 6,l         ; 75
0224    CB76        363     bit 6,(hl)      ; 76
0226    CB77        364     bit 6,a         ; 77
0228    CB78        365     bit 7,b         ; 78
022A    CB79        366     bit 7,c         ; 79
022C    CB7A        367     bit 7,d         ; 7a
022E    CB7B        368     bit 7,e         ; 7b
0230    CB7C        369     bit 7,h         ; 7c
0232    CB7D        370     bit 7,l         ; 7d
0234    CB7E        371     bit 7,(hl)      ; 7e
0236    CB7F        372     bit 7,a         ; 7f
0238    CB80        373     res 0,b         ; 80
023A    CB81        374     res 0,c         ; 81
023C    CB82        375     res 0,d         ; 82
023E    CB83        376     res 0,e         ; 83
0240    CB84        377     res 0,h         ; 84
0242    CB85        378     res 0,l         ; 85
0244    CB86        379     res 0,(hl)      ; 86
0246    CB87        380     res 0,a         ; 87
0248    CB88        381     res 1,b         ; 88
024A    CB89        382     res 1,c         ; 89
024C    CB8A        383     res 1,d         ; 8a
024E    CB8B        384     res 1,e         ; 8b
0250    CB8C        385     res 1,h         ; 8c
0252    CB8D        386     res 1,l         ; 8d
0254    CB8E        387     res 1,(hl)      ; 8e
0256    CB8F        388     res 1,a         ; 8f
0258    CB90        389     res 2,b         ; 90
025A    CB91        390     res 2,c         ; 91
025C    CB92        391     res 2,d         ; 92
025E    CB93        392     res 2,e         ; 93
0260    CB94        393     res 2,h         ; 94
0262    CB95        394     res 2,l         ; 95
0264    CB96        395     res 2,(hl)      ; 96
0266    CB97        396     res 2,a         ; 97
0268    CB98        397     res 3,b         ; 98
026A    CB99        398     res 3,c         ; 99
026C    CB9A        399     res 3,d         ; 9a
026E    CB9B        400     res 3,e         ; 9b
0270    CB9C        401     res 3,h         ; 9c
0272    CB9D        402     res 3,l         ; 9d
0274    CB9E        403     res 3,(hl)      ; 9e
0276    CB9F        404     res 3,a         ; 9f
0278    CBA0        405     res 4,b         ; a0
027A    CBA1        406     res 4,c         ; a1
```

```
027C   CBA2          407     res 4,d         ; a2
027E   CBA3          408     res 4,e         ; a3
0280   CBA4          409     res 4,h         ; a4
0282   CBA5          410     res 4,l         ; a5
0284   CBA6          411     res 4,(hl)      ; a6
0286   CBA7          412     res 4,a         ; a7
0288   CBA8          413     res 5,b         ; a8
028A   CBA9          414     res 5,c         ; a9
028C   CBAA          415     res 5,d         ; aa
028E   CBAB          416     res 5,e         ; ab
0290   CBAC          417     res 5,h         ; ac
0292   CBAD          418     res 5,l         ; ad
0294   CBAE          419     res 5,(hl)      ; ae
0296   CBAF          420     res 5,a         ; af
0298   CBB0          421     res 6,b         ; b0
029A   CBB1          422     res 6,c         ; b1
029C   CBB2          423     res 6,d         ; b2
029E   CBB3          424     res 6,e         ; be
02A0   CBB4          425     res 6,h         ; b4
02A2   CBB5          426     res 6,l         ; b5
02A4   CBB6          427     res 6,(hl)      ; b6
02A6   CBB7          428     res 6,a         ; b7
02A8   CBB8          429     res 7,b         ; b8
02AA   CBB9          430     res 7,c         ; b9
02AC   CBBA          431     res 7,d         ; ba
