SOFTWARE MANUAL

AMOS

ASSEMBLY LANGUAGE PROGRAMMER'S MANUAL

DWM-00100-43

REV. B00

alpha micro

SOFTWARE MANUAL

AMOS
ASSEMBLY LANGUAGE PROGRAMMER'S MANUAL

DWM-00100-43

REV. B00

7006

alpha micro

First printing: April 1979
Second printing: 30 April 1981

This manual reflects AMOS Versions 4.5 and later

ALPHA MICROSYSTEMS
17881 Sky Park North
Irvine, CA 92714

PREFACE

This manual covers the procedures for writing assembly language programs for the Alpha Micro AM-100 and AM-100/T based computer systems. We also discuss the operation of the programs that make up the AMOS assembly program development system. We assume that you are familiar with assembly language programming techniques in general, and with the AM-100 machine instruction set in particular.

The WD16 Microcomputer Programmer's Reference Manual, (DWM-00100-04), describes the instruction set for the AM-100 and AM-100/T CPUs. For information concerning interfacing with AMOS via the AMOS monitor calls, refer to the AMOS Monitor Calls Manual, (DWM-00100-42).

NOTE: Because the AM-100 and the AM-100/T CPUs use the same instruction set, all references to "AM-100" in this manual also apply to the AM-100/T.

Table of Contents

CHAPTER 5 ASSEMBLER PSEUDO OPCODES

CHAPTER 6 USER DEFINED MACROS

CHAPTER 1

INTRODUCTION

The AM-100 and AM-100/T based computer systems support a flexible and efficient assembly language development system under the AMOS monitor. This system includes the assembler, linkage editor, symbol file generator, object file library generator, global symbol cross reference generator, and symbolic debugger programs.

The assembler is a multi-pass macro assembler with conditional assembly directives, library copy function, and external segment links. The linkage editor is used to link multi-segment programs together and to create a runnable program file. The operating system supports segment overlays thereby allowing a large program to be logically divided into smaller segments and executed sequentially. The debugger programs accept a specially created symbol file as input and allow the program to be traced and debugged in symbolic instructions using all the labels as they were entered in the source program. The library generator provides a mechanism for developing and maintaining a library file that contains frequently used routines, making them accessible to all programmers on the system. All components of the assembly language development system run under control of the standard AMOS monitor.

There currently exist over 70 monitor calls in macro form that the assembly language programmer uses to communicate with the AMOS monitor and to make use of the routines it has to offer. These macro calls are predefined in a file called SYS.MAC located in account [7,7] on the AMOS System Disk. The programmer uses a single COPY statement to include this complete library of predefined functions in his assembly language program and then refers to the monitor calls by their macro names; this makes for an easy-to-use communication link to the system resources. SYS.MAC also includes equate statements for many of the predefined system variables including the job table entries for the user's impure job variables.

If your programs are to be compatible with the AMOS system architecture, you must write them in totally relocatable code. A relocatable program may be loaded anywhere in RAM and executed without modifying any addresses within the program itself. There are machine instructions which assist in writing totally relocatable code, and by obeying a few simple restrictions the task of writing assembly language programs for the AM-100 and AM-100/T becomes almost foolproof.

Optionally, you may write programs which are re-entrant and then incorporate these programs or subroutines into system memory to be shared by all users without requiring a separate copy for each user. (To add programs to system memory, you must modify the system initialization command file. For information on the system initialization command file, see the "System Operator's Information" section of the AMOS Software Update Documentation Packet.)

We will not delve into the rules for re-entrant programming in great detail here since it is an advanced programming technique and requires specific rules that are not machine dependent. There are numerous books on the subject and all general practices apply to the programming of the Alpha Micro computer system. There are a number of features in the instruction set which do lend themselves quite nicely to writing re-entrant code, some of which are detailed in Chapter 8.

1.1 NOTE TO USERS OF PREVIOUS VERSIONS OF MACRO, LINK, SYMBOL AND DDT

If you are familiar with versions of MACRO, LINK, SYMBOL, and DDT that were released before AMOS Versions 4.5 and later, you would probably like a summary of what changes were made to these programs with AMOS Release 4.5. If you are new to the AMOS system, please skip on to Section 1.2, below.

THE OBJECT FILE LIBRARY

One of the most important changes made was the introduction of the new program LIB, the object file library generator. You can now use LIB to combine collections of .OBJ files into an object file library. Then when you use LINK or SYMBOL to link your program, you can optionally specify a library file from which routines will be linked into your program if your program references symbols in that library file. Besides generating new library files, you may update existing library files by deleting or replacing existing modules or adding new modules, and you may obtain a library listing file that tells you what object files are in a specific library. For more information on LIB and the use of library files, refer to Section 10.3, "Library and Optional Files," and Chapter 11, "The Object File Library Generator (LIB)."

LOCAL SYMBOLS

MACRO, DDT, and FIX now support the use of local symbols. A brief discussion of local symbols occurs in Section 4.7, "Local Symbols." For information on the use of local symbols within macro definitions, see

Section 6.1.5, "Local Symbols," and for a discussion on accessing local symbols through DDT and AlphaFIX, see Section 13.4, "Expressions."

CHANGES TO MACRO:

The macro assembler now gives a new assembly display which provides more information. (For example, if MACRO is automatically EXTERNing symbols, it lists those symbols alphabetically in Phase 2. For information on automatically EXTERNing undefined symbols, see AUTOEXTERN, below, in the section on Pseudo Opcodes.) If you forgot to end your file with an END statement, MACRO now tells you so.

MACRO supports two new option request switches that allow you to: 1) request a symbol cross reference listing; and, 2) use the parameterized assembly option.

The cross reference listing (which appears at the end of a regular assembly listing) contains an alphabetic list of all symbols, tells you which lines of your source program they appeared on. and whether the symbols are label definitions, equate definitions, are INTERNed, EXTERNed, or are overlays. The listing also tells you which symbols were never defined. The cross reference then gives a similar listing for all macro definitions and references. For information on the MACRO cross reference, see Section 9.4.3, "Generating a Cross Reference."

The parameterized assembly option allows you to specify a value at the time you assemble your program which your program can analyze. This feature is very useful when used with the conditional assembly directive pseudo opcodes. For more information, see Section 9.2.3, "The Parameterized Assembly Option."

LINK and SYMBOL

Both LINK and SYMBOL have changed quite a bit. They both now support a number of option request switches. By combining these switches, LINK and SYMBOL can be made to perform the same functions. (For example, LINK can generate a symbol table file, and SYMBOL can generate a resolved program file.)

LINK and SYMBOL both support library files and optional files.

The LINK options are:

Designate a file as a library file.
Designate a file as an optional file.
Designate a file as a required file (the default).
Generate a load map file.
Generate a symbol table file.
Include equated symbols in the symbol table file.
Generate a program file (the default).
Suppress program generation.

NOTE: An "optional file" contains only one .OBJ file, and is linked in only if references are made by your program to symbols in that file. For information on optional files, see Section 10.3, "Library and Optional Files." A load map file contains a map of how the linked together items will be loaded into memory when you execute the program file. It also contains additional information on each item. See Section 10.4, "The Load Map File," for more information.

The SYMBOL options are:

```
Designate a file as a library file.
Designate a file as an optional file.
Designate a file as a required file (the default).
Generate a load map file.
Generate a symbol table file (the default).
Include equated symbols in the symbol table file.
Generate a program file.
Suppress symbol table file generation.
```

GLOBAL

GLOBAL generates a global symbol cross reference for a collection of .OBJ files. This listing tells you which files the symbols were defined in and which files the symbols were referenced in. (NOTE: This differs from the MACRO cross reference in that GLOBAL is meant to be used for a collection of .OBJ files to determine the symbol references between those files; the MACRO cross reference gives detailed information on the symbols within a single file.) See Chapter 12, "The Global Cross Reference Generator (GLOBAL)," for more information.

PSEUDO OPCODES

This manual now documents the search pattern MACRO uses in looking for the copy file specified by the COPY pseudo opcode. Please see Section 5.1.1, "COPY."

Several new pseudo opcodes have been added:

OBJNAM - Allows you to modify the name and extension given to the output files created by MACRO, LINK, and SYMBOL.

LIST, NOLIST - Allow you to suspend and re-enable output to the assembly listing.

CREF, NOCREF, MAYCREF - Allow you to suspend and re-enable output to the cross reference portion of the assembly listing.

NVALU - Allows your program to make use of the value supplied on the MACRO command line via the /V parameterized assembly option switch.

AUTOEXTERN - Tells MACRO to automatically EXTERN any undefined symbols.

ENDMX - Terminates macro expansion.

You may find information on all of these pseudo opcodes except ENDMX by referring to Chapter 5, "Assembler Pseudo Opcodes." For information on ENDMX, see Section 6.1.8, "Suppressing Macro Expansion - ENDMX."

FILES

Several new files are now created by the AMOS assembly language system:

.LIB files - Library files generated by LIB.
.GLB files - Global cross reference listing created by GLOBAL.
.MAP files - Load map files generated by LINK and SYMBOL.
.TMP files - Temporary work file generated by LIB.

OTHER FEATURES:

This manual contains information on two previously undocumented operators:

The expression evaluation operator, \, for use within macro definitions (see Section 6.1.7, "Special Macro Operators"); and,

The binary shift operator, (underscore); see Section 4.3, "Expressions."

This book also now includes two Appendices: "Appendix A, The ASCII Character Set," and "Appendix B, Summary of Program Switches."

1.2 THE CONTENTS OF THIS MANUAL

Part I - INTRODUCTION TO ALPHA MICRO ASSEMBLY LANGUAGE PROGRAMMING

Chapters 2 through 8 contain information on the form of your assembly language programs. For example, Chapter 4 discusses labels, terms, and expressions in your assembly language program statements. Chapter 5 discusses the pseudo opcodes available to you, and Chapter 6 discusses how to construct and call macros.

Part II - USING THE ALPHA MICRO ASSEMBLY LANGUAGE PROGRAMMING SYSTEM

Chapters 9 through 13 give operating information for the various components of the Alpha Micro assembly language programming system:

MACRO - The macro-assembler
LINK - The linkage editor
SYMBOL - The symbol table generator
LIB - The object file library generator
GLOBAL - The global cross reference generator
DDT - The dynamic debugging and patching program

Appendix A gives the complete ASCII character set, with values specified in decimal, octal, and hexadecimal. Appendix B gives a brief summary of all option request switches used by MACRO, LINK, SYMBOL, LIB, and GLOBAL.

1.3 READER'S COMMENTS FORM

Please note the Reader's Comment Form at the back of this manual. We would very much appreciate any comments or criticisms you may have concerning this book. Any suggestions for future documentation projects are also welcome.

1.4 CONVENTIONS USED IN THIS MANUAL

To make our examples concise and easy to understand, we've adopted a number of graphics conventions throughout our manuals:

Number Base	Unless otherwise noted, all numbers are decimal (base 10).
PPN	A Project-programmer number. This number identifies a user disk account (e.g., [100,2]). We also represent an account number as [p,pn].
Filespec	A file specification. Identifies a file. It usually has the elements: Devn:Filename.Ext[p,pn] where "Devn:" is a device specification that identifies a logical unit of a physical device, "filename" gives the name of the file, and "ext" specifies the file's extension.
{}	Optional elements of a command line. When these symbols appear in a sample command line, they designate elements that you may omit from the command line.
_____	Underlined characters indicate those characters that AMOS prints on your terminal display. For example, in the latter chapters of this manual you may see an underlined dot, .., which indicates the AMOS monitor prompt symbol.
RET	Carriage return symbol. This symbol marks the place in your keyboard entry to press the RETURN key.
^	Indicates a Control-character. For example, if you type a Control-C, you see it echoed on your terminal as ^C.
$	Escape symbol. This symbol marks the place in your keyboard entry to press the ESCAPE key (sometimes labeled ALT MODE or ESC).

PART I

INTRODUCTION TO ALPHA MICRO ASSEMBLY LANGUAGE PROGRAMMING

These chapters introduce the experienced assembly language programmer to assembly language programming for the AM-100 and AM-100/T based computer systems.

CHAPTER 2

FILES USED IN THE ASSEMBLY LANGUAGE SYSTEM

This section describes the files that are used during the normal course of building and testing an assembly language program. We will refer to these files by their extensions; i.e., a .MAC file is any file with an extension of "MAC". All files described here will not necessarily be used by all programmers during any one programming session, but you will eventually run across all of them at one time or another so you might as well know briefly what they are used for and how they are created.

2.1 .MAC - SOURCE FILES

.MAC files are the original ASCII source files that you create using the EDIT or VUE program. .MAC files are input files for the assembler program (MACRO) which makes one or more passes over them depending on the assembly options selected. If you want to make any changes to a program, you make the changes to the .MAC file by using the EDIT or VUE program; you then reassemble and relink it. Files that you include with the COPY assembly pseudo opcode must also be ASCII source files with an extension of .MAC.

2.2 .OBJ - INTERMEDIATE OBJECT FILES

.OBJ files are the direct output of the assembler (Phase 2) and contain the assembled binary code, symbol references, internal symbol definitions, and unresolved external symbol references. .OBJ files are not directly usable for anything by themselves but must first be processed by one or more of several other programs to get a finished file that has a direct use by itself. The linkage editor program (LINK) reads one or more .OBJ files and creates a fully resolved and runnable binary program file in memory image format. The library generating program (LIB) combines specified .OBJ files into an object file library. The GLOBAL program reads .OBJ files and creates a global symbol cross reference file. The symbol file program (SYMBOL) reads the .OBJ files and creates a file which contains all user defined symbols and their resolved addresses. (This symbol table file is used by the symbolic debugger programs DDT and FIX.) The assembler itself

also rereads the .OBJ file during Phase 3 together with the .MAC source file to create the ASCII list file.

2.3 .PRG - BINARY PROGRAM FILES

.PRG files are created by the linkage editor program (LINK) and are the end result of the assembly process. The .PRG file is a binary memory image of the assembled program which is loaded into user RAM when the program is requested for execution. (That is, the .PRG file is the final, fully assembled and resolved machine language program of which the .MAC file was the source.)

The .MAC file from which the .PRG file was generated must have been written using the rules for totally relocatable code so that the .PRG file may be dumped into any memory location and executed without modification. One or more .OBJ files may have been input to the linker for the creation of the single .PRG file. Once you have tested the .PRG program file, you may place it into the System Library Account, DSK0:[1,4], where it will become available to all users on the system.

2.4 .OVR - BINARY OVERLAY FILES

If the program contains overlay segments which do not all reside in memory at the same time, the linkage editor generates one .PRG main segment file and one or more .OVR overlay segment files. LINK generates each overlay file in response to an OVRLAY assembler pseudo opcode. The .PRG program segment will be responsible for the calling and executing of each of the other .OVR segments during the running of the program. Your program may selectively bypass overlay segments as does the assembler itself, which contains six overlays. Overlay files have the same memory image format as the .PRG program files except that they are resolved at an effective address other than zero so that they will not completely overlay the controlling segment. This addressing is the direct responsibility of the programmer; for more information on creating overlays, see Section 5.3.5, "OVRLAY."

2.5 .LST - PROGRAM LISTING FILES

An optional output of the assembler is a complete resolved listing of the source program with the associated binary code that was generated. MACRO creates this list file during Phase 3 of the assembly process; you may generate it directly from the .MAC and .OBJ files by bypassing Phases 1 and 2 with the /O assembly switch. The .LST file is formated ASCII; you may display it via the TYPE command or examine it by either the EDIT or VUE programs. Or, you may print the list file using the PRINT command.

The .LST file may optionally contain a full symbol cross reference if you use the /R assembly switch. (See Section 9.2 for information on the /O and /R MACRO switches.)

2.6 .LIB - LIBRARY FILES

The .LIB file is a library file. (A library file contains a collection of .OBJ files that are linked into the main program as required.) The LIB program allows you to generate and maintain object file libraries. The LINK and SYMBOL programs accept these library (.LIB) files as input and automatically include any object files from such a library necessary to resolve external references. See Chapter 11 for information on creating and maintaining program libraries.

2.7 .GLB - GLOBAL CROSS REFERENCE FILE

The GLOBAL program reads a group of .OBJ files and creates an alphabetic cross reference .GLB file that lists all global symbols in the files, and shows which files define them and which files accept them as externally defined symbols. (For information on GLOBAL, see Chapter 12.)

2.8 .MAP - LOAD MAP FILE

Both the linkage editor LINK and the symbol table file generator SYMBOL generate a load map file in response to the optional /M switch. The load map (.MAP) file shows how the assembled and linked object files will be located in memory when the program is loaded into memory prior to execution. It also gives information about each object file linked into the final .PRG file. For information on the load map, see Section 10.4, "The Load Map File."

2.9 .SYM - RESOLVED SYMBOL FILES

The .SYM file is a direct output of the symbol file generation program (SYMBOL) which takes one or more object (.OBJ) files and creates a symbol table with all user defined symbols and their resolved machine addresses. The .SYM file is used as input to the debugger programs DDT and FIX which may then operate with references to the user symbols in the program instead of absolute machine addresses. In a system where the program is always offset by some amount in memory, this is almost essential if you are to be able to trace the execution flow of a program under test. The .SYM file is in a special packed binary form and, as such, is not much good for anything except input to DDT and FIX. (NOTE: The LINK program can also generate a .SYM symbol table file.)

2.10 .IPF - INTER-PHASE WORK FILE

The .IPF file is a temporary work file built during the assembly process by Phase 1 of the assembler to carry information on to Phase 2. The .IPF file is packed binary junk and the only reason we mention it here is that if the system crashes during an assembly you may find one left on your disk. Erase it; it is useless and just takes up space. There is no problem if it exists and you don't find it, since the next assembly of the same program will erase any .IPF file it finds during Phase 1 before attempting to create a new one.

2.11 .TMP - TEMPORARY WORK FILES

The LIB program creates a temporary work file named Jobnam.TMP ("Jobnam" is the name of your job). As with the .IPF file, you should never see this file unless something goes wrong. The next time you run LIB, the .TMP file should disappear.

CHAPTER 3

MACRO SOURCE PROGRAM FORMAT

A macro source program is a single .MAC file composed of a sequence of ASCII source statement lines. Each line must be complete in itself since there is no provision for multiple-line statements. Each statement may be one of the following, depending on its function:

1. Valid machine instruction
2. Data generation statement
3. Symbolic equate statement
4. Assembly control statement
5. Conditional assembly directive
6. Macro definition
7. Macro call
8. Comment or blank line

The maximum line length is 100 characters. Each line is terminated by a carriage-return and line-feed pair which the editor provides when you press the RETURN key. Unless otherwise specified, all of the above lines may contain an optional comment field following the actual statement; this comment field starts with a semicolon (;) and extends to the end of the line The assembler treats spaces and tabs (Control-I) as equal; they are used to delimit fields within statements. Tabs are useful to keep statement fields aligned and make for clean listings. Tabs are an important part of generating readable code.

NOTE: This manual refers to the term "user symbol" several times during later discussions, so we will define it at this point. A user symbol is any name defined by you within your program. It must be unique to that program, and must be from 1-6 characters in length. Legal characters for a user symbol include the alphabetic characters A-Z, the numeric characters 0-9, and the two special symbols "." and "$". The first character of a user symbol must be non-numeric. MACRO folds all lower case characters to upper case. Symbols are packed RAD50 and stored as two words in the symbol table during the assembly process along with their current assigned value and attribute flags.

3.1 MACHINE INSTRUCTIONS

One machine statement is allowed per line and is assembled into a single machine hardware instruction which generates one, two, or three words of binary code depending on the instruction and addressing modes used. The general format of a machine instruction statement is:

{label:} {opcode} {operands} {;comments}

The label field is optional and is used to give a symbolic name to the current instruction being assembled. It must terminate with a colon. The label may be any valid user symbol that has not been previously defined. The value of the label may be either absolute or relocatable depending on the current assembly status. Relocatable symbols will be resolved during link-edit time by adding the label value to the current program relocation bias (calculated by LINK). More than one label may appear on the same statement line separated by colons; in this case, each label is given the same value as the current location. Any symbol used in a label field may not be redefined later in the program. A label may appear as the only item on a line in which case it is assigned the address of the next byte of generated code.

The opcode field is required and contains one of the machine instruction opcodes in mnemonic form such as MOV, CLRB, TST, ADD, etc. (Refer to the WD16 Microcomputer Programmer's Reference Manual, (DWM-00100-04), for a complete description of all the machine instructions available in the AM-100 system.) The opcode field terminates with a space, tab, semicolon or carriage-return. If a label field was used, a space or tab between the colon and the opcode is optional but recommended.

The operands field is required on those instructions that have either one or two operands. The operands field is separated from the opcode field by one or more spaces or tabs. If the instruction being used requires two operands, the operands are separated from each other by a comma. Leading spaces are always ignored in the operands field while the operands themselves terminate with a space, tab, comma, semicolon or carriage-return.

The comments field is optional and is defined by a leading semicolon. The comments field then extends through the remainder of the line up to the carriage-return. Any valid ASCII characters are legal in the comments field.

3.2 DATA GENERATION STATEMENTS

Data generation statements resemble machine instructions in format and generate binary data within the program flow. The data generated is normally not interpreted during program run as executable instructions but rather as constant data such as ASCII messages to be typed or numeric values to be used by those instruction being executed. The general format of the data generation statement is:

{label:} {operator} {operands} {;comments}

The label field is optional and follows the same format and rules as the machine instruction label field. The operator field contains the specific data generation mnemonic for the type of data desired. We discuss these codes in Section 5.2, "Data Generation Pseudo Opcodes." The operands field contains the actual data to be generated by the statement and its format depends on the type of operator in use. Some operators such as WORD and BYTE allow multiple operands within the same statement so that the amount of binary data generated by the one statement is variable. If a label is used, its value is always that of the address into which the first byte of data will be assembled. As with machine instructions, the comments field is optional.

There is a special default type of data generation statement which you should be aware of. If no operator is present, MACRO assumes the statement is a WORD statement and it interprets the operands field as such. The assembler works in the following manner when analyzing statements:

1. Leading symbols terminated by colons are processed as labels and stored in the assembler symbol table.

2. The next symbol is first scanned for a match in the macro table which consists of all macros previously defined in the program.

3. If the operator symbol is not a macro name, it is then matched against the table of machine instruction opcodes, data generation operators, and assembly control pseudo opcodes.

4. If none of the above result in a defined operator, the default WORD processor is entered and the symbol is assumed to be the beginning of the associated operands field for the WORD statement.

3.3 SYMBOLIC EQUATE STATEMENTS

A user symbol may be assigned a value by entering it on a statement line followed by an equal-sign (=) and the expression to which it is to be equated. The general format of the equate statement is:

{user symbol} = {expression} {;comments}

The equal-sign may have leading or trailing spaces and tabs if desired for formatting purposes. The expression may be any valid numeric expression but since all equate statements must be fully resolved during Phase 1, any user symbols used in the expression must be defined at the time that the equate statement is encountered. Equate statements may not contain references to external symbols. The comments field is optional as in the machine instruction statement.

User symbols that are assigned values in the program may be reassigned a different value later in the program by using another equate statement to redefine the desired symbol. Labels may not be redefined by equate statements, however. If the relocation attribute of the evaluated expression is zero, the value assigned to the symbol is absolute. If the relocation attribute is non-zero, then the value assigned is relocatable. If the expression contains a register symbol, then the equated symbol is also given a register attribute. In other words, the value assigned to the user symbol pretty much follows the attributes of the expression to which it is equated.

3.4 ASSEMBLY CONTROL STATEMENTS

Assembly control statements cover a wide range of functions that generally set up or alter the parameters which control the assembly process. They do not themselves generate any binary code but are used for such purposes as listing format control, numeric radix assignment, and program generation or addressing information. The general format for assembly control statements is:

{pseudo-opcode} {arguments} {;comments}

The pseudo opcode is the mnemonic that defines the function to be performed. Chapter 5 lists all pseudo opcodes along with an explanation of what each one does. Some of them require arguments that are needed to set up parameters. These arguments are separated from the pseudo opcode by one or more spaces or tabs. As in other statement formats, the comments field is optional. Unless the explanation in Chapter 5 for a pseudo opcode specifies otherwise, labels are not normally permitted in assembly control statements.

3.5 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly is defined as selectively assembling or bypassing statements within defined bounds depending on the value of some variable at the time the assembly is performed. The bounds are made by conditional assembly directives which specify the variable or variables to be tested and the condition to be met in order for the assembly to occur. Conditional assembly directives are most commonly used in conjunction with macro definitions to direct the tailoring of each macro call as it is encountered. We discuss conditional assembly directives in Chapter 7.

3.6 MACRO DEFINITIONS AND MACRO CALLS

Macros are defined as one or more valid statements which may be called for by using a single symbol (the macro name) within the program anytime after the macro has been defined. Macros are always defined by you within your program or within a copy file which is called into your program by the COPY

statement. The copy file called SYS.MAC is a macro library of over 70 such macro definitions which define all the supervisor calls available to your programs for communicating with the monitor routines. This library file is supplied on the AMOS System Disk in account [7,7].

Macro calls are those statements which name the defined macro as the operator of the statement and give the specific arguments to be used by the macro (if any are required). A macro call within the program causes the defined macro to be included in its tailored form at the point of the call. Macro calls normally cause one or more machine instructions to be assembled and the respective binary code to be generated.

Chapter 6 defines macro definitions and macro calls more fully.

3.7 COMMENT LINES AND BLANK LINES

Statements which begin with a semicolon (after any leading spaces and tabs) are considered comment lines and do not result in the generation of any binary code or in the alteration of any assembly control parameters. They are useful only for documenting the source programs and making them easier to read and maintain. Blank lines are also considered comment lines and are for appearances only in the source file. It is most important to fully document your programs, so use comments liberally.

CHAPTER 4

TERMS AND EXPRESSIONS

This section describes the various terms and components used in MACRO source statements, including the defined character set for the construction of symbols and expressions.

4.1 CHARACTER SET

The entire ASCII character set is legal in MACRO source programs except for the control-characters. MACRO translates lower case characters to upper case before it checks the syntax of each source line. The characters that are valid in user defined symbols are limited to A-Z, 0-9, "$" and "." because symbols are packed RAD50 before being stored in the symbol table. The folowing list gives the special characters that are recognized by the assembler when scanning source lines:

:	Label terminator
;	Comment field indicator
=	Equate statement operator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminating register indicator
,	Operand field or macro argument separator
.	Value of the assembly current location counter when used as a term
<	Initial argument or expression indicator
>	Terminating argument or expression indicator
+	Arithmetic addition operator or autoincrement mode indicator
-	Arithmetic subtraction operator or autodecrement mode indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical inclusive OR operator

'	Single ASCII character term indicator
"	Double ASCII character term indicator
[	Initial RAD50 triplet term indicator
]	Terminating RAD50 triplet term indicator
^	Universal unary indicator
_	(Underscore) Binary shift operator

The use of the above legal characters out of context for their designed purposes will cause the generation of a syntax error (code Q).

4.2 TERMS

A term is the basic unit of data in an arithmetic expression and may be one of the following:

1. A number as composed of legal digits within the current radix of the system or as temporarily defined by the inclusion of a leading temporary radix change operator;

2. A user symbol (as previously defined) which is given an assigned value either by its use as a label or a direct equate statement;

3. An ASCII conversion defined by the single or double quote indicators;

4. A RAD50 triplet enclosed in square brackets;

5. The period symbol (.) which represents the current value of the assembly current location counter;

6. An expression or term enclosed within angle brackets. Angle brackets are used to alter the normal hierarchy of expression evaluation which is normally done in a left-to-right manner. Any quantity enclosed within angle brackets will be evaluated before the remainder of the expression in which it is found. The action of angle brackets within a MACRO source expression is the same as that of parentheses within a normal arithmetic expression such as is used in the BASIC language. Angle brackets may also be used to apply a unary operator to an entire expression such as -<16/A>.

4.3 EXPRESSIONS

An expression is a combination of terms and operators which will evaluate to an unsigned 16-bit value in the decimal range of 0-65535. Negative values in the range of -32768 through -1 will be stored properly after evaluation but will be treated the same as their unsigned counterparts in the range of 32768 through 65535.

The evaluation of any expression also includes the evaluation of the mode of that expression (absolute, relocatable, and external) and the register designation of the expression.

Operators are defined as unary or binary. Unary operators precede a single term and alter the evaluation of that term alone. Multiple unary operators may be applied in sequence to the same term and are evaluated in reverse order. Binary operators combine two terms to give a resultant effective single term value. Multiple binary operators are illegal.

Expressions are evaluated left to right under the hierarchy of the operators which are in use within that expression. Angle brackets may be used to alter the normal process of evaluation. Unary operators always take precedence over binary operators and are applied to the associated terms first.

The legal operators are:

+	Unary plus sign (default if term not preceded by another unary)
-	Unary minus sign which negates the associated term value
^C	Unary one's complement operator (XOR's the term with all ones)
^D	Temporary radix change to decimal for the associated term
^B	Temporary radix change to binary for the associated term
^O	Temporary radix change to octal for the associated term
^H	Temporary radix change to hexadecimal for the associated term
+	Binary addition operator
-	Binary subtraction operator
*	Binary multiplication operator
/	Binary division operator
_	(Underscore.) Binary shift operator (given A_B, binary representation of A is shifted B number of times. If B is positive, shifts A left; if B is negative, shifts A right.)
&	Binary logical AND operator
!	Binary logical inclusive OR operator

NOTE: Two special operators (\ and ') also exist for use within macro definitions. See Section 6.1.7, "Special Macro Operators," for more information.

Expressions are evaluated as being absolute, relocatable, or external. This distinction becomes particularly important since we are writing totally relocatable code for the AM-100 system. The following rules apply in the evaluation of the relocation attribute of an expression:

1. An expression is absolute if its value is fixed and contains no relocatable terms. Also, a relocatable term minus another relocatable term results in an absolute value. Labels allocated within an absolute section (ASECT) will be assigned absolute values and attributes.

2. An expression is relocatable if its value is fixed relative to the current program base which is relocatable at load time. The value may have an offset added to it by LINK if it is not within the

first segment of a program file. Labels allocated within a relocatable section (RSECT) will be assigned relocatable values and attributes. (For information on the ASECT and RSECT pseudo opcodes, see Section 5.1.5.)

3. An expression is defined as external when one or more of its terms is an external symbol reference. This expression will not be fully resolved until the program file is generated by the linkage editor (LINK) when the external terms are defined. The final resolution of an external expression may be relocatable or absolute, depending on the attributes of the terms involved (both internal and external). The linkage editor also contains all the mechanics for evaluating the attributes of resolved expressions. (See Section 5.3, "Segmentation Pseudo Opcodes," for information on the EXTERN, INTERN, and AUTOEXTERN pseudo opcodes.)

4.4 NUMBERS

Any source item which starts with a digit (0-9) is considered to be a number and this number will be evaluated under the currently prevailing radix unless preceded by a temporary radix operator or followed immediately by a decimal point. The prevailing radix always starts as octal (base 8) at the beginning of any assembly but may be changed by the RADIX assembly control statement. Any number that terminates with a decimal point will be evaluated as decimal (base 10) regardless of the prevailing radix. Fractional numbers are not allowed in MACRO source statements since all numbers must evaluate to a 16-bit binary integer value.

The prevailing radix controls the default evaluation of numbers and may be set by the RADIX statement to any value from 2 (binary) through 36. Numbers in a base above 10 (decimal) use the alphabetic characters A-Z to represent the digit values of 10 through 35. The most common system above base 10 is hexadecimal where the letters A-F represent the decimal digit values 10-15. All numbers must begin with a digit 0-9 to distinguish them from a user symbol, so the hexadecimal value of F56 must be entered as 0F56.

Negative numbers are preceded by a minus sign; MACRO evaluates them and stores them in two's complement form. You may optionally precede positive numbers with a plus sign but this is not required.

4.5 REGISTER SYMBOLS

The WD16 chipset (the heart of the AM-100 and AM-100/T systems) contains eight 16-bit registers which are symbolically named and used as follows:

R0 - register 0, general purpose
R1 - register 1, general purpose
R2 - register 2, general purpose
R3 - register 3, general purpose
R4 - register 4, general purpose
R5 - register 5, general purpose
SP - register 6, stack pointer
PC - register 7, program counter

These eight symbols are already defined to the assembler and must be used when the address mode explicitly requires a register to be referenced. The above register symbols have a register attribute associated with them and you may equate your own symbols to these registers if you so desire. The register attribute will be carried over to this newly defined symbol. For example, the equate statement IOPTR=R4 will equate the user symbol IOPTR to the value of 4 and also give it a register attribute so that it may be used in place of R4 for address modes.

4.6 ASSEMBLY LOCATION COUNTER

During the assembly process, MACRO assigns sequential memory locations to all machine instructions and data constants as it encounters them in the source program. At any given statement, the next byte to be assigned will be internally stored in the assembly location counter. This address may be used in expressions by referencing the period (.) as a symbolic term. For example, the instruction "JMP .+6" will cause a jump to the address which is 6 bytes in front of the first byte of the instruction itself.

The assembly location counter has an attribute associated with it which is either absolute or relocatable. Initially, it is set up in the relocatable mode and cleared to zero value for the allocation of relocatable binary code as machine instructions and data constants are assembled. If MACRO encounters an ASECT statement, MACRO changes the attribute of the assembly location counter to absolute which means the address associated with it will not be adjusted by the LINK program. If MACRO encounters an RSECT statement, MACRO sets the attribute back to relocatable again which means that the address associated with it will be adjusted by the LINK program to compensate for the program segment offset. The assembler also maintains two separate address counters for switching between ASECT and RSECT sections.

Initially, the value of the assembly location counter is set to zero and is incremented as each statement which produces binary code is assembled during Phase 1. You may explicitly change the setting of the assembly location counter at any time by using a direct equate statement that uses the period symbol instead of a user symbol. For example, the statement ".=500" forces the assembly location counter to take on a value of 500 and to begin all assembly allocation from that point.

4.7 LOCAL SYMBOLS

MACRO supports local symbols of the form nnn$, where nnn may be any number from 0 through 65535, decimal. A program using local symbols will require less symbol table space and will assemble faster than a similar program without local symbols.

(NOTE: Local symbols of the form nnn$$ are used within macros and have scope within a particular macro expansion. For information on this kind of local symbol, see Section 6.1.5, "Local Symbols.")

A local symbol only has scope between two non-local symbols. For example:

```
SEND:   MOVB    (R0)+,R1
        BEQ     1$
        TTY
        BR      SEND
1$:     RTN

RCV:    KBD
        LEA     R0,BUF
1$:     MOVB    (R2)+,(R0)+
        BNE     1$
        RTN

SUBR:   ...
```

1$ is defined twice in the program above. The first 1$ has a range from the definition of SEND up to but not including the definition of RCV. The second 1$ has a range from RCV up to SUBR.

NOTE: You may also define local symbols with an equate (=).

CHAPTER 5

ASSEMBLER PSEUDO OPCODES

A pseudo opcode is so named because although it looks much like a regular operation code, a pseudo opcode is not a true machine instruction and may or may not generate actual binary code. Pseudo opcodes are built into the assembler and provide a variety of useful functions that make the life of the programmer easier.

This chapter discusses the MACRO pseudo opcodes available for your use. We classify the functions of the pseudo opcodes into four categories: 1) assembly control; 2) data generation; 3) segmentation; and, 4) convenience. The sections below discuss each of these types of pseudo opcodes.

Note that other chapters discuss several other pseudo opcodes that are used in special circumstances. For example, Chapter 6, "User Defined Macros," discusses the pseudo opcodes you can use inside of macro definitions. For a full list of all pseudo opcodes, refer to the index.

5.1 ASSEMBLY CONTROL PSEUDO OPCODES

Assembly control statements perform a wide variety of functions which do not in themselves generate any binary code but, instead, set up or alter certain parameters which control the assembly process. Each statement consists of a defined assembly control pseudo opcode followed by optional arguments as required by the specific format. These pseudo opcodes are described here along with the required arguments for each.

5.1.1 COPY

The COPY statement allows another file to be included in the assembled program at the point where the COPY statement is located. The entire copied file is assembled, but you may use conditional assembly statements to omit certain portions if desired. The most common use of this statement is for the inclusion of the standard copy file SYS.MAC which defines all system call macros and system parameters. (The SYS.MAC file is in account

DSK0:[7,7].) The COPY statement includes a file specification that specifies the file that is to be copied into the source program during assembly. For example:

```
COPY DEF        ; My own set of macro definitions in the file DEF.MAC.
```

Note that the actual source program is not modified; rather, the assembler merely gets the input from the copied file and then returns to the original source file as it assembles the source file. A copy file may not include another COPY statement within itself although the original file may include as many individual COPY statements as desired. The filespec may actually be a complete file specification containing a device and account specification. If you do not specify an extension, MACRO uses the default extension of .MAC.

If you specify both a device and account, MACRO looks for the copy file in the specified device and account. However, if you omit either a device or an account specification, MACRO goes through several steps in trying to find the specified file:

If you omit both the device and the account specification:

1. MACRO looks for the file in the device and account you are logged into.

2. If the file does not exist in that account and if the source file is on a different device than the one you are logged into, MACRO looks in the account you are logged into on the device containing the source file.

3. If the file does not exist in that account either, and if the source file is in a different account and device than the ones you are logged into, MACRO looks in the account and device of the source file.

4. Finally, MACRO looks in the System MACRO account, DSK0:[7,7].

If you omit just the device specification:

1. MACRO looks in the specified account on the device containing the source file.

2. If the file does not exist in that account, MACRO looks in the specified account on the device you are logged into.

3. Finally, if the account specified is [7,7], MACRO looks in the System MACRO account, DSK0:[7,7].

If you omit just the account specification:

1. MACRO looks in the account containing the source file on the specified device.

2. If the file does not exist in that account, and if the source file is in a different account than the one you are logged into, MACRO looks on the specified device in the account you are logged into.

3. Finally, MACRO looks in the System MACRO account, DSK0:[7,7].

You may find it convenient to place copy files into the System MACRO account, DSK0:[7,7], since they will then become available to all programmers through the COPY statement.

MACRO does not normally output the source statements in the copied file during the listing phase of the assembly since most users do not want the system copy file (SYS.MAC) and other collections of common routines to be repeated in all program listings. You may override this by using a /L switch following the filespec in the statement; this will cause the copied file to be included in the assembly listing. For example:

```
COPY MYMAC.MAC/L
```

As it assembles your program, MACRO reports any COPY statements encountered. For example:

```
Copying from DSK0:SYS.MAC[7,7]
```

5.1.2 OBJNAM

The OBJNAM pseudo opcode controls the names of output files produced by LINK, SYMBOL, and MACRO. It tells these programs how you want to modify the output file name and extension. If you do not use OBJNAM, MACRO, LINK, and SYMBOL produce an output file with the same name as the input file and the appropriate extension.

The OBJNAM statement takes the form:

```
OBJNAM filnam.ext
```

or:

```
OBJNAM expr1{,...exprN}
```

where 1<=N<=3. That is, OBJNAM is followed by a filename and extension or by one to three expressions. If OBJNAM takes the second form, each expression is either 0 or a RAD50 value. The first expression denotes the first three characters of the filename, the second expresson denotes the last three characters of the filename, and the third expression denotes the three characters of the file extension.