02AE   CBBB          432     res 7,e         ; bb
02B0   CBBC          433     res 7,h         ; bc
02B2   CBBD          434     res 7,l         ; bd
02B4   CBBF          435     res 7,a         ; bf
02B6   CBC0          436     set 0,b         ; c0
02B8   CBC1          437     set 0,c         ; c1
02BA   CBC2          438     set 0,d         ; c2
02BC   CBC3          439     set 0,e         ; c3
02BE   CBC4          440     set 0,h         ; c4
02C0   CBC5          441     set 0,l         ; c5
02C2   CBC6          442     set 0,(hl)      ; c6
02C4   CBC7          443     set 0,a         ; c7
02C6   CBC8          444     set 1,b         ; c8
02C8   CBC9          445     set 1,c         ; c9
02CA   CBCA          446     set 1,d         ; ca
02CC   CBCB          447     set 1,e         ; cb
02CE   CBCC          448     set 1,h         ; cc
02D0   CBCD          449     set 1,l         ; cd
02D2   CBCE          450     set 1,(hl)      ; ce
02D4   CBCF          451     set 1,a         ; cf
02D6   CBD0          452     set 2,b         ; d0
02D8   CBD1          453     set 2,c         ; d1
02DA   CBD2          454     set 2,d         ; d2
02DC   CBD3          455     set 2,e         ; d3
02DE   CBD4          456     set 2,h         ; d4
02E0   CBD5          457     set 2,l         ; d5
02E2   CBD6          458     set 2,(hl)      ; d6
02E4   CBD7          459     set 2,a         ; d7
02E6   CBD8          460     set 3,b         ; d8
02E8   CBD9          461     set 3,c         ; d9
02EA   CBDA          462     set 3,d         ; da
02EC   CBDB          463     set 3,e         ; db
02EE   CBDC          464     set 3,h         ; dc
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

02F0  CBDD       465    set 3,l          ; dd
02F2  CBDE       466    set 3,(hl)       ; de
02F4  CBDF       467    set 3,a          ; df
02F6  CBE0       468    set 4,b          ; e0
02F8  CBE1       469    set 4,c          ; e1
02FA  CBE2       470    set 4,d          ; e2
02FC  CBE3       471    set 4,e          ; e3
02FE  CBE4       472    set 4,h          ; e4
0300  CBE5       473    set 4,l          ; e5
0302  CBE6       474    set 4,(hl)       ; e6
0304  CBE7       475    set 4,a          ; e7
0306  CBE8       476    set 5,b          ; e8
0308  CBE9       477    set 5,c          ; e9
030A  CBEA       478    set 5,d          ; ea
030C  CBEB       479    set 5,e          ; eb
030E  CBEC       480    set 5,h          ; ec
0310  CBED       481    set 5,l          ; ed
0312  CBEE       482    set 5,(hl)       ; ee
0314  CBEF       483    set 5,a          ; ef
0316  CBF0       484    set 6,b          ; f0
0318  CBF1       485    set 6,c          ; f1
031A  CBF2       486    set 6,d          ; f2
031C  CBF3       487    set 6,e          ; f3
031E  CBF4       488    set 6,h          ; f4
0320  CBF5       489    set 6,l          ; f5
0322  CBF6       490    set 6,(hl)       ; f6
0324  CBF7       491    set 6,a          ; f7
0326  CBF8       492    set 7,b          ; f8
0328  CBF9       493    set 7,c          ; f9
032A  CBFA       494    set 7,d          ; fa
032C  CBFB       495    set 7,e          ; fb
032E  CBFC       496    set 7,h          ; fc
0330  CBFD       497    set 7,l          ; fd
0332  CBFE       498    set 7,(hl)       ; fe
0334  CBFF       499    set 7,a          ; ff
0336  DD09       500    add ix,bc        ; 09
0338  DD19       501    add ix,de        ; 19