OBJNAM causes the output file names to be modified as follows (where you have specified "file" and "ext" in the OBJNAM statement line):

```
source.OBJ      --->      file.OBJ
source.PRG      --->      file.ext
source.OVR      --->      file.ext
source.LST      --->      file.LST
source.MAP      --->      file.MAP
source.SYM      --->      file.SYM
```

If you omit "ext" or if any expression is omitted or is zero, the corresponding portion of the file name remains unmodified. For example, if you were assembling DEVCPY.MAC, and specified the OBJNAM statement:

```
OBJNAM TEST
```

(omitting the extension), the assembled and linked output file would have the name:

```
TEST.PRG
```

5.1.3 PAGE

The PAGE statement causes your assembly listing to begin a new page before continuing with the listed output. No action takes place other than this during assembly.

5.1.4 LIST - NOLIST

You may obtain an assembly listing by using the /L assembly switch. The LIST and NOLIST pseudo opcodes control which portions of your program will appear in the listing file. NOLIST disables listing, and LIST re-enables listing. The LIST and NOLIST pseudo opcodes do not appear in the listing. NOTE: MACRO will ignore the LIST and NOLIST pseudo opcodes if you use the optional /X assembly switch.

5.1.5 ASECT - RSECT

The ASECT statement causes the assembler to generate code for the absolute section of the program. This code will not be modified during LINK editing and the values assigned to labels will not have the relocatable attribute flag set.

The RSECT statement causes the assembler to generate code for the relocatable section of the program. This is the normal section for the AM-100 and AM-100/T systems which always relocates the program in user memory. This code will be modified during LINK editing and the values assigned to labels will have the relocatable attribute flag set. Two separate assembly location counters are maintained during program assembly.

5.1.6 SYM - NOSYM

The SYM statement causes all following user symbols to be output to the object file along with their assigned values. The NOSYM inhibits this output for all following user symbols. These symbols are later used by the SYMBOL program to generate a reference file for the dynamic debuggger programs DDT and FIX. The use of SYM and NOSYM does not cause any noticeable change in the actual program.

5.1.7 CREF - NOCREF - MAYCREF

To obtain a full cross reference listing, you may specify the /R assembly switch. (To see the cross reference listing on your terminal, specify the /RT switch.)

The three pseudo opcodes CREF, NOCREF, and MAYCREF control which portions of your program will be processed in creating the cross reference.

CREF enables normal cross referencing.

NOCREF suppresses from the cross reference listing all defined symbols until MACRO encounters a CREF or MAYCREF statement.

MAYCREF tells MACRO to suppress all symbols defined from the cross reference listing if those symbols are never referenced.

For a full discussion of the format of the cross reference listing, see Section 9.4.3, "Generating a Cross Reference."

5.1.8 EVEN

The EVEN statement forces the next binary code to be generated on a word boundary (next even byte) by incrementing the assembly location counter if it is odd (no change if it is even). This is necessary since all instructions must lie on a word boundary for proper execution by the AM-100 system.

5.1.9 RADIX

The RADIX statement forces a new default radix to be set up in the assembler. The default radix of the system determines how all numbers that are not preceded by a temporary radix operator (^B,^D,^H,^O) will be interpreted. The statement takes the form:

```
RADIX n
```

where the radix change argument "n" must be a decimal number in the range of 2-36. Radix values above 10 use the letters A-Z to represent the digit values of 10-35 inclusively. The default radix of all assemblies is base 8 (octal) in the absence of any explicit RADIX statement.

5.1.10 NVALU

MACRO provides a parameterized assembly facility by allowing you to use the /V switch to specify a value on the MACRO command line. The value switch may take one of these forms:

/V:x	x is an octal or hex number (depending on the prevailing radix setting)
/VO:x	x is an octal number
/VH:x	x is a hexadecimal number
/VD:x	x is a decimal number
/VA:x	x is one or two ASCII characters
/VR:x	x is one to three RAD50 characters

The NVALU pseudo opcode allows your program to access the value specified in the /V assembly switch. The NVALU statement takes the form:

```
NVALU sym
```

which sets the symbol "sym" to one of the values below, depending on which /V switch was used:

```
sym=x
sym=^Ox
sym=^HOx
sym=^Dx
sym='x
sym="x
sym=[x]
```

5.1.11 END

The END statement terminates the source file and is included only to give a defined end on the listing. In the absence of an END statement, the assembly will terminate with the logical end of input file. Note that if an END statement is encountered anywhere in the source input (including inside a copied file) the assembly will terminate whether the logical end of the input file has been reached or not.

NOTE: As it assembles your program, MACRO warns you if your program file does not contain an END statement:

```
Phase 1:    Missing END statement
```

5.2 DATA GENERATION PSEUDO OPCODES

The MACRO assembler has several pseudo opcodes which generate specific data constants within the program area for use as text messages, constant values, tables, etc. This section lists these pseudo opcodes and gives details on the data formats which are generated by them. All statements may have labels in which case the label is assigned the address that will receive the first byte of the generated data. All data statements begin allocating their specific data formats at the address specified by the assembly current location counter and generate multiple bytes in sequence, incrementing the current location counter as necessary. Those statements which generate byte data (BYTE, ASCII, BLKB) may begin and end on any byte address, odd or even. Those statements which generate word data (WORD, RAD50, BLKW) must begin on a word boundary (even byte) or else a boundary error (B) will result. The EVEN statement may be used at any point where the status of the current location counter is in doubt to insure an even boundary.

5.2.1 BYTE

The BYTE statement generates one or more bytes (eight bits each) of data. The arguments for generating the data consist of expressions separated by commas. Any legal expression is valid but only the lower byte will be stored after evaluation. Some examples are:

```
ZER:    BYTE    0               ;Generates 1 byte of data containing zero
        BYTE    1,2,3           ;Generates 3 bytes of data containing 1,2,3
MULTI:  BYTE    A-B,TAG*4,SAM   ;Generates 3 bytes of data
        BYTE     'A,'Q           ;Generates 2 bytes of ASCII data
```

5.2.2 WORD

The WORD statement generates one or more words (16 bits each) of data. The arguments for generating the data consist of expressions separated by commas. Any legal expression is valid which evaluates into a 16-bit value. WORD statements may also be generated by default if the first symbol on a line (after any labels) is not defined as an opcode, pseudo opcode or macro name. Some examples are:

```
ZER:    WORD    0               ;Generates 1 word (2 bytes) of data zero
        WORD    1,2,3           ;Generates 3 words of data containing 1,2,3
        WORD    A-B,"QT,SAM-.   ;Generates 3 words of data
        SAM                     ;Generates by default the value of SAM
```

5.2.3 ASCII

The ASCII statement generates one or more bytes of ASCII data. The argument for generating the data is a string of legal ASCII characters bounded on both ends by the same character which must not be included in the data string itself. Any printing character may be used as a delimiter. Only one such string may be generated by each ASCII statement. Some examples are:

```
MSG:    ASCII  /THIS IS A MESSAGE/  ;Generates a string of 17 data bytes
        ASCII  /Q/                  ;Generates a single data byte of "Q"
MSG2:   ASCII  $ I/O TERM $         ;Generates a string of 10 data bytes
```

5.2.4 RAD50

The RAD50 statement generates one or more words (16 bits each) of data. The argument is a string of valid RAD50 packable characters bounded on both ends by the same character which must not be included in the data string. Any printing character may be used as a delimiter. The legal characters for RAD50 packing are A-Z, 0-9, dollar-sign ($), period (.) and space. One packed word will be generated for each three characters in the string or fraction thereof with trailing spaces being assumed to fill out the last triplet. Some examples are:

```
DDB:    RAD50   /DSK/       ;Generates one word of packed data
        RAD50   /SAM QQ/    ;Generates two words of packed data
        RAD50   /ABCD/      ;Generates two words (same as RAD50 /ABCD  /)
```

5.2.5 BLKB - BLKW

These statements do not actually generate data but are included in this section because they result in the allocation of memory in a defined manner. The BLKB allocates an area of bytes and the BLKW allocates an area of words. In all other respects they operate the same. The argument for each is a single expression which evaluates to a value between 0 and 65535. This value is then added to the assembly current location counter (twice if BLKW) which effectively reserves that block of memory and continues allocating memory at the new address. Normally this results in a contiguous area of all zeros since the linker clears all blank areas when it generates the program file. This action does not always happen, however, because the location counter may be stepped back into the reserved area in which case the new data will overlay the reserved block of memory. This is an important concept in dealing with the absolute section since no data is actually generated by these statements, only memory addresses are reserved. Some examples are:

```
DATA:   BLKB    44          ;Reserves 44 bytes of memory
        BLKB    A*B         ;Reserves A*B bytes of memory
        BLKW    200         ;Reserves 200 words (400 bytes) of memory
```

5.3 SEGMENTATION PSEUDO OPCODES

The MACRO assembler, together with the LINK editor and monitor overlay calls, support a powerful method of segmenting and overlaying programs for both convenience during system development and memory conservation during execution. This section describes the methods available for the various options and also the assembler pseudo opcodes which help support the system. The pseudo opcodes we will discuss are AUTOEXTERN, INTERN, EXTERN and OVRLAY. This section also briefly discusses the concept of program libraries.

5.3.1 Segmenting Assembly Language Programs

There are several reasons for segmenting a program and also different methods for doing so, depending on the end result desired. A very large source program takes longer to edit (even a small change) and gives a greater opportunity for total loss if some disaster strikes the file links. A large program also takes longer to assemble and more memory in which to do so. Segmented programs may be organized in such a manner as to allow portions of the program to be resident in memory and other portions to be called in from disk only as required. Segmented programs may also contain duplicate symbols if the program segments are assembled separately and linked together by LINK. Also, program segments which are assembled separately may also be listed separately resulting in less listing time (and less paper used) for each change that is made.

The simplest method for creating a program in segments gains one of the above advantages. This method makes use of the COPY statement and allows a large program to be edited as multiple segments which are then copied into the main source program by using one COPY statement for each segment. As changes are made to the source program, you need only edit the segment which requires the changes. The assembly is done, however, on the complete source program since all copied files are included in the source input. Only one object file results and only one single list file can be created. The /L option on the COPY statement may be used to control those segments that are desired to be included on the listing itself.

A more complex but flexible method is to break up the program into logical segments which may be assembled separately and then linked together at a later time by the LINK program. Several object (.OBJ) files result as output of the different segment assemblies which are then input to the LINK program which creates a fully resolved and runnable program (.PRG) file. The advantages of the COPY method are realized as well as the added advantage of having to assemble only those segments which require changes. The LINK process runs much faster and requires less user memory than the assembly process. One of the requirements of a program which is segmented in this manner is that all references to routines and data constants which reside in another segment must be done through two special assembler pseudo opcodes, INTERN and EXTERN. Since a reference to a routine in another segment is not defined during the assembly of the calling segment, the symbol (name of the routine) is said to be "external." It is declared

external by the EXTERN statement which tells the assembler that it is defined and will be resolved by the linkage editor at a later time. The segment in which the routine exists then declares that symbol as "internal" via the INTERN statement which tells the assembler to output the symbol with a special code which defines it to the linkage editor for final resolution.

The method of segmenting a program and then creating a single runnable program with LINK may be extended one step further using a feature in the monitor which allows program segments to be called in from the disk and overlay an existing portion of the main program. A segment which is to be used as an overlay defines itself as such by using the OVRLAY statement and giving the address at which the overlay is to be loaded. The main program then uses a special form of the FETCH supervisor call to load the overlay segment and then executes it by jumping to a known segment start address. This implementation of overlaying segments is used in the MACRO assembler itself and conserves user memory during execution of large system programs. The LINK program creates one program (.PRG) file for the main segment and one overlay (.OVR) file for each overlay segment in use.

NOTE: Still another method for modularizing programs is the use of library files. Program libraries allow you to make use of frequently used routines in many different programs without rewriting those routines each time you need them.

You may specify one or more library (.LIB) files to LINK which then links in only those object files in the .LIB file that are necessary to resolve external references. For full information on generating and maintaining program library files, see Chapter 11, "The Object File Library Generator (LIB)."

5.3.2 AUTOEXTERN

The AUTOEXTERN pseudo opcode tells MACRO to automatically EXTERN any undefined symbols; those symbols are then displayed at the end of Phase 2 of the assembly. When AUTOEXTERN is in effect you do not have to explicitly EXTERN symbols.

5.3.3 INTERN

The INTERN statement defines one or more user symbols as internal to the program segment so that they will be defined to the linkage editor program for final resolution. The INTERN statement takes the form:

```
INTERN sym1{,sym2,...symN}
```

Each INTERN statement may be followed by one or more internal user symbols separated by commas. As many INTERN statements as required may be used in the program. There is also no limit to the number of symbols that may be referenced by each INTERN statement except for the physical line length.

Each symbol that is referenced in an INTERN statement must be defined within the segment either as a label on a routine or constant or as a value by an equate statement. The symbol will then be available to the LINK program for resolving references to it which come from EXTERN statements in other segments. Any symbol defined as external in a segment that has not been defined as internal in another segment will result in an undefined error during linkage editing. A symbol may never be defined by more than one INTERN statement during any one LINK run; i.e., the same symbol cannot appear as internal in two different segments that will eventually be linked into the same program.

A short hand notation for INTERNing a label or equated symbol exists. Instead of writing:

```
INTERN Symbol
Symbol:
```

you may now write:

```
Symbol::
```

Instead of writing:

```
INTERN Symbol
Symbol = Expression
```

you may now write:

```
Symbol == Expression
```

5.3.4 EXTERN

The EXTERN statement is used to define one or more user symbols as external to the segment so that they may be resolved by the linkage editor program. The EXTERN statement takes the form:

```
EXTERN sym1{,sym2,...symN}
```

Each EXTERN statement may be followed by one or more user symbols separated by commas. As many EXTERN statements as required may be used in the program. There is also no limit to the number of symbols that may be defined by each EXTERN statement except for the physical line length.

Each symbol that is defined by an EXTERN statement may be referenced within the segment just as if it had been defined within the segment as a label or an equate statement item. There is no limitation placed on its use as a term within any operand expression since the LINK program has complete expression resolution mechanics built in. There are two restrictions to its use within the segment. An externally defined symbol may not be used within the address operand of any branch instructions (BR, BEQ, BGT etc.) due to the fact that there is no way to insure that the resulting placement will

fall within the 127-word relative requirement. It may, however, be used within the address operand of the jump (JMP) instruction. The second restriction is that an equate statement may not contain any externally defined symbols in its operand expression since all equates must be fully resolvable as they are encountered.

The LINK program builds a symbol table from all the symbols referenced in all INTERN statements in all program segments. It then goes back and resolves all expressions containing symbols defined by EXTERN statements by looking them up in the table of INTERN symbols. Any symbol defined in an EXTERN statement but not matched by some INTERN symbol will give an error message during linkage editing.

5.3.5 OVRLAY

The OVRLAY statement identifies a program segment as being an overlay file instead of a continuation of the main program file. It also defines the address of the base of the overlay relative to the base of the main program so that the loading of the overlay segment is done at the proper spot in the program memory area. The OVRLAY statement takes a single argument which is a user symbol that must be defined in some other segment in an INTERN statement. For example:

```
        OVRLAY Sym
```

NOTE: It is legal to write:

```
            OVRLAY Sym
    Sym:    ...
```

as long as "sym:" appears at the start of the overlay. (The symbol "sym" is essentially defined twice with the same value.) The OVRLAY address will be resolved by LINK when the files are processed. Information on the code used to load the overlay segments into memory will be found in the description of the FETCH supervisor call in the AMOS Monitor Calls Manual. Further information on processing of the OVRLAY statement may be found in the section describing the LINK program processing.

5.4 CONVENIENCE PSEUDO OPCODES

There exist a few pseudo opcodes in the assembler that we refer to as convenience opcodes for lack of a better term. These opcodes do not really do anything that cannot already be accomplished by the existing source language in some other format, but they are easier to understand and make the listing more readable when used in the form that has been implemented here. Some of them are implemented directly in the assembler program itself while others exist as predefined macro calls in the system copy file SYS.MAC which is normally called by all programs.

5.4.1 Extended Conditional Jumps

One very frustrating thing about editing some new changes into a program is when you find that an existing BNE (or other conditional branch) no longer reaches due to the new code extending the address out of the 127-word limit for branches. The most common solution to this problem is to replace the offending branch with a branch of the opposite condition followed by a jump to the desired address. In other words, our BNE TAG could be replaced by BEQ .+6 followed by JMP TAG which effectively does the same thing. The only problem here is that this makes the listing somewhat less than clear when trying to decipher the flow of the program. We have therefore implemented into the assembler a set of conditional jump opcodes which effectively generate this two-instruction code sequence for the proper opposite conditional but which still look very readable in the source listing. These opcodes have been listed here along with the actual WD16 instructions generated:

JEQ	TAG	generates	BNE	.+6	followed by	JMP	TAG
JNE	TAG	"	BEQ	.+6	"	JMP	TAG
JPL	TAG	"	BMI	.+6	"	JMP	TAG
JMI	TAG	"	BPL	.+6	"	JMP	TAG
JLO	TAG	"	BHIS	.+6	"	JMP	TAG
JHI	TAG	"	BLOS	.+6	"	JMP	TAG
JLOS	TAG	"	BHI	.+6	"	JMP	TAG
JHIS	TAG	"	BLO	.+6	"	JMP	TAG
JLT	TAG	"	BGE	.+6	"	JMP	TAG
JGT	TAG	"	BLE	.+6	"	JMP	TAG
JLE	TAG	"	BGT	.+6	"	JMP	TAG
JGE	TAG	"	BLT	.+6	"	JMP	TAG
JCC	TAG	"	BCS	.+6	"	JMP	TAG
JCS	TAG	"	BCC	.+6	"	JMP	TAG
JVC	TAG	"	BVS	.+6	"	JMP	TAG
JVS	TAG	"	BVC	.+6	"	JMP	TAG

Remember that although these opcodes are easier (require less planning) than the simple branches they do actually generate three words of binary code instead of only one so, if space is at a premium, use them only when necessary.

5.4.2 PUSH - POP

The hardware stack in the WD16 is normally referenced by its index register (SP) and transferring words of data to and from the stack is done by MOV instructions. Many machines have dedicated instructions to push and pop data to and from the stack. In order to make the flow of system programs a little clearer for those of us used to pushing and popping, two macros have been implemented in SYS.MAC which recognize the PUSH and POP instructions. Each takes a normal source address argument but each also has a special default format which is used when no specific argument address is desired. These instructions generate the following code:

```
PUSH  SRC    generates    MOV  SRC,-(SP)   ;Pushes SRC onto stack
PUSH              "       CLR  -(SP)       ;Pushes a zero onto stack
POP   DST         "       MOV  (SP)+,DST   ;Pops stack into DST
POP               "       TST  (SP)+       ;Removes top stack word
```

5.4.3 CALL - RTN

The normal subroutine calling sequence of the WD16 is the JSR instruction which links its arguments through any of the eight registers. The assembler recognizes the more popular mnemonic opcode CALL for which it generates a JSR instruction. In addition, if no register is specified in the CALL or RTN instructions, the assembler assumes the most commonly used register PC for its argument linkage. In other words:

```
CALL  TAG    generates    CALL  PC,TAG
RTN               "       RTN   PC
```

5.4.4 OFFSET

There are many times during the programming of totally relocatable code where an address must be expressed and stored as a relative offset from the location of the constant itself. In other words, the storage of the address TAG must be in the form of TAG-. which is actually the offset from the current position of the constant itself to the address defined as TAG. The value of this constant offset will not change no matter what its position in memory happens to turn out to be. A good example of the use of relative address offsets is in the tables associated with the instructions TJMP and TCALL which must be relative offsets and not direct addresses. The OFFSET pseudo opcode has been implemented to make the listings a little more obvious as to intent. The OFFSET opcode takes a single address argument and generates the relative offset to that address from the current position of the constant.

5.4.5 PSI

Although intended only to be used internally to generate the system monitor macros, the PSI (PSeudo-Instruction) will be defined here as a result of the numerous inquiries about it. The PSI instruction will generate an instruction similar in format to the double-address instructions (such as MOV, ADD, SUB etc.) which may be one, two or three words in length depending on the address modes used. In addition, it allows a 4-bit pseudo opcode to be specified explicitly in the operand field. Basically, the format is:

```
PSI     opcode,source-address,destination-address
```

This results in a normal instruction format with the opcode comprising the top 4 bits (bits 12-15), the source address comprising the middle 6 bits

(bits 6-11) and the destination address comprising the low 6 bits (bits 0-5). Additional index words are generated if required by the addressing modes in use.

The instruction generated by the PSI statement is never executed directly by the machine since, in actuality, it duplicates one of the existing legal instructions. Instead, it follows a specific SVCB instruction and is used to generate the pseudo-instruction to be executed by the SVCB calling sequence and thereby results in an easy method for generating the standard address arguments.

CHAPTER 6

USER DEFINED MACROS

It is often convenient to create your own opcode definitions which when used in the source program result in the creation of a predefined sequence of one or more source code statements. These user-created opcodes are called "macros" in assembly language programming and the Alpha Micro assembler supports a flexible macro subsystem. There are two phases that you go through when using macro calls. First, you define the macro opcode once in the program as a series of source code statements along with possible dummy arguments. You only do this once; the macro remains defined throughout the remainder of the assembly process. Second, you then invoke the macro by a single source statement giving the macro name along with optional real arguments that replace the defined dummy arguments in the macro source code which is generated. Calling the macro in this manner causes the macro statement to be replaced by the defined sequence of source code statements that have been custom tailored by the optional real arguments in the calling statement. You may perform this calling sequence as many times as needed in the source program with as many different real arguments as desired.

6.1 MACRO DEFINITION

Defining a macro generates no actual binary code in the program but merely places the macro definition in a special table in the assembler memory work area. Calling the macro (which then generates the sequence of source statements) is the process that actually generates the binary code. If your program never calls the macro or if the macro does not contain any code-generating source statements, MACRO produces no binary code for the macro. The use of conditional assembly directives within a macro definition may result in no code-generating statements for this particular call to the macro. The fact that no code is actually generated if the macro is never called is an important concept since it then allows macro libraries to be created that may contain many macro definitions that are standard for a particular user system. Those macros that are never called in any specific program do not generate any code and therefore take up no additional memory. The system library SYS.MAC contains over 70 such macro definitions that define the supervisor calls to the monitor.

6.1.1 Macro Definition Formats

There are two formats available for use in defining macros. The normal format allows one or more source lines to be generated as a result of the macro call. The single-line format restricts the macro definition to one line of generated source code but takes up less room on the source listing. For several sample macros, see Section 6.1.10, below.

The general format for multiple-line macros is:

```
DEFINE  name    {dummy argument list}
        source line 1
        source line 2
            ...
            ...
        source line n
        ENDM
```

The general format for a single-line macro is:

```
DEFINE  name  {dummy argument list} = source line
```

In both forms above, the macro name is any legal user symbol; it effectively becomes the opcode by which the macro is called. This symbol may duplicate a label in the program or may even redefine an AM-100 pseudo opcode or a WD16 machine opcode (e.g., you can redefine the MOV opcode to do an ADD if you really want to confuse some people). You may only define a macro name once and an attempt to redefine it later in the program will give unspecified results.

6.1.2 The Macro Source Statements

The multiple-line macro definition source statements begin with the line immediately following the DEFINE statement and continue through to but not including the ENDM termination line. NOTE: Every macro definition must end with the ENDM pseudo opcode.

When the program text calls the macro, MACRO will generate and assemble all macro source lines just as if they had been explicitly entered directly into the source program. In the single-line form, the source line begins with the character following the equal sign and continues through (and including) the carriage-return and line-feed pair which terminates the DEFINE statement line.

Macro definitions must not be nested within other macro definitions. Macro processing is done on a special prepass scheme which prohibits the processing of any DEFINE statements within another DEFINE statement.

6.1.3 The Dummy Argument List

The dummy argument list is optional in both forms of macros and consists of one or more user symbols separated by commas. These symbols are unique only within the actual definition of the current macro and may be duplicated in other macro argument lists or may even be other opcodes and defined symbols. These dummy argument symbols will never appear as such in the generated sequence of source statements when the macro is called but will be replaced by the equivalent real arguments supplied in the calling statement. The dummy argument symbols may appear anywhere in the definition source lines, even as labels. Each time MACRO encounters a dummy argument when generating the source lines during a macro call, it replaces the dummy argument with the corresponding real argument that was supplied by the calling statement.

6.1.4 Labels

A label must not be used on the DEFINE statement line since it has no meaning. Labels may be used on the calling statements. A label must not be used on the ENDM line or the ENDM line will not be detected.

6.1.5 Local Symbols

MACRO supports local symbols of the form nnn$ and nnn$$, where nnn is a number between 0 and 65535, decimal. Local symbols of the form nnn$ have scope only between two non-local labels, and may be used outside of macro definitions.

Local symbols of the form nnn$$ are for use only within macro definitions. If a nnn$$ label appears outside of a macro, MACRO will treat the label like nnn$ except that the label will not appear in the symbol table file (used for debugging purposes). NOTE: You may define a local symbol with an equate (=).

Below are two sample macros that use local symbols:

```
DEFINE  LEAMSG  X
        LEA     R0,10$$         ; Get address of message
        BR      20$$            ; Branch around message
10$$:   ASCII   'X'
        BYTE 0
        EVEN
20$$:
ENDM
```

Now we call the macro:

```
        LEAMSG  HELLO
        TTYL    @R0             ; Display HELLO
        LEAMSG  BYE
        TTYL    @R0             ; Display BYE
```

The example above works correctly even though it generates two occurrences of 10$$ and 20$$ because the symbols are local to each macro call.

The example below demonstrates that local labels of the form nnn$ can be passed as arguments to macros, and that they will be distinguished from labels of the form nnn$$ even if "nnn" is the same:

```
DEFINE  JGT10   X,Y
        CMP     X,#10
        BLE     1$$
        JMP     Y
1$$:
ENDM
```

Now we call the macro:

```
        JGT10   R0,$1   ; expands to:
                        ;       CMP     R0,#10
                        ;       BLE     1$$
                        ;       JMP     1$
                        ; 1$$:
        DEC     R0
1$:     RTN
```

6.1.6 Comments

A comment may follow the dummy argument list in the multiple-line form but you should not use a comment with the single-line form. You should avoid comments in the actual generated source lines in the macro definition simply because MACRO stores the entire source text in work memory as ASCII characters (including all comments). This may tend to use up work memory to the extent that you may not have enough memory to finish the assembly.

6.1.7 Special Macro Operators

Two special operators exist that are used only within macro definitions: the argument concatenation operator (') and the expression evaluation operator (\).

6.1.7.1 Argument Concatenation (') - Since dummy arguments must be valid user symbols, the apostrophe (') is a legal delimiter for any dummy argument within a macro definition source line. When an apostrophe immediately precedes and/or follows a dummy argument in the source text, the apostrophe is removed and the substitution of the real argument occurs at that point. This is useful for building symbols with arguments that are to be a part of that symbol.

Given the following macro definition and eventual calls:

```
DEFINE  BUILD   AA,BB
TAG'AA: MOV     R1,Q'BB'7
        ENDM
         ..
         ..
        BUILD   RA,STS
        BUILD   T,P
```

the effective code generated by the two calls would be:

```
TAGRA:  MOV     R1,QSTS7
TAGT:   MOV     R1,QP7
```

6.1.7.2 Expression Evaluation (\) - The \ operator tells MACRO to evaluate the expression that follows and to return its value. (Before local symbols were supported by MACRO, the \ operator was often used to simulate local symbols. For information on true local symbols, see Section 6.1.5, "Local Symbols.") You may use an expression of the form:

```
\expr
```

(a "\" followed by an expression) within a macro definition. MACRO then evaluates the expression and returns its value as a string. By placing a symbol in front of the \, you can direct MACRO to append the value of the expression following the \ onto the end of the symbol. For example:

```
LABEL\4*4:
```

evaluates to:

```
LABEL16:
```

and:

```
$ = 1
STC/$:
```

evaluates to:

```
STC1:
```

Symbols generated in this way do take up room in the symbol table.

NOTE: Be very careful that the expression following the \ operator does not contain any macro arguments; they will not be expanded properly and will probably cause a syntax error (Q code).

6.1.8 Suppressing Macro Expansion - ENDMX

The ENDMX pseudo opcode ends the expansion of the current macro. This pseudo opcode is illegal outside of a macro definition. You will find this pseudo opcode useful when using conditional assembly directive pseudo opcodes to control macro expansion. (NOTE: ENDMX controls what macro code is generated at the time of a macro call; it does not affect whether the macro expansion is included in your assembly listing.)

6.1.9 NCHR, NTYPE, NEVAL and NSIZE

These four macro directives return a value that specifies the number of characters in an argument (NCHR), the addressing mode type of an argument (NTYPE), the value of any extra word generated by the addressing mode evaluation, or the length of any extra words generated by an addressing mode. These statements function similarly to the equate statement (=) in that they assign a value to a user symbol which may be reassigned as many times as desired during the course of the assembly. They are normally used to control the development of macro source code based on the size and type of arguments passed to the macro and therefore are defined in this section dealing with macros. In actuality, you may use them anywhere in the source program with any valid source code as an argument but they are fairly meaningless unless used within a macro.

Once the symbol has been assigned a value by one of the NCHR, NTYPE, NEVAL directives, you may use it by itself or within expressions to control the development of the macro source code through the conditional assembly statements.

6.1.9.1 NCHR - The NCHR statement assigns a value to a user symbol that is equivalent to the number of characters in the argument string. It has the format:

```
NCHR    symbol,string
```

6.1.9.2 NTYPE - The NTYPE statement assigns a value to a user symbol that is equivalent to the 6-bit addressing mode of the argument. It has the format:

```
NTYPE   symbol,argument
```

The following is a list of the addressing modes and the values that they will deliver via the NTYPE statement. The upper case "R" represents any of the eight registers (R0-R5, SP, PC) which have a corresponding result value of 0-7 added to the resulting mode they are used in.

```
R       direct register delivers 0R
@R      indirect register delivers 1R
(R)+    autoincrement delivers 2R
@(R)+   indirect autoincrement delivers 3R
-(R)    autodecrement delivers 4R
@-(R)   indirect autoincrement delivers 5R
X(R)    indexed delivers 6R
@X(R)   indirect indexed delivers 7R
#X      immediate delivers 27
TAG     relative delivers 67
@TAG    indirect relative delivers 77
```

For example, if you use register R4 in indirect addressing mode, NTYPE returns a 14 (i.e., 1R where R = register 4).

6.1.9.3 NEVAL - The NEVAL statement assigns a value to a user symbol that is equivalent to the value of the extra word generated by one of the indexed, relative or immediate addressing modes. This word represents the index augment for indexed modes, the relative offset for relative modes or the immediate value for the immediate mode. It has the format:

```
NEVAL   symbol,argument
```

6.1.9.4 NSIZE - The NSIZE statement assigns a value to a user symbol that is equal to the size of the address form (i.e., 0 if no extra word is generated, 2 if an extra word is generated). It has the format:

```
NSIZE   symbol,argument
```

6.1.10 Sample Macro Definitions

Below are several sample macro definitions.

A macro called ADDIT which generates four instructions:

```
        DEFINE  ADDIT
                MOV     R1,R3
                ADD     R3,SUM
                ASL     R3
                ADD     R3,SUM
                ENDM
```

A macro called XCHNG which exchanges two memory words:

```
        DEFINE  XCHNG   MEMA,MEMB
                MOV     MEMA,R1
                MOV     MEMB,MEMA
                MOV     R1,MEMB
                ENDM
```

A macro called STKSUB which subtracts a memory word from the top stack word:

```
        DEFINE  STKSUB  TAG
                SUB     TAG,@SP
                ENDM
```

The same STKSUB macro in the single-line format since only one line is used:

```
        DEFINE  STKSUB  TAG = SUB  TAG,@SP
```

For some more complex examples of macro definitions, print out or inspect the system macro library SYS.MAC that defines all of the supervisor calls used by the AM-100 computer system.

6.2 MACRO CALLS

The actual generation of the defined source code comes when you call the macro by its name within the text of your source program. The macro must have been defined prior to its first reference. Macros are only processed for definition during Phase 1 of the assembly process. Macro calls have the same format regardless of whether the macro definition is multiple or single line format:

```
{label:}  name  {real arguments}  {;commments}
```

6.2.1 Name

Name represents the name given to the macro definition; this becomes the effective opcode by which your program calls the macro.

6.2.2 Real Arguments

Use real arguments when the definition of the macro has a dummy argument list; they actually replace the dummy arguments in the source code text of the macro definition. The real arguments replace the dummy arguments on a one-for-one basis in exactly the same order as the elements of the dummy argument list. The first real argument in the call takes the place of each occurrence of the first dummy argument in the definition, and so on for all the arguments. If there are not enough real arguments given in the call to fill all required dummy arguments, the unfilled dummy arguments take on a null value and are effectively replaced with nothing. If there are more arguments in the call then required to fill the dummy arguments in the definition, MACRO ignores the excess arguments.

6.2.2.1 Real Argument Format - Normally, the real arguments are separated by commas and the assembler expects this format. Also, leading and trailing blanks are ignored when processing each real argument in the macro call statement. Often you may want to include a comma or blank as part of the real argument without having it act as a delimiter or be bypassed. Any argument that is enclosed in angle brackets will be passed onto the source code generation verbatim including any blanks and commas.

The macro call:

```
XPURT   ONE,TWO,THREE
```

has three real arguments while the call:

```
XPURT   <ONE,TWO,THREE>
```

has only one argument which includes the two commas. The call:

```
XPURT   <ONE,TWO>,THREE
```

has two real arguments of which the first includes one comma.

The system macro TYPE is another good example:

```
DEFINE  TYPE    MSG
                TTYI
                ASCII   /MSG/
                BYTE    0
                EVEN
                ENDM
```

This macro is one of the AMOS monitor calls and is designed to type out the ASCII message which appears as the argument to the TYPE macro call. The BYTE 0 statement insures a null terminator and the EVEN statement insures that the next instruction is again synchronized on a word boundary.

The call:

```
TYPE            HELLO
```

will type out the message "HELLO" because all the leading blanks are automatically ignored before the argument is processed. The call:

```
TYPE    <     HELLO >
```

will type out the message " HELLO " because the blanks are included in the argument as a result of the angle brackets. Similarly, the call:

```
TYPE    HELLO, I AM A COMPUTER
```

will type out the message "HELLO" because the comma will terminate the argument and the rest of it will be ignored. The call:

```
TYPE    <HELLO, I AM A COMPUTER>
```

will type out the message "HELLO, I AM A COMPUTER" because the comma is included in the argument as a result of the angle brackets.

6.2.3 Label

The label is optional and will be assigned the address contained by the assembly current location counter. This will normally be the address of the first byte of code which is generated by the macro source lines (assuming that the macro does actually generate code). If the macro does not generate code, then the label will still be defined but it will represent the address of the next byte of code that is generated after the macro call.

6.2.4 Comments

As in other statements, comments are optional.

6.2.5 Nested Macro Calls

Macro calls may be nested to a depth of 16 levels. A nested macro is defined as a macro call within the source statements generated by another macro call. Arguments may be passed to nested macros by naming the dummy arguments the same throughout the levels. Arguments that contain blanks or commas may be passed through successive levels by enclosing them in one set of angle brackets for each level of nesting since one set of angle brackets will be removed from an argument with each nesting level. For example, to pass the argument A,B through three levels of nested macro calls you would enter the argument as <<<A,B>>> in the first level macro call.

6.2.6 Sample Macro Calls

Consider this example:

```
        DEFINE  TBLADD  ARG1,ARG2,ARG3
                MOV     ARG1,R1
                ADD     ARG2,R1
                MOV     R1,ARG1(ARG3)
                ENDM
```

This macro is called TBLADD and requires three real arguments. Assume the following call in your program:

```
        SAM:    TBLADD  SUMS,ENTRY,R5
```

The following source statements would be generated:

```
        SAM:    MOV     SUMS,R1
                ADD     ENTRY,R1
                MOV     R1,SUMS(R5)
```

It is evident from its usage that ARG3 must be a register. Assume that only two arguments were given in the call:

```
        SAM:    TBLADD  SUMS,ENTRY
```

The following source statements would be generated:

```
        SAM:    MOV     SUMS,R1
                ADD     ENTRY,R1
                MOV     R1,SUMS()
```

Notice that the third instruction would contain an error due to the missing register term which resulted from the missing third argument. Sometimes a missing argument may be used to advantage by altering the generation of the source statements with the conditional assembly statements. These statements (described in the next chapter) can detect the fact that the argument is missing and be used to selectively omit portions of code.

CHAPTER 7

CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives allow you to selectively include or bypass certain lines or segments of source code based on variable parameters which are tested during assembly. This allows several different versions of the same program to be generated from one source file. Conditional assembly directives find their widest use within macro definitions where they are used to tailor the macro based on the real arguments used in the macro call.

NOTE: You may find the MACRO parameterized assembly option especially useful when used with conditional assembly directives. The MACRO /V switch allows you to provide a value on the MACRO command line which can be examined by your source program. See Section 9.2.3 for information on this feature.

7.1 CONDITIONAL DIRECTIVE FORMATS

Like the macro definitions, conditional directives follow two general forms. The normal form allows one or more lines of source code to be selected or bypassed based on the current status of a variable. The single line form performs the same function but is a shorter version and only allows the control of a single line of source code.

The general form of a normal conditional block is:

```
        IF      condition,argument
        source line 1
        source line 2
            ...
            ...
        source line n
        ENDC
```

The general form of a single-line conditional is:

```
        IF      condition,argument, source-line
```

Both forms employ the IF pseudo opcode to identify the conditional directive and both forms require a condition code which specifies the type of test to be performed and an argument upon which to perform that test. The condition code is a symbol which identifies the test which is performed at the time the conditional is encountered during Phase 1 of the assembly process. The argument may be a symbol, expression or macro argument, depending on the type of test being performed.

Note that the item that distinguishes the two forms is the comma that follows the argument in the single-line form. If the comma exists, the remainder of the line up to and including the carriage-return and line-feed will be the source line that will either be assembled or bypassed depending on the result of the conditional test. If the comma does not exist, the conditional assembly will be done on the source line that follows the conditional directive (IF) line up to but not including the ENDC terminating line.

7.2 CONDITION CODES

The following is a list of the condition codes that are legal and the type of condition that the associated argument is tested for. Unless otherwise specified, the argument is evaluated as an expression and the 16-bit result of that evaluation is the quantity that is tested to meet the condition. The conditional source lines are assembled if the argument meets the condition listed next to the code below.

EQ The argument is equal to zero.

NE The argument is not equal to zero.

LT The argument is less than zero.

GT The argument is greater than zero.

LE The argument is less than or equal to zero.

GE The argument is greater than or equal to zero.

DF The argument is completely defined at this point.

NDF The argument contains one or more undefined symbols at this point.

B The argument (a string of ASCII characters) is blank or null.

NB The argument (a string of ASCII characters) is not blank or null.