033A  DD218605 R  502    ld ix,nn         ; 21
033E  DD228605 R  503    ld (nn),ix       ; 22
0342  DD23       504    inc ix           ; 23
0344  DD29       505    add ix,ix        ; 29
0346  DD2A8605 R  506    ld ix,(nn)       ; 2a
034A  DD2B       507    dec ix           ; 2b
034C  DD3405     508    inc (ix+ind)     ; 34
034F  DD3505     509    dec (ix+ind)     ; 35
0352  DD360520   510    ld (ix+ind),n    ; 36
0356  DD39       511    add ix,sp        ; 39
0358  DD4605     512    ld b,(ix+ind)    ; 46
035B  DD4E05     513    ld c,(ix+ind)    ; 4e
035E  DD5605     514    ld d,(ix+ind)    ; 56
0361  DD5E05     515    ld e,(ix+ind)    ; 5e
0364  DD6605     516    ld h,(ix+ind)    ; 66
0367  DD6E05     517    ld l,(ix+ind)    ; 6e
036A  DD7005     518    ld (ix+ind),b    ; 70
036D  DD7105     519    ld (ix+ind),c    ; 71
0370  DD7205     520    ld (ix+ind),d    ; 72
0373  DD7305     521    ld (ix+ind),e    ; 73
0376  DD7405     522    ld (ix+ind),h    ; 74
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

0379  DD7505      523    ld (ix+ind),l  ; 75
037C  DD7705      524    ld (ix+ind),a  ; 77
037F  DD7E05      525    ld a,(ix+ind)  ; 7e
0382  DD8605      526    add a,(ix+ind) ; 86
0385  DD8E05      527    adc a,(ix+ind) ; 8e
0388  DD9605      528    sub (ix+ind)   ; 96
038B  DD9E05      529    sbc a,(ix+ind) ; 9e
038E  DDA605      530    and (ix+ind)   ; a6
0391  DDAE05      531    xor (ix+ind)   ; ae
0394  DDB605      532    or (ix+ind)    ; b6
0397  DDBE05      533    cp (ix+ind)    ; be
039A  DDE1        534    pop ix         ; e1
039C  DDE3        535    ex (sp),ix     ; e3
039E  DDE5        536    push ix        ; e5
03A0  DDE9        537    jp (ix)        ; e9
03A2  DDF9        538    ld sp,ix       ; f9
03A4  DDCB0506    539    rlc (ix+ind)   ; 06
03A8  DDCB050E    540    rrc (ix+ind)   ; 0e
03AC  DDCB0516    541    rl (ix+ind)    ; 16
03B0  DDCB051E    542    rr (ix+ind)    ; 1e
03B4  DDCB0526    543    sla (ix+ind)   ; 26
03B8  DDCB052E    544    sra (ix+ind)   ; 2e
03BC  DDCB053E    545    srl (ix+ind)   ; 3e
03C0  DDCB0546    546    bit 0,(ix+ind) ; 46
03C4  DDCB054E    547    bit 1,(ix+ind) ; 4e
03C8  DDCB0556    548    bit 2,(ix+ind) ; 56
03CC  DDCB055E    549    bit 3,(ix+ind) ; 5e
03D0  DDCB0566    550    bit 4,(ix+ind) ; 66
03D4  DDCB056E    551    bit 5,(ix+ind) ; 6e
03D8  DDCB0576    552    bit 6,(ix+ind) ; 76
03DC  DDCB057E    553    bit 7,(ix+ind) ; 7e
03E0  DDCB0586    554    res 0,(ix+ind) ; 86
03E4  DDCB058E    555    res 1,(ix+ind) ; 8e
03E8  DDCB0596    556    res 2,(ix+ind) ; 96
03EC  DDCB059E    557    res 3,(ix+ind) ; 9e
03F0  DDCB05A6    558    res 4,(ix+ind) ; a6
03F4  DDCB05AE    559    res 5,(ix+ind) ; ae
03F8  DDCB05B6    560    res 6,(ix+ind) ; b6
03FC  DDCB05BE    561    res 7,(ix+ind) ; be
0400  DDCB05C6    562    set 0,(ix+ind) ; c6
0404  DDCB05CE    563    set 1,(ix+ind) ; ce
0408  DDCB05D6    564    set 2,(ix+ind) ; d6
040C  DDCB05DE    565    set 3,(ix+ind) ; de
0410  DDCB05E6    566    set 4,(ix+ind) ; e6
0414  DDCB05EE    567    set 5,(ix+ind) ; ee
0418  DDCB05F6    568    set 6,(ix+ind) ; f6
041C  DDCB05FE    569    set 7,(ix+ind) ; fe
0420  ED40        570    in b,(c)       ; 40
0422  ED41        571    out (c),b      ; 41
0424  ED42        572    sbc hl,bc      ; 42
0426  ED438605 R  573    ld (nn),bc     ; 43
042A  ED44        574    neg            ; 44
042C  ED45        575    retn           ; 45
042E  ED46        576    im 0           ; 46