7.3 SUBCONDITIONALS

There are three subconditional directives that allow the alteration of the normal conditional processing within a conditional block. These subconditionals (IFF, IFT and IFTF) require no other parameters and must be used within the source code that is between the IF and ENDC statements. The following functions may be performed through the proper use of subconditionals:

1. Assembly of an alternate block of code when the main conditional code is being bypassed due to a failed conditonal test.

2. Assembly of a noncontiguous body of code within the conditional block depending on the result of the main conditional test.

3. Unconditional assembly of a block of code within a conditional block regardless of the result of the conditional test.

The three subconditionals and their functions are:

IFF The source lines following the IFF statement up to the next subconditional or end of main conditional are assembled if the main conditional test result was false.

IFT The source lines following the IFT statement up to the next subconditional or end of main conditional are assembled if the main conditional test result was true.

IFTF The source lines following the IFTF statement up to the next subconditional or end of main conditional are assembled regardless of the main conditional test result.

7.4 NESTING OF CONDITIONALS

Conditionals and subconditionals may be nested to a maximum depth of 16 levels. Any conditionals within a higher level conditional will be bypassed (the test will not be performed) if the result of the higher level conditional test was false. Subconditionals within outer level conditional blocks will be tested while those within inner level untested blocks will be ignored. Consider the following simple example:

```
TEST1:  IF      EQ,3-3          ;True so assemble following code
        WORD    33              ;Assembled since EQ,3-3 was true
        IF      NE,4-4            ;False so bypass following code
        WORD    44                ;Not assembled since NE,4-4 was false
        IFF                         ;Tested - true since NE,4-4 was false
        WORD    441                 ;Assembled since IFF was true
        IFT                         ;Tested - false since NE,4-4 wasn't true
        WORD    442                 ;Not assembled since IFT was false
        IFTF                        ;Tested - true regardless of NE,4-4
        WORD    443                 ;Assembled since IFTF was true
        ENDC                      ;End of NE,4-4 conditional block
        ENDC                    ;End of EQ,3-3 conditional block
TEST2:  IF      EQ,5-6          ;False so bypass following code
        WORD    56              ;Not assembled since EQ,5-6 was false
        IF      EQ,6-6            ;Not tested since EQ,5-6 was false
        WORD    61                ;Not assembled since EQ,6-6 was untested
        IFF                         ;Not tested since EQ,6-6 was untested
        WORD    661                 ;Not assembled since IFF was untested
        IFT                         ;Not tested since EQ,6-6 was untested
        WORD    662                 ;Not assembled since IFT was untested
        IFTF                        ;Not tested since EQ,6-6 was untested
        WORD    663                 ;Not assembled since IFTF was untested
        ENDC                      ;End of EQ,6-6 conditional block
        ENDC                    ;End of EQ,5-6 conditional block
```

The system macro for the PUSH convenience opcode is a good example of how conditionals may be used to control the code generated by a macro:

```
DEFINE  PUSH    SRC
        IF      B,SRC, CLR -(SP)
        IF      NB,SRC, MOV SRC,-(SP)
        ENDM
```

If the macro is called without an argument (SRC is blank) then the first conditional is true and the code CLR -(SP) is generated to push a zero word onto the stack. The second conditional is therefore false and generates no code. If the macro is called with an argument (SRC is not blank) then the reverse happens and the code MOV SRC,-(SP) is generated with SRC being replaced by the real argument in the calling statement. This causes the SRC word to be pushed onto the stack.

The same PUSH macro could have been alternately coded using subconditionals:

```
DEFINE  PUSH    SRC
        IF      B,SRC
        CLR     -(SP)
        IFF
        MOV     SRC,-(SP)
        ENDC
        ENDM
```

For some more examples of conditionals used within macros, print out or inspect the system library SYS.MAC which defines all of the supervisor calls used by the AM-100 computer system. This file is on the System Disk in account [7,7].

CHAPTER 8

WRITING RELOCATABLE AND RE-ENTRANT CODE

The Alpha Micro computer system not only supports relocatable programs, but requires that all programs written for operation under control of the AMOS monitor be written in totally relocatable code. This means that a program may be loaded physically into memory at any location and it will run without modification. No addresses within the program ever need to be modified since all references to memory are made in relation to the current value of the program counter register (PC). The program may even be dynamically moved about in memory without modification so long as it is not currently active while it is being moved. The code is actually independent of its position in memory and therefore has often been referred to by other manufacturers as "position independent code."

Writing relocatable code for the AM-100 system has been simplified by the incorporation of several instructions which make references to the current position of the program automatic. The load effective address (LEA) instruction may be used to calculate the current value of any relocatable address and to load that current value into any register. The table referencing instructions (TJMP and TCALL) both use relative offsets to perform their functions as opposed to absolute or calculated addresses.

8.1 VALID ADDRESSING MODES

Due to the normally relocatable nature of the AM-100 instruction set and addressing modes, writing totally relocatable code merely involves obeying a few specific restrictions in the course of programming. The most important of these is to never refer to any absolute address in main memory unless you are sure of its location and contents. Two of the addressing modes will always generate absolute memory references and must be avoided when writing relocatable code. Note the following examples:

```
CLR     @#TAG
CLR     TAG(R4)
```

In the first example the absolute address of TAG is stored in immediate mode and then used to indirectly address that absolute memory location. This addressing mode is not relocatable unless the reference to TAG is a reference to a known absolute memory location. In the second example, the most common method of indexing can be shown to be non-relocatable. Normal indexing address schemes take the base of some area (in this case it is TAG) and add an offset from some calculation which is stored in an index register (in this case R4) to develop the target memory address. The value of TAG is stored in the instruction as an absolute value and no offset is ever added to compensate for relocation of the program. This mode would not be relocatable unless, as in the first example, the reference to TAG is to a known absolute memory location.

The two above addressing modes are the most commonly made errors that violate the rules for relocatable code. A more subtle mistake is made when a register is set up as an index to a table within the user program to be referenced later through the register. Take these examples:

```
MOV     #TABLE,R0
LEA     R0,TABLE
```

The first example stores the address of TABLE as an absolute value due to the immediate mode addressing. Since the assembly of the program is done starting at location zero, the value of TABLE during assembly is really the offset from TABLE to the base of the program. When the program actually runs, it will not be located at zero (the operating system resides in the first 12K or so) and the actual address of TABLE will not be the same as at assembly time. The second example is the proper instruction to be used when setting up a register to a memory reference. The instruction is coded at assembly time as an offset from the instruction itself to the location marked as TABLE and when the LEA instruction is executed, the actual value of TABLE in its current location is calculated and loaded into the register.

Addressing modes that involve only register references are totally relocatable. These modes are:

```
Rx      direct register
@Rx     indirect register
(Rx)+   autoincrement
@(Rx)+  indirect autoincrement
-(Rx)   autodecrement
@-(Rx)  indirect autodecrement
```

The two relative addressing modes are also relocatable:

```
TAG     relative
@TAG    indirect relative
```

8.1.1 Index Modes

Index modes can be relocatable or non-relocatable depending on their usage and set up procedure. Generally speaking, if the register is absolute and the index offset is a relative tag in the program, the indexing is not relocatable and will deliver wrong results. If the register is first loaded with the effective value of the relative address within the program and the index offset is the absolute component, then the scheme is relocatable and will give the desired results. Take the following two examples of clearing the third word (sixth byte) in TABLE:

This is the wrong way:

```
        MOVI    6,R3            ;R3 gets absolute component offset
        CLR     TABLE(R3)       ;absolute location TABLE(R3) is cleared
```

This is the right way:

```
        LEA     R3,TABLE        ;R3 gets current address of TABLE in program
        CLR     6(R3)           ;relocatable location at TABLE+6 is cleared
```

8.2 RE-ENTRANT CODE

Writing re-entrant programs involves a little trick which can be played with relative code machines. Re-entrant programs distinguish themselves by their ability to be placed into system memory (via the SYSTEM command in your SYSTEM.INI file) and simultaneously shared by multiple users. A good example of a re-entrant program is the AlphaBASIC compiler and runtime package. More than one user may share this program without loading it into each of their individual memory partitions. The main problem with writing re-entrant programs deals with the local variables that must be used as a work space for each user. These individual work spaces must be allocated within the user's own memory partition and yet must be accessed by the common re-entrant program. Remember, the re-entrant program must never store variables within its own program area or else it is no longer re-entrant.

8.2.1 Using Base Registers

If a table of the named local variables is created using BLKB and BLKW statements at the beginning of the re-entrant program, the labels assigned to these variables may be used as indexes to the variable area once it has been allocated within the user's memory space. This concept requires that one register (R0-R5) be dedicated throughout the program as the base point for the local variable area. For an example, let's suppose that your program will require four variables called VARA through VARD with the following sizes:

```
        ASECT
        .=0
VARA:   BLKW    4               ;variable 1 size is 4 words
VARB:   BLKW    1               ;variable 2 size is 1 word
VARC:   BLKB    16.             ;variable 3 size is 16 bytes
VARD:   BLKW    1               ;variable 4 size is 1 word
        .=0
        RSECT
```

The above table will be at the beginning of the re-entrant program defining a local variable area of 14 words (or 28 bytes). The two ".=0" statements surrounding the table are required to insure that the area generates no code but is merely used to set up the index values assigned to the labels VARA through VARD. Generation of the actual program code which follows will then begin at relative location 0 where it is expected. The ASECT call sets the assembler into absolute mode so that the variables are defined as non-relocatable. The RSECT call restores relocation for the following program code. The program must set up the above variable area by allocating the required space within the user's memory partition (probably with a GETMEM call) and set the selected index register to point to its absolute base address (returned by the GETMEM call).

If we assume that you have chosen R5 to be your index to the variable area and have set it to point to the allocated 14-word block, the four variables may then be referenced throughout the program execution by the following addresses:

```
VARA(R5) for variable 1
VARB(R5) for variable 2
VARC(R5) for variable 3
VARD(R5) for variable 4
```

In addition to the above direct addressing method, another index (say R2) may be set to index an individual variable with the following statement:

```
LEA     R2,VARC(R5)     ;index the 16-byte variable 3
```

The index R2 now points to the specific VARC variable which might be used for incremental indexing within itself (perhaps to store 16 1-byte flags).

Remember that in the above scheme, the base index register R5 in this example) must never be destroyed in the program execution or else you will not be able to reference any of the variables.

In summary, the best way to learn how to evaluate the relocatablity of a particular programming technique is to become thoroughly familiar with the addressing modes used by the WD16 chipset and the type of code that they generate. This information can be found in the WD16 Microcomputer Programmer's Reference Manual, (DWM-00100-04).

PART II

USING THE ALPHA MICRO ASSEMBLY LANGUAGE PROGRAMMING SYSTEM

These chapters describe the use of:

MACRO	-	The macro assembler.
LINK	-	The linkage editor
SYMBOL	-	The symbol table generator
LIB	-	The object file library generator
GLOBAL	-	The global symbol cross reference generator
DDT	-	The dynamic debugging and patching program

For information on the screen-oriented assembly language program debugger AlphaFIX, see the AlphaFIX User's Manual, (DWM-00100-69).

CHAPTER 9

THE ALPHA MICRO ASSEMBLER (MACRO)

This chapter discusses the Alpha Micro assembler program, MACRO.

After writing your source code (the .MAC file), you must assemble it. The assembler translates your assembly language program into machine language (the .OBJ file). The linkage editor (discussed in the next chapter) processes the .OBJ files to resolve all symbol references and to create the final, executable program (.PRG or .OVR) file.

This chapter gives information on the operation of the macro assembler program.

9.1 THE MACRO PHASES

The assembler actually runs in five distinct phases that are selectively called depending on what functions are needed. A brief summary of their respective functions follows:

PHASE 0 - Interprets the command line and sets up parameters in the common area for use by successive phases.

PHASE 1 - Reads the source (.MAC) file and performs Pass 1 of a standard two-pass assembly process by expanding macros, building the user symbol table, and generating the interphase work (.IPF) file.

PHASE 2 - Reads the interphase (.IPF) file and performs Pass 2 of a standard two-pass assembly process by resolving symbols and generating the object code (.OBJ) file. MACRO then deletes the interphase work file.

PHASE 3 - Reads the source (.MAC) file and the object (.OBJ) file and creates a list (.LST) disk file or outputs the assembly listing to the terminal.

PHASE 4 - Actually not part of the assembler but an automatic call to the LINK program to read the object (.OBJ) file and create a runnable program (.PRG or .OVR) file. Only occurs if there were no internal or external symbol references in the program. (If Phase 4 is not called, you will later have to use LINK to link this file with the other files that contain the symbols that will resolve the external and internal references.)

9.2 COMMAND LINE

The general format for the assembler command line is:

.MACRO filespec{/switches} RET

9.2.1 Filespec

Filespec specifies the source file you want to assemble; it may optionally be a complete file specification containing account and device specifications.

The /switches option request is a slash followed by one or more alphabetic characters. A switch alters the normal assembly process. If you enter no switches, MACRO performs an assembly on the specified source file and creates an object file but no list file (i.e., Phase 3 is bypassed). If the program is a single segment (i.e., it contains no INTERN or EXTERN statements), then MACRO enters Phase 4, which creates an executable (.PRG or .OVR) program file.

9.2.2 Assembler Options

You may select one or more of the assembly options below by specifying the appropriate switch on the MACRO command line:

/B text Generates a bottom footer line on every page of the listing using the rest of the text on the command line following the /B switch as title information. For example:

.MACRO DEVCPY/B Version A00 RET

generates a listing file of which every page contains the bottom line title: "Version A00." /B must be the last switch on the command line.

/C Includes conditionals in the listing. (Conditionals are normally suppressed.)

/E Writes to the assembly listing only those lines that contain an error.

/H Lists binary code in hexadecimal instead of octal in the assembly listing.

/L Generates a list file by calling Phase 3 during the assembly. Creates the output file with the same name as your source file, but a .LST extension. (You may modify the name of your listing file by using the OBJNAM pseudo opcode in your source program-- see Section 5.1.2, "OBJNAM.")

/O Uses the current object file by omitting Phases 1 and 2.

/R Generates a cross reference, which appears at the end of the assembly listing. See Section 9.4.3, "Generating a Cross Reference," for information on the cross reference listing.

/T Prints the assembly listing on your terminal instead of writing it to a disk file.

/V{a}:X Allows you to specify a value on the MACRO command line which can be examined during the assembly process. "a" specifies the type of value specified, and X is the value. See Section 9.2.3, "Parameterized Assembly Option," for more information.

/X Lists in your assembly listing all macro expansions. (Macro expansions are normally suppressed.)

NOTE: You do not have to specify the /L switch when you use the /B, /C, /E, /H, /R, /T, or /X switches to tell MACRO to generate a listing.

You may combine any of the above switches as desired in a single command line by entering them after a single / character at the end of the command line. For example:

```
.MACRO NEWDVR.MAC/RT(RET)
```

The command line above tells MACRO to generate a listing file for NEWDVR.MAC that contains a cross reference and to output that listing to the terminal.

The most common method of assembling new programs is as follows:

1. Assemble the program with the command:

```
.MACRO filespec(RET)
```

   This will allow you to count any errors that occur during Phases 1 and 2.

2. If no errors occur, create a list file with:

   .MACRO filespec/LO (RET)

   or, optionally, list it on the terminal with:

   .MACRO filespec/TO (RET)

   or, get a cross reference with the listing:

   .MACRO filespec/RO (RET)

3. If there were errors, list them alone with:

   .MACRO filespec/TOE (RET)

   Correct the errors and go back to Step 1.

4. If the program has only one segment, then MACRO automatically calls Phase 4 which creates the .PRG or .OVR program file; otherwise, you will need to use the LINK or SYMBOL program to generate the final program file-- see the next chapter for information on LINK and SYMBOL.

9.2.3 Parameterized Assembly Option

MACRO provides a parameterized assembly facility by allowing you to use the /V switch to specify a value on the MACRO command line. The value switch may take one of these forms:

/V:x	x is an octal or hex number (depending on the prevailing radix setting)
/VO:x	x is an octal number
/VH:x	x is a hexadecimal number
/VD:x	x is a decimal number
/VA:x	x is one or two ASCII characters
/VR:x	x is one to three RAD50 characters

The NVALU pseudo opcode allows your program to access the value specified in the /V assembly switch. The NVALU statement takes the form:

```
NVALU sym
```

which sets the symbol "sym" to one of the values below, depending on which /V switch was used:

```
sym=x
sym=^Ox
sym=^HOx
sym=^Dx
sym='x
sym="x
sym=[x]
```

You may find this feature especially useful when using conditional assembly directive pseudo opcodes to select which portions of code to assemble.

9.3 SAMPLE ASSEMBLY DISPLAY

Below we show a sample assembly display:

```
.MACRO SAVTXT.MAC/L (RET)

== Macro Assembler Version 1.1 ==

Processing SAVTXT.MAC

Phase 1:   Copying from DSK0:SYS.MAC[7,7]
           Work area: 3916 bytes, 3614 used
Phase 2:   Object file finished
Phase 3:   Listing file finished
Phase 4:   Program file finished  [Program size = 60. bytes]
.
```

If MACRO is automatically EXTERNing any symbols, it tells you so in Phase 2 (listing the symbols alphabetically). For example:

```
Phase 2: Object file finished
EXTERNs were generated for the following symbols:
        GETNUM  PRTNUM
```

In the case above, MACRO automatically EXTERNed the symbols GETNUM and PRTNUM. MACRO automatically EXTERNs symbols if those symbols are undefined and if the AUTOEXTERN pseudo opcode appears in your source file.

Notice that even if your program is a single segment, MACRO will not call Phase 4 to link your program if MACRO was not able to resolve all symbol references in your program (that is, if EXTERNs were generated). You will need to use LINK or SYMBOL to link your program with the other file(s) that contain the symbols referenced by your main program.

If you ask for a cross reference listing, you see the following message during Phase 3:

```
Phase 3: Cross reference file finished
```

9.4 THE ASSEMBLY LISTING

By specifying the appropriate assembly switches, you can direct MACRO to call Phase 3 of the assembly process to create a list file which is sent to a disk file or to your terminal. The listing is formatted and contains both the source of your program and binary code that is generated by the assembly.

9.4.1 Assembly Listing Format

Each page contains a page number and a title that gives the name of the program that has been assembled and the account number that the file was assembled in. Unless otherwise controlled by PAGE statements, each page contains 54 lines of source data. Each page is terminated by a form-feed character. If the system date has been set (via the monitor level DATE command), the date appears at the top of each page of the listing. If you specified the /B assembly switch, MACRO outputs to each page a page footer containing the text specified on your MACRO command line.

Each data line on the listing contains four sections:

1. Columns 1-5 list the error codes on the line that generated the error. (For a list of the MACRO error codes, see Section 9.5, "MACRO Errors.")

2. Columns 8-13 list the current address of the generated data if any data code was generated. Or, these columns give the value of the assignment if this is an equate statement.

3. Columns 16-37 list the generated binary data (maximum of the first three words) in octal (or hex if /H assembly switch was used).

4. Columns 40-132 list the source line.

9.4.2 Listing Control Pseudo Opcodes

Several pseudo opcodes exist that control your assembly listing; you will place these pseudo opcodes in your source program. We list them briefly below. For more information on each pseudo opcode, see Chapter 5.

OBJNAM - Allows you to modify the name of your assembly listing disk file.
LIST - Re-enables output to the listing file.
NOLIST - Turns off output to the assembly listing file. (LIST and NOLIST are ignored if you use the /X switch.)
PAGE - Begins a new page in the assembly listing.

9.4.3 Generating a Cross Reference

You may use the /R switch to generate a cross reference as part of the assembly listing. To see the cross reference on your terminal, use the /RT switches. You may specify the /O switch to bypass assembly Phases 1 and 2 if an object (.OBJ) file for the current source file already exists.

NOTE: For information on using the GLOBAL command to generate a global cross reference, see Chapter 12.

9.4.3.1 Cross Reference Control Pseudo Opcodes - The CREF, MAYCREF, and NOCREF pseudo opcodes control the generation of the cross reference listing:

CREF	Enables normal cross referencing.
NOCREF	Suppresses from the listing all defined symbols until MACRO encounters a CREF or MAYCREF pseudo-op.
MAYCREF	Suppresses from the listing all defined symbols if those symbols are never referenced.

9.4.3.2 Cross Reference Listing Format - The cross reference listing is similar to an ordinary assembly listing except that it also includes the following:

1. A column of sequence numbers appears at the left of the listing.

2. At the end of the assembly listing, an alphabetic listing of each symbol appears giving, in numeric order, the sequence numbers of the lines in which each symbol appears. These sequence numbers are sometimes followed by a code of the form -X, where X identifies the type of symbol. X may be one of the following:

   L - a label definition
   E - an equate definition
   I - an INTERNed symbol
   X - an EXTERNed symbol
   O - an OVRLAY.

   Also, a single quote (') appears after symbols that were never defined. (MACRO will automatically EXTERN such symbols if the AUTOEXTERN pseudo opcode is present in your source program.)

3. A similar listing of macro definitions and references follows the symbol listing. (The sequence number corresponding to a macro definition is flagged by a "-M" code.)

9.4.3.3 Sample Cross Reference Listing - Remember that the cross reference appears at the end of a regular assembly listing. Below is a sample of what the cross reference portion of the assembly listing for a small program, MATH.MAC, might look like:

```
MATH    [110,5]          CROSS REFERENCE LISTING          PAGE 001
ACCUM   394     434     520     530     542     543     553     554
ADD     423     520-L
DIVI    429     553-L
EXIT    365     370     459     597-L
GETEXP  364-L
GETNUM'         386     415
NUMERR  393     416     567-L
OPRERR  407     411     583-L
OPRTBL  468     613-L
PARSE   383-L
PRTNUM'         441     450
START   354-L
SUB     425     530-L
S..RDX  30-E    309
$VAL    615     616-E   618     619     619-E   623     626     627
        643     643-E

MATH    [110,5]          CROSS REFERENCE LISTING          PAGE 002
BYP     181-M   366     383     398     412     494     503
CRLF    173-M   457
EXIT    174-M   573     589     597
GTDEC   186-M   496
OPERAT  331-M   613     621     629     637
TYPECR  292-M   451     567     583
```

Notice that the cross reference above identifies equated symbols and macro definition symbols. It also identifies the GETNUM and PRTNUM symbols as undefined or automatically EXTERNed symbols.

9.5 MACRO ERRORS

MACRO displays two types of error messages: errors codes that appear in your assembly listing and error messages that appear on your terminal screen as you assemble the program.

9.5.1 Error Codes

Below are the error codes that can appear in your assembly listing. Each code appears on the line of the source program in which the error occurred.

A Branch address was out of the 127-word range.

B Boundary error - a word operand was on an odd byte address. (See Chapter 5 for information on the EVEN pseudo opcode.)

C Conditional statement syntax error.

D Duplicate user symbol. (Symbol defined more than once.)

I Illegal character in source line.

M Missing term or operator in operand or expression.

N Numeric error which indicates a digit out of the current radix range.

P An expression that had to be resolvable on the first pass was not.

Q Questionable syntax - this is a general catch-all error code.

R Register error - a register expression was not in the range of 0-7.

T Source line or operand terminated improperly.

U Undefined user symbol during Pass 2.

V Value of an absolute parameter was out of its defined range.

X Assembler system error - please notify Alpha Micro.

9.5.2 Error Messages

You may see several error messages during the program assembly:

INVALID CONTROL PARAMETER VALUE
: You used the /V assembly switch to specify a value on the MACRO command line, but something was wrong with the format of the option request. For example, the value after the /V switch was missing or incorrect.

?Cannot OPEN Devn: - invalid filename
: There is something wrong with the format of your command line. For example, you may have tried to use an assembly switch but forgot to place it at the end of the file specification. All switches must appear at the end of the command line.

?File specification error
: There is something wrong with the format of your command line. For example, you typed MACRO followed by a RETURN (omitting the file specification).

?MACn.OVR not found
: where n is a number from 0 to 5. MACRO cannot find one of the overlays that are a part of MACRO. Make sure that the missing file is in account DSK0:[1,4]. If the file is not there, contact the System Operator.

?Copy file filespec not found
: where filespec is the file specification you supplied to the COPY pseudo opcode. For detailed information on the search pattern MACRO now uses to search for the copy file, see Section 5.1.1, "COPY."

?Expression stack error
: This is an internal MACRO error. You should never see it-- but if you do, check your source program to see if you made any errors in specifying expressions.

[SYNC ERROR]
: MACRO generates a listing file by reading the source file and the object file and synchronizing the source lines with the resolved object data to come up with the listing line data. If these two files get out of sync, there is no way that the listing may proceed and the message [SYNC ERROR] appears on your terminal. MACRO will then close the list file at the point of the sync error, but the line that caused the error will not have been included. A sync error of this sort means one of two things: either you have an out-of-date object file that you are using with the /O switch, or you have found an undiagnosed assembler bug. These bugs usually occur when you get fancy with nested macros and conditionals that have a valid error buried down deep within.

NOTE: The most probable cause for this error is that you are using an object file that was generated by a different version of MACRO than the one you are using now. If you see no obvious errors in your program, try generating a listing without the /O switch (thus building a new object file). If you still get [SYNC ERROR], report the problem to Alpha Micro.

CHAPTER 10

THE LINKAGE EDITOR (LINK) AND SYMBOL TABLE FILE GENERATOR (SYMBOL)

This chapter contains information on the linkage editor LINK and the symbol table generator program SYMBOL. We discuss both of these programs at this time because LINK and SYMBOL are very similar and, with the proper selection of option switches, can be made to perform virtually the same functions. LINK takes one or more object files produced by the assembler and resolves all external symbol references. The file that LINK produces is the final, executable program file. SYMBOL takes one or more object files and produces a symbol table file for that program. As we will see later, LINK and SYMBOL can also perform other functions.

Besides discussing how to link .OBJ files, this chapter also discusses the use of LINK and SYMBOL with library (.LIB) files. For more information on object file libraries, see Chapter 11, "The Object File Library Generator (LIB)."

10.1 LINK

The assembler itself does not produce a file that is directly usable as an executable program. (Unless of course, your program is a single segment file that contains no EXTERNed, INTERNed, or AUTOEXTERNed symbols, in which case MACRO calls LINK as Phase 4 of the assembly.)

The assembler output file is an object (.OBJ) file that is not fully resolved and which contains symbol definitions and embedded cross-segment commands.

It is the linkage editor (LINK) that resolves the object file. LINK reads one or more of these object files and creates one runnable program (.PRG) file which the operating system can load into memory and run. Furthermore, if the program contains overlay segments, LINK resolves them and creates one overlay (.OVR) file for each one. These overlay files are loaded into memory upon command during the running of the program and allow memory conservation for large programs such as the assembler itself.

We mentioned previously that if your program has only one segment, MACRO automatically calls the linkage editor to create a program file (as Phase 4 of the assembly). In this case, no further action is necessary and you are ready to run the program. If, however, the program is comprised of more than one segment, you must run the LINK program yourself, specifying the name and order of the segment files involved.

10.1.1 LINK Command Line

The general format of the LINK command is:

.LINK {/switches }filespec1{,filespec2,...filespecN}{/switches} RET

where filespec selects an object file. The default extension is .OBJ. The first file specified may not be a library file or an overlay file. If a filespec includes a device and account specification, LINK searches for the file in that account. If you omit a device and account specification, LINK searches for the file first in the account and device you are logged into; secondly, in your project library account (account [P,0]); and, finally, in the System Macro Library account, DSK0:[7,7].

LINK treats switches in the same way that a standard AMOS wildcard command does; this means that the files affected by the option switches you use can depend on where you place the switches. Any switch that appears in front of a filespec becomes the default switch and thus affects the rest of the filespecs on the command line (unless canceled by a subsequent switch). Any switch that appears at the end of a filespec affects only the files selected by that specification. For example, suppose you want to use the /O switch to identify one or more .OBJ files as optional files:

.LINK FILBCK,/O DIRBCK,TAPBCK RET

selects the files DIRBCK and TAPBCK as optional files because the /O switch precedes the filespec DIRBCK, and thus becomes the default. The command line:

.LINK FILBCK,DIRBCK/O,TAPBCK RET

selects only the file DIRBCK as an optional file because the /O switch follows the DIRBCK filespec and appears before the next comma in the command line.

NOTE: Special switches (identified as "operation switches" in the discussions below) affect ALL filespecs specified on the command line no matter where you place the switch. For example, it doesn't matter where you place the /M switch on the command line-- it affects all files selected by the filespecs on the command line.

10.1.1.1 Continuation Lines - If the program you want to link contains more files than will fit on the command line, you may continue the files on the next line by terminating the last filespec with a comma. LINK continues to accept files as long as the last filespec on the line terminates with a comma.

10.1.1.2 LINK Options

/E Include equated symbols in the symbol table file. (You must use /E with the /M or /S switch.) (Operation switch.)

/L Designates a library file. See Section 10.3, "Library and Optional Files," for information on library files.

/M Generate a load map (.MAP) file. See Section 10.4, "The Load Map File," for a discussion of the load map. (Operation switch.)

/N Suppress /P switch. (Operation switch.)

/O Designates an optional file. See Section 10.3, "Library and Optional Files," for information on optional files.

/P Generate program (.PRG) and overlay (.OVR) files. The default switch. (Operation switch.)

/R Designates a required file. The default switch. Cancels the /L and /O switches.

/S Generate a symbol table (.SYM) file. (Operation switch.)

You may specify multiple switches by preceding each switch with a /. (See the command line below.)

10.1.2 Sample LINK Display

Below is a sample LINK display. Note that we are using the /L switch to specify a library file, and are using the /M switch to generate a load map.

```
.LINK MATH,UTILIT.LIB/L/M (RET)

== Linkage Editor Version 2.0 ==

Processing MATH.OBJ  [Base = 0, Size = 348. bytes]

-- Optional and Library Request --

Processing UTILIT.LIB(NUM) [Base = 534, Size = 144. bytes]

Program and Map files finished. [Program size = 492. bytes]
.
```

Notice that LINK tells you the size (in decimal bytes) of each module. If you specify a library file, LINK tells you which of the object files in the library file are being linked in. (In the sample above, LINK linked in the NUM routine from the UTILIT.LIB library file.)

10.1.3 LINK Errors

LINK reads each of the files specified and creates the necessary program and optional overlay files. LINK displays any error messages on the terminal if it encounters any errors during processing. The most common error is the undefined global symbol error which means you have an EXTERN symbol in one segment which is not defined in another segment by an INTERN statement. LINK does not generate a program file if it cannot find one or more of the segments in its assembled object (.OBJ) form. For a list of the LINK error messages, see Section 10.5.

10.2 THE SYMBOL TABLE FILE GENERATOR (SYMBOL)

The object files output by the assembler contain complete information on the symbols used in your program, as well as the actual generated code. To make this list of symbols available to the debugger programs, you must use the SYMBOL program. Just like LINK, the SYMBOL program takes one or more .OBJ files and creates an output file, in this case a symbol (.SYM) file. DDT and FIX use this file to provide symbolic debugging of programs.

Unlike the program file, the symbol file is not generated automatically even if only one program segment is used. You must explicitly run SYMBOL if you wish to create a symbol file.

10.2.1 SYMBOL Command Line

The format for calling SYMBOL is identical to the LINK command line:

.SYMBOL {/switches }filespec1{,filespec2,...filespecN}{/switches} RET

where filespec selects an object file. The default extension is .OBJ. The first file specified may not be a library file or an overlay file. If a filespec includes a device and account specification, SYMBOL searches for the file in that account. If you omit a device and account specification, SYMBOL searches for the file first in the account and device you are logged into; secondly, in your project library account (account [P,0]); and, finally, in the System Macro Library account, DSK0:[7,7].

SYMBOL treats switches in the same way that a standard AMOS wildcard command does; this means that the files affected by the option switches you use depends on where you place the switches. Any switch that appears in front of a filespec becomes the default switch and thus affects the rest of the filespecs on the command line (unless canceled by a subsequent switch). Any switch that appears at the end of a filespec affects only the files selected by that specification. For example, suppose you want to use the /O switch to identify one or more .OBJ files as optional files:

.SYMBOL MAIN,/O SUB1,SUB2 RET

selects the files SUB1 and SUB2 as optional files because the /O switch precedes the filespec SUB1, and thus becomes the default. The command line:

.SYMBOL MAIN,SUB1/O,SUB2 RET

selects only the file SUB1 as an optional file because the /O switch follows the SUB1 filespec and appears before the next comma in the command line.

NOTE: Special switches (identified as "operation switches" in the discussions below) affect ALL filespecs specified on the command line no matter where you place the switch. For example, it doesn't matter where you place the /M switch on the command line-- it affects all files selected by the filespecs on the command line.

The output of SYMBOL is placed into a file named filespec.SYM, where filespec is the first file specified on the SYMBOL command line. No symbol file will be generated if one or more of the specified files is not found in its assembled object (.OBJ file) form. (NOTE: You may use the OBJNAM pseudo opcode within your .MAC file to modify the name used for the SYMBOL output file. See Section 5.1.2, "OBJNAM.")

10.2.1.1 Continuation Lines - As with LINK, if the program contains more files than will fit on the command line, you may continue the file specifications on the next line by terminating the last filespec with a comma. SYMBOL will continue to accept filespecs as long as the last filespec on the line terminates with a comma.

10.2.1.2 SYMBOL Options

/E Include equated symbols in the symbol table file. You may also use this switch with /M to tell SYMBOL to include equated symbols in the load map. (Operation switch.)

/L Designates a library file. See Section 10.3, "Library and Optional Files," for information on library files.

/M Generate a load map (.MAP) file. See Section 10.4, "The Load Map File," below, for a discussion of the load map. (Operation switch.)

/N Suppress /S switch. (Operation switch.)

/O Designates an optional file. See Section 10.3, "Library and Optional Files," below, for information on optional files.

/P Generate program (.PRG) and overlay (.OVR) files. (Operation switch.)

/R Designates a required file. The default switch. Cancels the affect of a /L or /O switch.

/S Generate a symbol table (.SYM) file. The default switch. (Operation switch.)

You may specify multiple switches by preceding each switch with a /. (See the command line below.)

10.2.2 Sample SYMBOL Display

Below is a sample SYMBOL display. Note that we are using the /L switch to specify a library file, and are using the /M switch to generate a load map.

```
.SYMBOL MATH,UTILIT.LIB/L/M (RET)

== Linkage Editor Version 2.0 ==

Processing MATH.OBJ

-- Optional and Library Request --

Processing UTILIT.LIB(NUM)

Symbol and Map files finished.
.
```

If you specify a library file, SYMBOL tells you which of the object files in the library file it is including in the symbol table file. (In the sample above, SYMBOL included the NUM routine from the UTILIT.LIB library file.)

NOTE: If you compare this display with that of the LINK program (Section 10.1.2, "Sample LINK Display," above), you will notice that it is very similar. In fact, LINK and SYMBOL can be made to perform exactly the same functions. If we had specified the /P switch and the /N switch (specifying that we wanted a .PRG file generated and did not want a symbol table file), the display above would have looked exactly like the LINK display in Section 10.1.2.

10.3 LIBRARY AND OPTIONAL FILES

Both LINK and SYMBOL support the use of library files and optional files.

Most programmers have been faced at one time or another with the task of having to write a standard routine again and again for multiple programs. Library and optional files help you to avoid this situation by allowing your programs to contain references to previously written routines in an object file library or an optional file.

Besides making your life easier by making it possible for you to write frequently used routines only once, library and optional files also help to standardize programs by providing the same error checking, input checking, message display, etc., for multiple programs.

LINK and SYMBOL place any object files from a library file and any optional files at the end of your program in the order that they are needed to resolve external references.

10.3.1 Library Files

A library file is a file produced by the LIB program (discussed in the next chapter). The library file contains a group of .OBJ files. The purpose of generating a library file is to gather together a group of subroutines that are frequently used by programs on your system. These routines are then easily accessed by all programmers on the system by using the EXTERN or AUTOEXTERN pseudo opcodes in their source programs and specifying the required routine. Unlike using the COPY pseudo opcode, which physically incorporates the entire source file specified by the COPY statement into your assembled program when you assemble it, using a library file causes only those subroutines within the library file that are referenced by your program to be linked into your program.

For example, if a library file contains the following object files: SWTCH, SPACE, STRCHK, and GETLIN, and the program you link with the library file only references the routine GETLIN, only the object code for that routine will be linked into your program.

IMPORTANT NOTE: You should note that the entire .OBJ file that is a component of a library file will be linked in if your program references a symbol in it; not just that portion of the .OBJ file required by your program. For example, suppose you create a library file (using LIB) that contains the following .OBJ files: STRCHK, GETLIN, and GETNUM. If your program references a symbol within the GETNUM object file, the entire GETNUM file is linked in even if it also contains several other routines. For this reason, you should limit each .OBJ file that is a component of the library to only one subroutine.

You may not specify the library file first on the LINK or SYMBOL command line. (This is because to resolve symbol references, LINK and SYMBOL must first access the file that makes those references before it accesses the file that defines them.)

10.3.2 Optional Files

By using the /O switch with LINK or SYMBOL, you may request that the specified file (called an "optional file") be included in the linked program if the optional file is needed to resolve any external references in one of the other files being linked. If such a reference exists, the optional file will be incorporated into your program; otherwise, it will not. Unlike a library file, an optional file only contains the contents of a single .OBJ file. An optional file may not be an overlay.

10.4 THE LOAD MAP FILE

A load map shows how the modules linked together will be loaded into memory when the program is invoked for execution. Using the /M switch with LINK or SYMBOL, you may ask that a load map file be generated. A load map file has the name of the first file specified on the LINK or SYMBOL command line and the extension .MAP.

A load map lists each object file used in the order that it was used. For each object file, the load map gives the following information:

1. The file's offset from the beginning of the program;

2. the size of the file in decimal bytes;

3. in alphabetic order, all the symbols defined in that file and their relocated values after the linking process. If the symbols are relocatable relative to the base of the program, the load map flags them with a "r" symbol.

For example, the following LINK command line:

```
.LINK MATH,NUM/M RET
```

generated the load map file below, MATH.MAP:

```
[Linkage Editor Version 1.0]
Program Load Map
================
```

Module	Base	Size	Symbol	Value	Symbol	Value	Symbol	Value
MATH	000000r	348.	ACCUM	000520r	ADD	000330r	BASCHG	000262r
			BASE	000516r	DIVI	000362r	EXIT	000514r
			GETEXP	000006r	GETOPR	000224r	MULT	000344r
			NUMERR	000406r	OPRERR	000446r	OPRTBL	000522r
			PARSE	000024r	START	000000r	SUB	000336r
NUM	000534r	144.	CHGTBL	000706r	GETNUM	000534r	PRTNUM	000616r

10.5 LINK AND SYMBOL ERROR MESSAGES

?Command error
: There was something wrong with your command line. For example, you tried to use LINK or SYMBOL without specifying a file on which to work.

?Fatal error - Insufficient memory
: You must increase the size of your memory partition; there was not enough room to perform the procedure you specified.

?Undefined switch /x - ignored

Refer to Appendix B, "Summary of Program Switches," to make sure that you specified a valid switch.