0430  ED47        577    ld i,a         ; 47
0432  ED48        578    in c,(c)       ; 48
0434  ED49        579    out (c),c      ; 49
0436  ED4A        580    adc hl,bc      ; 4a
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

0438  ED4B8605 R  581    ld bc,(nn)      ; 4b
043C  ED4D         582    reti            ; 4d
043E  ED4F         583    ld r,a          ; 4f
0440  ED50         584    in d,(c)        ; 50
0442  ED51         585    out (c),d       ; 51
0444  ED52         586    sbc hl,de       ; 52
0446  ED538605 R   587    ld (nn),de      ; 53
044A  ED56         588    im 1            ; 56
044C  ED57         589    ld a,i          ; 57
044E  ED58         590    in e,(c)        ; 58
0450  ED59         591    out (c),e       ; 59
0452  ED5A         592    adc hl,de       ; 5a
0454  ED5B8605 R   593    ld de,(nn)      ; 5b
0458  ED5E         594    im 2            ; 5e
045A  ED5F         595    ld a,r          ; 5f
045C  ED60         596    in h,(c)        ; 60
045E  ED61         597    out (c),h       ; 61
0460  ED62         598    sbc hl,hl       ; 62
0462  ED67         599    rrd             ; 67
0464  ED68         600    in l,(c)        ; 68
0466  ED69         601    out (c),l       ; 69
0468  ED6A         602    adc hl,hl       ; 6a
046A  ED6F         603    rld             ; 6f
046C  ED72         604    sbc hl,sp       ; 72
046E  ED738605 R   605    ld (nn),sp      ; 73
0472  ED78         606    in a,(c)        ; 78
0474  ED79         607    out (c),a       ; 79
0476  ED7A         608    adc hl,sp       ; 7a
0478  ED7B8605 R   609    ld sp,(nn)      ; 7b
047C  EDA0         610    ldi             ; a0
047E  EDA1         611    cpi             ; a1
0480  EDA2         612    ini             ; a2
0482  EDA3         613    outi            ; a3
0484  EDA8         614    ldd             ; a8
0486  EDA9         615    cpd             ; a9
0488  EDAA         616    ind             ; aa
048A  EDAB         617    outd            ; ab
048C  EDB0         618    ldir            ; b0
048E  EDB1         619    cpir            ; b1
0490  EDB2         620    inir            ; b2
0492  EDB3         621    otir            ; b3
0494  EDB8         622    lddr            ; b8
0496  EDB9         623    cpdr            ; b9
0498  EDBA         624    indr            ; ba
049A  EDBB         625    otdr            ; bb
049C  FD09         626    add iy,bc       ; 09
049E  FD19         627    add iy,de       ; 19
04A0  FD218605 R   628    ld iy,nn        ; 21
04A4  FD228605 R   629    ld (nn),iy      ; 22
04A8  FD23         630    inc iy          ; 23
04AA  FD29         631    add iy,iy       ; 29
04AC  FD2A8605 R   632    ld iy,(nn)      ; 2a
04B0  FD2B         633    dec iy          ; 2b
04B2  FD3405       634    inc (iy+ind)    ; 34
04B5  FD3505       635    dec (iy+ind)    ; 35
04B8  FD360520     636    ld (iy+ind),n   ; 36
04BC  FD39         637    add iy,sp       ; 39
04BE  FD4605       638    ld b,(iy+ind)   ; 46
```

```
04C1   FD4E05      639    ld c,(iy+ind)  ; 4e
04C4   FD5605      640    ld d,(iy+ind)  ; 56
04C7   FD5E05      641    ld e,(iy+ind)  ; 5e
04CA   FD6605      642    ld h,(iy+ind)  ; 66
04CD   FD6E05      643    ld l,(iy+ind)  ; 6e
04D0   FD7005      644    ld (iy+ind),b  ; 70
04D3   FD7105      645    ld (iy+ind),c  ; 71
04D6   FD7205      646    ld (iy+ind),d  ; 72
04D9   FD7305      647    ld (iy+ind),e  ; 73
04DC   FD7405      648    ld (iy+ind),h  ; 74