?Fatal error - Overlays of code are not permitted
Next expected address is xxxx
Overlay code address is xxxx

Your program is trying to overlay previous code. Check your .MAC programs to make sure that your overlay references are correct.

?xxxx undefined

An external symbol is undefined. This is a very common error. You have referenced a symbol which has not previously been defined (e.g., you have made a reference to a label that does not exist). Make sure that an EXTERN statement in one segment is defined by an INTERN statement in another segment.

?Fatal error - First file must not be a library

To enable LINK or SYMBOL to correctly resolve external references to a library, you must specify the program that references that library before you specify the library file itself.

?Fatal error - Attempt to specify overlay xxx as optional

You may not use the /O switch to designate a file as optional if that object file is an overlay.

?Fatal error - Overlay symbol "xxxx" in segment yyyy
was not defined in a previous input segment

You may not reference an undefined overlay. In other words, LINK is trying to process a supposed overlay file, but has seen no references to the overlay in a previous file. Without such a reference, LINK cannot construct the overlay, so it aborts and returns you to AMOS command level.

?Fatal error - First file must not be an overlay

To enable LINK or SYMBOL to correctly resolve external references to an overlay, you must specify the program that references that overlay before you specify the overlay file itself.

?Fatal error - Expression stack error

An error occurred when LINK or SYMBOL evaluated some expressions in your files. This indicates an internal error-- you should never see this error message.

?Fatal error - Expression stack overflow

You exceeded the number of nested expressions that LINK or SYMBOL can handle. Try to find the exceedingly complex expression in your source file and simplify it.

CHAPTER 11

THE OBJECT FILE LIBRARY GENERATOR (LIB)

One of the more aggravating programming tasks is rewriting a utility program that you've used many times before and that you know you will use many times in the future. Many kinds of routines are so useful that you need them again and again in many different programs: e.g., routines that check for ASCII characters, that input and output characters, that sort data, etc.

The purpose of the library file is to collect together these frequently used routines where they can be accessible to your program files when you link them into final, executable programs. Not only do library files help you to avoid writing and rewriting the same routines over and over, but they can also give help to every other programmer on the system. An added benefit of library files is that they tend to help standardize programs on the system by providing standard input, output, error checking, and message display routines used by everyone on the system.

The Alpha Micro object file library generator, LIB, constructs library files out of .OBJ files. Each of the .OBJ files which is built into the library file is a separate routine that can be accessed by your programs. The final library file has a .LIB extension and can be used by both LINK and SYMBOL.

11.1 LIB COMMAND LINE

The LIB command line takes one of two forms:

```
.LIB{/L} output=input1{,input2,...inputN} RET
or:
.LIB(/L} inout{,input2,...inputN} RET
```

(The second format is equivalent to: LIB inout=inout{,input2,...inputN} if you do not use the /L switch; otherwise, it is equivalent to: TRM:=inout{,input2,...inputN}.)

"Output" is an output file specification; it specifies the name of your library file. The output file has the extension .LIB and the name specified by the output or inout specification.

"Input" specifies the .OBJ files you want to place in the library. The input specification can take the following forms:

```
filespec
filespec\item1
filespec\(item1,item2,...itemN)
filespec(item1,item2,...itemN)
```

The \ symbol designates an exception. For example, in the command line:

```
.LIB MYLIB\SUB1,NEWSUB,READIT (RET)
```

tells LIB that we want to modify the existing library MYLIB (the "inout" specification) by removing the object file SUB1, and adding NEWSUB and READIT.

The parentheses specify a group of object files. For example:

```
.LIB MYLIB\(SUB1,NEWSUB,READIT),GETNUM (RET)
```

tells LIB to modify the existing library MYLIB by deleting the collection of object files SUB1, NEWSUB, and READIT, and to add the object file GETNUM.

LIB looks for the specified files in the account and device specified. If you omit the device and account specification from the filespec, LIB searches first in the account and device you are logged into; then your project library account on the device you are logged into (account [P,0]); finally, LIB searches in the System Macro Library account, DSK0:[7,7].

11.1.1 Continuation Lines

As with LINK and SYMBOL, you may enter as many filespecs as you wish on as many lines as you wish as long as you end the last filespec on the line with a comma.

11.1.2 LIB Option Switch (/L)

The only LIB switch at this time is the /L switch which tells LIB to generate a library listing. This listing looks similar to a load map listing (see Section 10.4., "The Load Map File."), and lists all object files in the library file and all INTERNed symbols.

If you specify an output file (e.g., LIB LIST=MYLIB/L) LIB creates the listing with the name and extension you specified. (The default extension is .LST.) If you do not specify an output file (e.g., LIB MYLIB/L), LIB sends the library listing to your terminal display.

11.2 SAMPLE LIB DISPLAY

Suppose we are creating a new library called USEFUL from the .OBJ files ERRMSG, GETLIN, and FORMAT:

```
.LIB USEFUL=GETLIN,FORMAT,ERRMSG (RET)

== Object File Librarian Version 1.0 ==

Processing GETLIN.OBJ
Processing FORMAT.OBJ
Processing ERRMSG.OBJ

Library file finished
.
```

As LIB processes each new .OBJ file, it tells you so.

Suppose we want to add a routine to an existing library. The sample display might look like this:

```
.LIB USEFUL,LINSIZ (RET)

== Object File Librarian Version 1.0 ==

Processing USEFUL.LIB(GETLIN)
Processing USEFUL.LIB(FORMAT)
Processing USEFUL.LIB(ERRMSG)
Processing LINSIZ.OBJ

Library file finished
.
```

We've sucessfully added the new routine LINSIZ to our old library that already contained the object files GETLIN, FORMAT, and ERRMSG. Notice that LIB tells you as it processes each .OBJ file contained within the library file.

11.3 UPDATING A LIBRARY

Replacing one or more of the .OBJ files that make up a library file can be a bit tricky. If you simply try to add a new version of an existing .OBJ file without deleting the old one first, problems can result because both versions of the object file will be in the library. The recommended procedure is to first delete the old routine, and then to add the new one. For example, if we wish to replace the old version of FORMAT with a new one, we enter:

```
.LIB USEFUL\FORMAT, FORMAT (RET)
```

which first deletes the file and then adds it. Assume that our small library only contains three routines, GETLIN, ERRMSG, and FORMAT. The LIB display in response to this command line would look like this:

```
== Object File Librarian Version 1.0 ==

Processing USEFUL.LIB(GETLIN)
Processing USEFUL.LIB(ERRMSG)
Processing FORMAT.OBJ

Library file finished
*
```

Notice that LIB tells you what routines are contained in the library.

11.4 LIB ERROR MESSAGES

You may see the following error messages when you use LIB:

?Command error
: LIB did not understand your command line. For example, you entered LIB followed by a RETURN. Make sure that your file specifications are in standard form.

?Undefined switch /X - ignored
: where X is the switch you supplied. LIB currently uses only one option switch, /L, to produce a library listing. Make sure that you did not type a / by accident when you wanted to type a backslash.

?OBJ files are not libraries -- they can not be restricted with a modifier
: You may only use the "\" file restrictor and the "()" file inclusion symbols if you are modifying a library.

?Listing aborted
: LIB was not able to finish the library listing. For example, an error occurred while LIB was trying to access a file.

?The following module was not found - xxx
: You tried to modify an existing library, but the object files you specified were not present in the library file. Make sure that you did not accidentally use the \ restrictor symbol.

?Fatal error - xxx is an overlay
: You may not specify an overlay as an element of an object file library.

CHAPTER 12

THE GLOBAL CROSS REFERENCE GENERATOR (GLOBAL)

The GLOBAL program takes a group of object (.OBJ) files and produces an alphabetic global cross reference which lists all global symbols in the files, and shows which files define those symbols and which files accept them as externally defined symbols.

In other words, GLOBAL produces a listing file that contains a cross reference of all symbols that have been referenced in an INTERN, EXTERN, or OVRLAY statement so that you can see in which .OBJ files these references occur. (NOTE: GLOBAL produces a cross reference of all global symbols for a collection of .OBJ files. Remember that you can also see a cross reference listing as part of your assembly listing for all global and local symbols for an individual .OBJ file by specifying the MACRO /R switch when you assemble the file.)

GLOBAL is particularly useful when you want to find out what references are made to symbols between files. The /R assembly switch is most useful when you want more detailed information about a single .OBJ file.

NOTE: GLOBAL does not support library files.

12.1 GLOBAL COMMAND LINE

The GLOBAL command line takes this form:

.GLOBAL{/switches} filespec1,filespec2{,...filespecN} RET

where switches are optional and affect the format of the information in the listing file. Filespec1...filespecN is a list of file specifications that select the .OBJ files for which you want the global cross reference.

If you omit the extension from a file specification, GLOBAL uses the default extension of .OBJ.

GLOBAL produces the listing file in the account and device you are logged into with the name of the first file specification on the command line and a .GLB extension.

12.1.1 Continuation Lines

If there are too many file specifications to fit on one line, you may end the command line with a comma. GLOBAL continues to accept file specifications as long as the last filespec on the line ends with a comma. If the last filespec on the line ends with a comma, GLOBAL prompts you with an asterisk for more filespecs. For example:

```
.GLOBAL MAIN,SUB1,SUB2,SUB3,SUB4, (RET)
*SUB5,SUB6 (RET)
```

12.1.2 GLOBAL Options

You may request the following options by including the appropriate switches on your command line:

Line width options (default is 80 characters):

/W	Wide listing (same as /W:130). Produces a listing file that may have up to 130 characters on a line.
/W:n	Specifies characters per line, where n specifies the number of characters.

Page length options (default is 60 lines):

/L	Long listing (same as /L:80).
/L:n	Specifies lines per page, where n specifies the number of lines.

Each switch must begin with a slash. For example:

```
.GLOBAL/W/L MAIN,SUB1,SUB2 (RET)
```

12.2 SAMPLE GLOBAL DISPLAY

As GLOBAL processes the specified files, it displays a message telling you so ("Processing filespec"). After it processes all files, GLOBAL produces a .GLB file; as it works, it displays the name of the file it is building and displays a dot for each disk block it outputs. For example:

```
Building MAIN.GLB ....
```

This file has the same name as the first file you specified on the GLOBAL command line.

Below is a sample GLOBAL display:

```
.GLOBAL MAIN,SUB1,SUB2,SUB3 (RET)

== Global Cross Referencer (Version 2.0) ==

Processing MAIN.OBJ
Processing SUB1.OBJ
Processing SUB2.OBJ
Processing SUB3.OBJ

Building MAIN.GLB....

Global file finished
.
```

If GLOBAL found any reference errors, it tells you so. For example:

```
Global file finished, 2 errors exist
```

12.3 SAMPLE LISTING DISPLAY

The listing file that GLOBAL produces lists each defined symbol, and what .OBJ file the symbol was referenced in. The listing tells you whether the symbol was referenced as an internal symbol (I) via an INTERN pseudo opcode, an external symbol (E) via an EXTERN or AUTOEXTERN pseudo opcode, or an overlay symbol (O) via an OVRLAY pseudo opcode.

Here is a portion of what a GLOBAL listing file might look like:

```
Global Cross-Reference (Version 2.0)

          M S S S
          A U U U
          I B B B
          N 1 2 3
          - - - -
ALPHA     I E . E
BETA      I . E .
ZETA      I O . .
```

The listing file above tells us: 1) the symbol ALPHA appeared in an INTERN statement in the file MAIN.OBJ and in EXTERN statements in the files SUB1.OBJ and SUB3.OBJ; 2) the symbol BETA appeared in an INTERN statement in MAIN.OBJ and in an EXTERN statement in SUB2.OBJ; 3) the symbol ZETA appeared in an INTERN statement in MAIN.OBJ and in an OVRLAY statement in SUB1.OBJ.

12.4 GLOBAL ERROR MESSAGES

You may see the following error messages when using GLOBAL:

?Undefined switch /X - ignored
: You specified an invalid switch. The only switches GLOBAL recognizes are the /L and /W switches.

?Cannot OPEN filespec - not found
: GLOBAL could not find the file you specified. Make sure you are logged into the correct account on the right device.

CHAPTER 13

THE SYMBOLIC DEBUGGER (DDT)

A debugger is a program that helps you to test and examine a new program. The Alpha Micro system contains two dynamic debugger programs for assembly language programs: 1) AlphaFIX, a screen-oriented debugging program; and 2) DDT, a debugging and patching program. For information on AlphaFIX, see the AlphaFIX User's Manual, (DWM-00100-69). (AlphaFIX users please note Section 13.4.1.2, below, which discusses using local symbols with both DDT and AlphaFIX.)

The rest of this chapter discusses the operation of DDT. DDT is the AMOS dynamic debugging and patching program. It allows you to run your program and to examine or alter program data or flow at any point in the program. All of the examination and modification may be done via symbols, both on type-in and type-out. DDT automatically expands your program in memory to accommodate patches. This expansion capability, along with the ability to define new symbols, makes it easy to patch existing programs. As a matter of fact, all Alpha Micro system software patches are implemented using DDT.

NOTE: Most DDT commands terminate with an Escape. DDT echoes Escapes as dollar signs. (That is, when you press the ESCAPE key (sometimes labeled ALT MODE or ESC on your keyboard), DDT repeats the Escape as a $ symbol.) Except for our discussion of local symbols, whenever you see a dollar sign symbol in the discussions below, keep in mind that it represents the place in your command input where you should type an Escape.

13.1 THE DDT COMMAND LINE

You may use DDT on any program, whether it contains executable code or not. Its most common use will be with program (.PRG) files produced by the linkage editor. To invoke DDT, type:

```
.DDT filespec (RET)
```

where filespec specifies the file you want to debug. If you omit the extension, DDT uses the default extension .PRG. When DDT is called, the first thing it does is check to see if the specified file is already in

memory. If it is, the file is deleted from memory. The program is then loaded into memory ensuring that a fresh copy is now resident, and DDT proceeds to look for a symbol file.

Once DDT has loaded the program file and any associated symbol file, it prints the base memory address and the size in bytes of the program being debugged. For example:

```
.DDT DEVCPY.PRG (RET)
PROGRAM BASE: 32777
PROGRAM SIZE: 400
```

Now you can begin to enter the DDT commands discussed below. For information on exiting DDT, see Section 13.9, "Exiting DDT."

13.2 USING SYMBOL FILES

After loading the actual program to be debugged into memory, DDT searches for a symbol file. If one is currently in memory, DDT deletes it. DDT then searches your account for a file with the same name as the specified program file, but an extension of .SYM. If one is found, it is loaded into memory, and debugging can start. If no symbol file is found, DDT assumes that you wish to debug without user symbols and enters debug mode without a symbol table.

13.3 TERMINAL INPUT

Because DDT must accept characters on an individual basis, it runs in terminal image mode. This mode disables the usual functions of RUBOUT, Control-U, Control-S, Control-Q, etc. However, Control-C will still abort DDT and return you to AMOS. RUBOUT takes on a special meaning in DDT. Instead of the standard function of erasing the last character typed, RUBOUT in DDT will cancel the entire current command, and echoes as "XXX" followed by a tab.

13.4 EXPRESSIONS

DDT allows both input and output expressions to be in either numeric or symbolic form. The majority of commands will accept or display in either mode, although certain arguments, such as a breakpoint number, must be provided as a numeric value.

13.4.1 Input Expressions

Most commands will accept an expression whenever they require input. All numeric input to DDT is in octal. Both symbolic and numeric expressions can use the plus (+) or minus (-) operators. The following are all valid DDT input expressions:

```
123
12343+57725
TAG
TAG+77
TAG+IT
```

Where TAG and IT are defined symbols.

13.4.1.1 Special Symbols - In addition to the symbols defined in the program being debugged, DDT recognizes several special symbols in input expressions. In register mode, DDT recognizes the register names R0, R1, R2, R3, R4, R5, SP, and PC. In program-relative mode, DDT recognizes the special symbol dot (.) as being equal to the currently open location. Dot allows you to use relative offsets in an expression:

```
.+40/   MOV 7,R1        BR .+20
.$B
```

The above example of using dot in a breakpoint command ($B) is one of the most frequent uses of the special symbol dot.

13.4.1.2 Local Symbols - DDT correctly displays local symbols if the appropriate symbol table file is available. (If your version of DDT displays local symbols as garbled RAD50 names that begin with a colon, you have an obsolete version of DDT.) (For information on using local symbols in your source programs, see Sections 4.7 and 6.1.6.)

NOTE: Local symbols take the form nnn$. In the examples below, notice that a dollar sign preceding a character indicates a normal DDT command in which the dollar sign designates an Escape (for example: $A indicates Escape-A). When a dollar sign follows a character (e.g., 10$), we are talking about a local symbol.

DDT searches for local symbols by looking backward from the current open location to the first non-local symbol and then scanning forward from that location to the next non-local symbol. The local symbol you are looking for must fall within that region.

To access a local symbol, you must first set the current location counter to a location in the region containing the local symbol. (Remember that a local symbol only has scope between two non-local symbols. This is its "region.") You will probably want to simply open the location at the

non-local symbol that appears just before the local symbol; then you can access the symbol that is local to it. For example: The $A command displays a string of ASCII characters at the current location or at the location of the symbolic argument supplied:

```
LABEL$A
```

tells the $A command to use the location at "LABEL", a non-local symbol. If we want to see the ASCII characters at the local symbol 10$ which lies between LABEL and LABEL1, we would first open the non-local symbol that precedes 10$:

```
LABEL/
```

Now we can access 10$, which is local to the non-local symbol LABEL by entering the local symbol "10$" followed by the command "Escape-A":

```
10$$A
```

DDT also accepts a local symbol when assembling an instruction, searching for it in the range where the instruction is being assembled.

NOTE FOR ALPHA FIX USERS: FIX also correctly displays local symbols. Any of the FIX commands that allow you to specify non-local symbols may also be used to access local symbols. Just follow the non-local symbol with a space; then enter the symbol you want to access that is local to that non-local symbol. For example:

```
>S START 10$ RET
```

tells FIX to search for the symbol 10$ that is local to the non-local symbol START.

13.4.2 Output Expressions

DDT outputs data in both symbolic and numeric format. When in program-relative mode, DDT displays memory locations in symbolic form; in register mode, it displays register contents in octal. All numeric output, even when combined in a symbolic output expression (such as JMP TAG+12) will be in octal unless you have set J.HEX in your job status word via the SET HEX command, or you are executing a command which explicitly displays data in another radix (such as $D, the decimal typeout command).

13.5 DDT MODES

DDT has three modes in which it operates: program-relative mode, absolute mode, and register mode. The normal mode, and the one in which DDT initally comes up, is program-relative mode. In this mode, addresses are assumed to be relative to the base address of the program being debugged. Therefore, an expression of "12" refers to location 12 relative to the program base, not absolute location 12.

In absolute mode, all addresses are taken to represent absolute memory locations. In the example above, "12" would refer to absolute memory location 12, regardless of the fact that that location is outside of your memory partition as well as outside of the program being debugged. Absolute mode is entered via the TAB command, and left via the $R command.

In register mode, expressions refer to the registers instead of memory locations. Register mode may be entered by using one of the special symbols R0-R5, SP, or PC. Any of these symbols followed by a command which opens a location will enter register mode. Register mode may be left via the $R command.

13.6 DDT COMMANDS

DDT has a variety of commands to allow you to examine memory locations, change the contents of locations, display registers, set breakpoints, single-step, etc. Commands to DDT usually consist of giving a numeric or symbolic argument followed by a DDT command. Commands consist of single characters, such as the slash (/) command, and also of an Escape (ALTMODE on some terminals) followed by a single letter command, such as the Escape-B command. Escapes in DDT echo as a dollar-sign ($). The dollar-sign is used in this section to represent an Escape; therefore, when you see a command such as "$B", that should be interpreted as an Escape followed by a "B".