04DF   FD7505      649    ld (iy+ind),l  ; 75
04E2   FD7705      650    ld (iy+ind),a  ; 77
04E5   FD7E05      651    ld a,(iy+ind)  ; 7e
04E8   FD8605      652    add a,(iy+ind) ; 86
04EB   FD8E05      653    adc a,(iy+ind) ; 8e
04EE   FD9605      654    sub (iy+ind)   ; 96
04F1   FD9E05      655    sbc a,(iy+ind) ; 9e
04F4   FDA605      656    and (iy+ind)   ; a6
04F7   FDAE05      657    xor (iy+ind)   ; ae
04FA   FDB605      658    or (iy+ind)    ; b6
04FD   FDBE05      659    cp (iy+ind)    ; be
0500   FDE1        660    pop iy         ; e1
0502   FDE3        661    ex (sp),iy     ; e3
0504   FDE5        662    push iy        ; e5
0506   FDE9        663    jp (iy)        ; e9
0508   FDF9        664    ld sp,iy       ; f9
050A   FDCB0506    665    rlc (iy+ind)   ; 06
050E   FDCB050E    666    rrc (iy+ind)   ; 0e
0512   FDCB0516    667    rl (iy+ind)    ; 16
0516   FDCB051E    668    rr (iy+ind)    ; 1e
051A   FDCB0526    669    sla (iy+ind)   ; 26
051E   FDCB052E    670    sra (iy+ind)   ; 2e
0522   FDCB053E    671    srl (iy+ind)   ; 3e
0526   FDCB0546    672    bit 0,(iy+ind) ; 46
052A   FDCB054E    673    bit 1,(iy+ind) ; 4e
052E   FDCB0556    674    bit 2,(iy+ind) ; 56
0532   FDCB055E    675    bit 3,(iy+ind) ; 5e
0536   FDCB0566    676    bit 4,(iy+ind) ; 66
053A   FDCB056E    677    bit 5,(iy+ind) ; 6e
053E   FDCB0576    678    bit 6,(iy+ind) ; 76
0542   FDCB057E    679    bit 7,(iy+ind) ; 7e
0546   FDCB0586    680    res 0,(iy+ind) ; 86
054A   FDCB058E    681    res 1,(iy+ind) ; 8e
054E   FDCB0596    682    res 2,(iy+ind) ; 96
0552   FDCB059E    683    res 3,(iy+ind) ; 9e
0556   FDCB05A6    684    res 4,(iy+ind) ; a6
055A   FDCB05AE    685    res 5,(iy+ind) ; ae
055E   FDCB05B6    686    res 6,(iy+ind) ; b6
0562   FDCB05BE    687    res 7,(iy+ind) ; be
0566   FDCB05C6    688    set 0,(iy+ind) ; c6
056A   FDCB05CE    689    set 1,(iy+ind) ; ce
056E   FDCB05D6    690    set 2,(iy+ind) ; d6
0572   FDCB05DE    691    set 3,(iy+ind) ; de
0576   FDCB05E6    692    set 4,(iy+ind) ; e6
057A   FDCB05EE    693    set 5,(iy+ind) ; ee
057E   FDCB05F6    694    set 6,(iy+ind) ; f6
0582   FDCB05FE    695    set 7,(iy+ind) ; fe
                   696
```

```
0586                    697 nn   defs 2
                        698 ind  equ  5
                        699 m    equ  10h
                        700 n    equ  20h
                        701 dis  equ  30h
                        702      end
```

APPENDIX B

OBJECT MODULE FORMAT

```
Header - - - - - ESD - - - - - Code Blocks - - - - - ISD
16 Bytes        Variable      128 Bytes/Block       Variable
```

I. Header

Byte #0	: 80 hex (absolute object file mark) 81 81 hex (relocatable object file mark)
1	: Reserved for future use
2...3	: # of bytes of code in module
4...5	: # of ESD entries
6...7	: Record # of first code block
8...9	: Lowest absolute address used if absolute module
10...11	: Reserved for future use
12...13	: # of ISD entries
14...15	: Record # of first ISD block

II. ESD (EXTERNAL SYMBOL DICTIONARY)

Each entry is of variable length, with zero-filled entries, if necessary, to fill the last physical record of the ESD. The format of an entry of size n:

Byte #0	: Attribute byte - high order bit indicates a GLOBAL (bit=1) or EXTERNAL (bit=0) Low order bit indicates whether GLOBAL is relocatable (bit=1) or absolute (bit=0). All other bits are reserved for future use (currently=0).
1	: Symbol name length in bytes (i.e., n-4)
2...n-3	: Symbol name (ASCII characters)
n-2...n-3	: 16 bit value In the case of a GLOBAL name, the value is the address which defines the symbol. The value of an EXTERNAL name is its index (relative byte position) in the ESD which will be modified by the Linker.

III. CODE BLOCK

Byte #0 : Code Mark
(83 hex if this is the last code block in the module, 82 hex otherwise)

1...2 : Reference counter value of first byte in block

3 : # of code bytes in block

4...17 : Bitmap (only 5 high-order bits of #17 are used). The bitmap is organized with 1 bit per byte of code, where 0=absolute; 1,0=relocatable; 1,1=EXTERNAL reference.

18...126 : Object code
The two bytes for a relocatable reference is a 16 bit signed quantity to be added to the assigned starting location of the module at link time. The 2 bytes of code for an EXTERNAL reference is a 16 bit signed quantity to be added to the final assigned absolute address of the EXTERNAL. In addition, there are 2 extra bytes which contain a relative pointer to the attribute byte of the EXTERNAL name in the ESD (these two bytes are the relative byte position of the EXTERNAL in the ESD). These two bytes are not actually converted to machine code during the linking process, and thus have zero bits in the corresponding bitmap entries.

*** Relocatable and EXTERNAL references will not be split across a record boundary; instead, when necessary, they will start a new record.

127 : Checksum (sum of bytes up to Checksum + Checksum =0). In fact, the checksum byte actually follows the last good code byte in the block, any following bytes are ignored. Thus, in a block with zero bytes of code, the checksum byte will actually occur in byte #18.

IV. ISD (INTERNAL SYMBOL DICTIONARY - appended by Symbol Option in Assembler)

Same variable format as ESD - only difference is there are no GLOBALs and thus the GLOBAL bit in the attribute byte is always reset (=0). The low order bit of attribute byte indicates whether internal symbol is relocatable (bit=1) or absolute (bit=0).

APPENDIX C

ASSEMBLY ERROR MESSAGES AND EXPLANATIONS

1) WARNING - OPCODE REDEFINED
   Indicates that an opcode has been redefined by a macro so that future uses of the opcode will result in the appropriate macro call. This message may be suppressed by the NOW option.

2) NAME CONTAINS INVALID CHARACTERS
   Indicates that a name (either a label or an operand) contains illegal characters. Names must start with an alphabetic character and any following characters must be either alphanumeric (A...Z or 0...9), a question mark (?) or an underbar (_).

3) INVALID OPCODE
   Indicates that the opcode was not recognized. Occurs when the opcode contains an illegal character (including non-printing control characters), when the opcode is not either all upper case or all lower case, or when macros are used and the M option is not specified.

4) INVALID NUMBER
   Indicates an invalid character in a number. Occurs when a number contains an illegal character (including non-printing control characters) or a number contains a digit not allowed in the specified base (e.g., 8 or 9 in an octal number or a letter in a hexadecimal number where the trailing H was omitted).

5) INVALID OPERATOR
   Indicates use of an invalid operator in an expression. Occurs when an operator such as .AND. or .XOR. is misspelled or contains illegal characters.

6) SYNTAX ERROR
   Indicates the syntax error of the statement is invalid. Occurs when an expression is incorrectly formed, unmatched parentheses are found in an operand field, or a DEFM or DEFT string is either too long (greater than 63 characters) or contains unbalanced quotes.

7) ASSEMBLER ERROR
   Indicates that the assembler has failed to process this instruction. Usually occurs when an expression is incorrectly formed.

8) UNDEFINED SYMBOL
Indicates that a symbol in an operand field is misspelled or not declared as a label for an instruction or pseudo-op, or was not declared EXTERNAL.

9) INVALID OPERAND COMBINATION
Indicates that the operand combination for this opcode is invalid. Occurs when a register name or condition code is misspelled or incorrectly used with the particular opcode.

10) EXPRESSION OUT OF RANGE
Indicates that the value of an expression is either too large or too small for the appropriate quantity. Occurs on 16-bit arithmetic overflow
w or division by zero in an expression, incrementing the reference counter beyond a 16-bit value, or trying to use a value which will not fit into a particular bit-field--typically a byte.

11) MULTIPLE DECLARATION
Indicates that an attempt was made to redefine a label. Occurs when a label is misspelled, or mistakenly used several times. The pseudo-op DEFL can be used to assign a value to a label which can then be redefined by another DEFL.

12) MACRO DEFINITION ERROR
Indicates that a macro is incorrectly defined. Occurs when the M option is not specified but macros are used, when a macro is defined within another macro definition, when the parameters are not correctly specified, or an unrecognized parameter is found in the macro body.

13) UNBALANCED QUOTES
Indicates that a string is not properly bounded by single quote marks, or quote marks inside a string are not properly matched in pairs.

14) ASSEMBLER COMMAND ERROR
Indicates that an assembler command is not recognized or correctly formed. The command must begin with an asterisk (*) in column one, the first letter identifies the command, and any parameters such as 'ON', 'OFF' or a filename must be properly delimited. The command will be ignored.

15) MACRO EXPANSION ERROR
Indicates that the expansion of a single line in a macro has overflowed the expansion buffer. Occurs when substitution of a parameter causes the line to increase in length beyond the capacity of the buffer (currently 128 bytes). The line will be truncated.

16) MACRO STACK OVERFLOW
Indicates that the depth of nesting of macro calls has exceeded the macro parameter stack buffer capacity. Occurs when the sum of the parameter string lengths (plus some additional information for each macro call) is longer than the buffer (currently 256 bytes), which often happens if infinitely recursive macro calls are used. The macro call which caused the error will be ignored.

17) INCLUDE NESTED TOO DEEP
Indicates that an *Include command was found which would have caused a nesting of included source files to a depth greater than four, where the original source file is considered to be level one. The command will be ignored.

18) GLOBAL DEFINITION ERROR
Indicates that either a label was present on a GLOBAL pseudo-op statement, or there was an attempt to give an absolute value to a GLOBAL symbol in a relocatable module. The latter case is not allowed since all GLOBALs in a relocatable module will be relocated by the Linker. May occur either after a GLOBAL pseudo-op or after an EQU or DEFL statement which is attempting to absolutize a relocatable GLOBAL symbol.

19) EXTERNAL DEFINITION ERROR
Indicates that either a label was present on an EXTERNAL pseudo-op statement, or there was an attempt to declare a symbol to be EXTERNAL which had previously been defined within the module to have an absolute value. May occur due to a misspelling or other oversight.

20) NAME DECLARED GLOBAL AND EXTERNAL
Indicates that the name was found in both a GLOBAL pseudo-op and an EXTERNAL pseudo-op, which is contradictory. May occur due to a misspelling or other oversight.

21) LABEL DECLARED AS EXTERNAL
Indicates that a name has been declared in both an EXTERNAL pseudo-op and as a label in this module. May occur due to a misspelling or other oversight.

22) INVALID EXTERNAL EXPRESSION
Indicates that a symbol name which has been declared in an EXTERNAL pseudo-op is improperly used in an expression. May occur when invalid arithmetic operators are applied to an external expression or when the mode of an operand must be either absolute or relocatable.

23) INVALID RELOCATABLE EXPRESSION
Indicates that an expression which contains a relocatable value (either a label or the reference counter symbol $ in a relocatable module) is improperly formed or used. May occur when invalid arithmetic operators are applied to a relocatable expression or when the mode of an operand must be absolute. Remember that all relocatable values (addresses) must be represented in 16 bits.

24) EXPRESSION MUST BE ABSOLUTE
Indicates that the mode of an expression is not absolute when it should be. May occur when a relocatable or external expresion is used to specify a quantity that must be either constant or representable in less than 16 bits.

25) UNDEFINED GLOBAL(S)
Indicates that one or more symbols which were declared in a GLOBAL pseudo-op were never actually defined as a label in this module. A cross-reference listing may be helpful in determining which GLOBALs are undefined (value=FFFF).

26) WARNING - ORG IS RELOCATABLE
Indicates that an ORG statement was encountered in a relocatable module. This warning is issued to remind the user that the reference counter is set to a relocatable value, not an absolute one. May occur when the Absolute option is not specified for an absolute module. This warning may be suppressed by the NOW option.

Zilog

10460 Bubb Road
Cupertino, California 95014
Telephone (408) 446-4666
TWX 910-338-7621

03-0073-00

Price: $7.50

Printed in USA