Several of the commands refer to opening and closing memory locations or registers. When a location or register is said to be "open," it simply means that DDT will place into the open item any expression entered through your terminal followed by a command that closes the location. This is the method by which memory or register contents are modified. When a location is "closed," you may no longer modify it by entering an expression without first opening the location again.

13.6.1 Opening a Location or Register (/)

The slash command (/) displays the current contents of a memory location or register and leaves that location open for modification. The slash command takes a symbolic or numeric argument immediately preceding the slash. The contents of the opened item will be displayed in symbolic form. The contents may be examined in other formats via other commands such as equal (=), Escape-D ($D), etc. The slash command will not open locations outside

of the program being debugged unless DDT is in absolute mode. The following shows a few examples of using the slash command:

```
TAG/     MOVI 7,R1              examine location TAG
TAG+12/  SET QFLG(B)            examine location TAG+12
R1/      46623                  examine register R1
```

13.6.2 Closing a Location (Carriage-Return)

The carriage-return (RET) command closes the current location. As with other commands which close a location, it may be immediately preceded by a number or symbolic expression which will be placed into the open location. Note that the expression given prior to the closing command may generate more than one word of data, in which case the extra words are placed in the locations immediately following the open one.

13.6.3 Display a Value in Octal (=)

The equal (=) command displays the contents of the currently open item in octal unless you have SET HEX, in which case the display will be in hexadecimal. The equal command may be used to convert a symbolic typeout to numeric, or may be used to compute the value of an expression. The following are all common uses of the equal command:

```
TAG/     MOVI 7,R1 = 004166     display contents in octal
TAG=3252                        find value of symbol
26662+15252=44134               compute an expression
.=24233                         display current location addr
```

13.6.4 Opening the Next Location (Line-Feed)

The line-feed (LF) command functions the same as the carriage-return command except that it opens the next location after closing the current one. Depending on your terminal, to enter a line-feed, press the down-arrow key on your terminal or the key labeled "LF" or "LINEFEED."

If the contents of the current location have been displayed in symbolic form, LF will advance to the location following the entire instruction displayed, regardless of length. This allows you to easily step through a program, without regard to opcode length. If the current location has been displayed in octal (via the = command) the LF command will step to the next word. If new data is entered prior to the LF command, the length of the data entered will determine the next location opened.

In register mode, a line-feed will step to the next register. If you step past PC, R0 will be reopened.

13.6.5 Opening the Previous Location (^)

The up-arrow (^) command will close the current location and open the location immediately preceding the current one. Unlike LF, up-arrow does not automatically open a location on a valid opcode boundary; up-arrow always backs up one word.

(NOTE: This command is not the key labeled with an up-arrow on your terminal keyboard-- it is the "^" symbol, the circumflex.)

13.6.6 Opening a Location Indirectly (@)

The at-sign (@) command treats the contents of the current open location as a program relative address and opens that location.

13.6.7 Opening an Absolute Location Indirectly (TAB)

The TAB (Control-I) command treats the contents of the current open location as an absolute address and opens that location. It also sets DDT into absolute address mode. DDT will remain in this mode until you execute an $R command.

13.6.8 Starting a Program ($G)

The Escape-G ($G) command starts the program being debugged at relative address 0. DDT echoes a tab after the $G, and waits for one line of input terminated by a carriage-return, prior to beginning actual execution of the program. This line of input is passed to the program just as if it had been entered following the command if the program were being run without DDT. The proceed ($P) and single-instruction ($X) commands are not legal until an $G command has been entered. You may execute an $G command at any time to restart the execution of the program. This assumes, of course, that the program being debugged is self-initializing so that the same copy can be run more than once.

13.6.9 Setting Breakpoints ($B)

The Escape-B ($B) command sets or lists breakpoints within the program. DDT allows up to eight breakpoints to be set in the program. Each breakpoint is assigned a number from 0 to 7. The $B command accepts two arguments: the numeric or symbolic program-relative address at which you wish to set a breakpoint, and the breakpoint number which you wish to place at this point. The program-relative address is given first, immediately preceding the Escape. The breakpoint number is given after the Escape, immediately preceding the B. Both of the arguments are optional. If the address is

omitted, the breakpoints are listed on your terminal. If the breakpoint number is omitted, the first available breakpoint is assigned. The following list should make things clear:

$B	Lists all active breakpoints by number and symbolic or numeric address.
$xB	Lists breakpoint x, if it is active.
TAG$B	Sets a breakpoint at address TAG. The first inactive breakpoint is used. If no breakpoint is available a "?" is printed on your terminal.
TAG$xB	Sets a breakpoint at address TAG. Uses breakpoint x whether it was previously in use or not.

DDT will not allow odd address arguments or breakpoint numbers greater than 7 for $B, or for the $C command below.

13.6.10 Clearing Breakpoints ($C)

The Escape-C ($C) command clears one or all of the breakpoints currently set. It accepts two arguments in the same manner as $B.

$C	Clears all active breakpoints from the table.
$xC	Clears breakpoint x, if it was active.
TAG$C	Clears the breakpoint at address TAG, if such a breakpoint exists.
TAG$xC	Functions the same as $xC.

13.6.11 Proceeding From a Breakpoint ($P)

The Escape-P ($P) command proceeds from the last breakpoint. This command is only valid if a breakpoint has been reached in the program. When executed, $P causes program execution to resume until another breakpoint is encountered or the program exits.

The $P command accepts an optional argument before the Escape-P. This argument is a one word value telling DDT how many times to execute the current breakpoint before breaking again. Thus the command 5$P tells DDT to pass through this breakpoint five times before breaking again. If this argument is not given, DDT assumes a value of one. Using this argument is often useful if a breakpoint has been placed within a loop, and you wish to have DDT break only after several iterations of the loop.

13.6.12 Executing Single Instructions ($X and \)

The Escape-X ($X) and backslash (\) commands are identical. Both cause the execution of a single instruction. These commands are valid only after a breakpoint has been reached. They are usually used to monitor the execution of a small section of a program, allowing the examination or modification of registers and memory locations between each instruction. IMPORTANT NOTE: You are not allowed to single-step through a supervisor call (also known as a "monitor call").

13.6.13 Setting Program-Relative Mode ($R)

The Escape-R ($R) command enters program-relative mode once you have been in absolute or register mode.

13.6.14 Displaying Data in Decimal ($D)

The Escape-D ($D) command displays a location or series of locations in decimal. This command accepts one of two possible arguments, but not both. One of the arguments represents the expression to translate and the other is the number of locations to translate. The following table should explain the format:

$D	Displays the currently open location in decimal.
$xD	Displays x words in decimal, starting with the currently open location.
exp$D	Displays the decimal value of exp. Exp can be numeric, symbolic, or an opcode expression. As many words as are needed to display the entire expression are used.

13.6.15 Displaying Data in Octal ($=)

The Escape-equal ($=) command displays a location or a series of locations in octal. It is identical in format to the $D command.

13.6.16 Displaying Data in Hex ($H)

The Escape-H ($H) command displays a location or a series of locations in hexadecimal. It is identical in format to the $D command.

13.6.17 Displaying Data in RAD50 ($*)

The Escape-asterisk ($*) command displays the contents of the current location in unpacked RAD50 format.

13.6.18 Displaying Data as ASCII Characters ($")

The Escape-quote ($") command displays the contents of the current location as two ASCII characters.

13.6.19 Displaying Data as Bytes ($#)

The Escape-pound sign ($#) command displays the contents of the current location as two 8-bit bytes. The low order byte of the word is displayed first. Typeout is in octal.

13.6.20 Displaying a String of ASCII Characters ($A)

The Escape-A ($A) command displays a string of bytes as ASCII characters. This command terminates its typeout when a null byte is found, and adjusts the current location to the next even address following the null byte. The command accepts two formats:

$A	Display ASCII data starting with the current open location.
TAG$A	Display ASCII data starting at relative address TAG.

13.6.21 Displaying the Base Address and Size ($M)

The Escape-M ($M) command displays the absolute base address and the size in bytes of the program being debugged. This is the same information typed when DDT is first started.

13.6.22 Defining New Symbols (:)

The colon (:) command allows you to define new symbols and insert them into the symbol table. The location being given a label must be within the program, not outside of it. Symbols are, as usual, one to six RAD50 characters long, with the first character always alphabetic. A symbol may be defined by merely typing the label name followed immediately by a colon, as in:

```
TAG:
```

The value assigned to the symbol is the location of the last examined address. Once the symbol has been defined, it may be referenced symbolically by you throughout the program. The colon command is most often used during program patching (see Section 13.7, "Using DDT To Patch Programs"). New symbols are automatically inserted into your symbol table. Once you have exited from DDT, you can resave the symbol (.SYM) file so that the newly defined symbols are available next time you use DDT on the program.

13.6.23 Examining Register Contents (%)

The percent (%) command examines the contents of a register without entering register mode. It is often used to display the contents of a register as you single-step through a program, without having to enter and exit register mode. The format for the percent command is "%xx=", where xx is the CPU register that you want to display. The register argument must be in standard register notation (i.e., R1, R2, R3, R4, R5, SP, or PC). The contents of the register are displayed in octal.

13.7 USING DDT TO PATCH PROGRAMS

You will often use DDT to patch an existing program. This is often useful if you do not have the source code handy, or if you do not wish to go through a time-consuming reassembly of your program. DDT provides for patching through the use of the colon command to define symbols, and through automatic expansion of the program area. Patches may be placed at the end of the program after the last valid location in the program; DDT will automatically expand the program to fit the patches. Program patches may be done symbolically through the normal symbolic entry mode, and through the use of the colon command. A symbol may not, however, be referenced before you define it. If a label is defined at the start of the patch, the patch may be referred to symbolically throughout the main program.

13.8 DDT ERRORS

If DDT does not understand your input, it displays a "?".

Other error messages include:

?Cannot OPEN filespec - not found
: where filespec is the file you want to debug. Make sure that you are logged into the proper account and device.

?Cannot single step through SVC

You cannot use the $X command to single-step trhough a supervisor call. You must skip over the call by placing a breakpoint after the call and its arguments; then use the $P command to skip to that location. At that point you can resume single-stepping.

?DDT Internal buserr

A bus error occurred within the DDT program itself. This error was not caused by your program.

?Buserr at monitor PC nnnn

A bus error occurred, but was not caused by DDT. Your program is probably at fault. The number that appears in the message tells you what memory address was loaded into the Program Counter when the error occurred.

13.9 EXITING DDT

To leave DDT, type a Control-C. DDT will save the altered program and symbol table in memory, allowing you to use the SAVE command to make a permanent copy of either the modified program or symbol table. You should never save a program that has been partially run; it is a good idea to use DDT on the program once again, put in the patches, and save it, without running it. This ensures that there are no data storage areas that have been altered from their orginal state. If the program exits on its own while being run, you should NEVER save it if breakpoints were used anywhere in the program. Breakpoints are not cleared until the program goes back to DDT. Running through breakpoints when not under control of DDT can have disasterous results.

APPENDIX A

THE ASCII CHARACTER SET

The next few pages contain charts that list the complete ASCII character set. We provide the octal, decimal and hexadecimal representations of the ASCII values.

Note that the first 32 characters are non-printing Control-characters.

THE CONTROL CHARACTERS

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
NULL	000	0	00	Null (fill character)
SOH	001	1	01	Start of Heading
STX	002	2	02	Start of Text
ETX	003	3	03	End of Text
ECT	004	4	04	End of Transmission
ENQ	005	5	05	Enquiry
ACK	006	6	06	Acknowledge
BEL	007	7	07	Bell code
BS	010	8	08	Back Space
HT	011	9	09	Horizontal Tab
LF	012	10	0A	Line Feed
VT	013	11	0B	Vertical Tab
FF	014	12	0C	Form Feed
CR	015	13	0D	Carriage Return
SO	016	14	0E	Shift Out
SI	017	15	0F	Shift In
DLE	020	16	10	Data Link Escape
DC1	021	17	11	Device Control 1
DC2	022	18	12	Device Control 2
DC3	023	19	13	Device Control 3
DC4	024	20	14	Device Control 4
NAK	025	21	15	Negative Acknowledge
SYN	026	22	16	Synchronous Idle
ETB	027	23	17	End of Transmission Blocks
CAN	030	24	18	Cancel
EM	031	25	19	End of Medium
SS	032	26	1A	Special Sequence
ESC	033	27	1B	Escape
FS	034	28	1C	File Separator
GS	035	29	1D	Group Separator
RS	036	30	1E	Record Separator
US	037	31	1F	Unit Separator

PRINTING CHARACTERS

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
SP	040	32	20	Space
!	041	33	21	Exclamation Mark
"	042	34	22	Quotation Mark
#	043	35	23	Number Sign
$	044	36	24	Dollar Sign
%	045	37	25	Percent Sign
&	046	38	26	Ampersand
'	047	39	27	Apostrophe
(	050	40	28	Opening Parenthesis
)	051	41	29	Closing Parenthesis
*	052	42	2A	Asterisk
+	053	43	2B	Plus
,	054	44	2C	Comma
-	055	45	2D	Hyphen or Minus
.	056	46	2E	Period
/	057	47	2F	Slash
0	060	48	30	Zero
1	061	49	31	One
2	062	50	32	Two
3	063	51	33	Three
4	064	52	34	Four
5	065	53	35	Five
6	066	54	36	Six
7	067	55	37	Seven
8	070	56	38	Eight
9	071	57	39	Nine
:	072	58	3A	Colon
;	073	59	3B	Semicolon
<	074	60	3C	Less Than
=	075	61	3D	Sign
>	076	62	3E	Greater Than
?	077	63	3F	Question Mark
@	100	64	40	Commercial At

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
A	101	65	41	Upper Case Letter
B	102	66	42	Upper Case Letter
C	103	67	43	Upper Case Letter
D	104	68	44	Upper Case Letter
E	105	69	45	Upper Case Letter
F	106	70	46	Upper Case Letter
G	107	71	47	Upper Case Letter
H	110	72	48	Upper Case Letter
I	111	73	49	Upper Case Letter
J	112	74	4A	Upper Case Letter
K	113	75	4B	Upper Case Letter
L	114	76	4C	Upper Case Letter
M	115	77	4D	Upper Case Letter
N	116	78	4E	Upper Case Letter
O	117	79	4F	Upper Case Letter
P	120	80	50	Upper Case Letter
Q	121	81	51	Upper Case Letter
R	122	82	52	Upper Case Letter
S	123	83	53	Upper Case Letter
T	124	84	54	Upper Case Letter
U	125	85	55	Upper Case Letter
V	126	86	56	Upper Case Letter
W	127	87	57	Upper Case Letter
X	130	88	58	Upper Case Letter
Y	131	89	59	Upper Case Letter
Z	132	90	5A	Upper Case Letter
[	133	91	5B	Opening Bracket
\	134	92	5C	Back Slash
]	135	93	5D	Closing Bracket
^	136	94	5E	Circumflex
_	137	95	5F	Underline
`	140	96	60	Grave Accent
a	141	97	61	Lower Case Letter
b	142	98	62	Lower Case Letter
c	143	99	63	Lower Case Letter
d	144	100	64	Lower Case Letter
e	145	101	65	Lower Case Letter
f	146	102	66	Lower Case Letter
g	147	103	67	Lower Case Letter
h	150	104	68	Lower Case Letter
i	151	105	69	Lower Case Letter
j	152	106	6A	Lower Case Letter
k	153	107	6B	Lower Case Letter
l	154	108	6C	Lower Case Letter
m	155	109	6D	Lower Case Letter
n	156	110	6E	Lower Case Letter
o	157	111	6F	Lower Case Letter

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
p	160	112	70	Lower Case Letter
q	161	113	71	Lower Case Letter
r	162	114	72	Lower Case Letter
s	163	115	73	Lower Case Letter
t	164	116	74	Lower Case Letter
u	165	117	75	Lower Case Letter
v	166	118	76	Lower Case Letter
w	167	119	77	Lower Case Letter
x	170	120	78	Lower Case Letter
y	171	121	79	Lower Case Letter
z	172	122	7A	Lower Case Letter
{	173	123	7B	Opening Brace
|	174	124	7C	Vertical Line
}	175	125	7D	Closing Brace
	176	126	7E	Tilde
DEL	177	127	7F	Delete

APPENDIX B

SUMMARY OF PROGRAM SWITCHES

The sections below list the option request switches used by the various components of the Alpha Micro assembly language programming system:

MACRO
LINK
SYMBOL
LIB
GLOBAL

For more information on a particular option request, see the chapter in this book that discusses the appropriate program.

B.1 THE MACRO ASSEMBLER - MACRO

/B text — Generates bottom footer title on each listing page using the rest of the command line following the switch. /B must be the last switch on the command line.

/C — Includes conditionals in the listing.

/E — Writes to the listing only those lines that contain an error.

/H — Lists binary code in hexadecimal instead of octal in the listing.

/L — Generates an assembly listing file. Creates the output file with the same name as your source file, but a .LST extension.

/O — Uses current object file by omitting Phases 1 and 2.

/R — Generates a cross reference, which appears at the end of the assembly listing.

/T Prints the listing on your terminal instead of writing it to a disk file.

/V{a}:X Allows you to specify a value on the MACRO command line which can be examined during the assembly process. "a" specifies the type of value specified, and X is the value.

/X Lists in your assembly listing all macro expansions.

NOTE: You do not have to specify the /L switch when you use the /B, /C, /E, /H, /R, /T, or /X switches to tell MACRO to generate a listing.

You may combine any of the above switches as desired in a single command line by entering them after a single / character at the end of the command line. For example:

```
.MACRO NEWDVR.MAC/RT RET
```

B.2 THE LINKAGE EDITOR - LINK

/E Include equated symbols in the symbol table file. (You must use /E with the /M or /S switch.) (Operation switch.)

/L Designates a library file.

/M Generates a load map (.MAP) file. (Operation switch.)

/N Suppress /P switch. (Operation switch.)

/O Designates an optional file.

/P Generates program (.PRG) and overlay (.OVR) files. The default switch. (Operation switch.)

/R Designates a required file. The default switch. Cancels the /L and /O switches.

/S Generate a symbol table (.SYM) file. (Operation switch.)

You may specify multiple switches if you precede each switch with a slash. For example:

```
.LINK MAIN,SUB1/M/S RET
```

B.3 THE SYMBOL TABLE FILE GENERATOR - SYMBOL

/E Include equated symbols in the symbol table file. You may also use this switch with /M to tell SYMBOL to include equated symbols in the load map. (Operation switch.)

/L Designates a library file.

/M Generate a load map (.MAP) file. (Operation switch.)

/N Suppress /S switch. (Operation switch.)

/O Designates an optional file.

/P Generate program (.PRG) and overlay (.OVR) files. (Operation switch.)

/R Designates a required file. The default switch. Cancels the affect of a /L or /O switch.

/S Generate a symbol table (.SYM) file. The default switch. (Operation switch.)

You may specify multiple switches if you precede each switch with a slash. For example:

```
.SYMBOL MAIN,SUB1/M/S (RET)
```

B.4 THE OBJECT FILE LIBRARY GENERATOR - LIB

The only LIB switch at this time is the /L switch which tells LIB to generate a library listing. This listing looks similar to a load map listing (see Section 10.4., "The Load Map File."), and lists all object files in the library file and all INTERNed symbols.

If you specify an output file (e.g., LIB LIST=MYLIB/L) LIB creates the listing with the name and extension you specified (the default extension is .LST). If you do not specify an output file (e.g., LIB MYLIB/L), LIB sends the library listing to your terminal display.

B.5 THE GLOBAL CROSS REFERENCE GENERATOR - GLOBAL

Line width options (default is 80 characters):

/W Wide listing (same as /W:130). Produces a listing file that may have up to 130 characters on a line.

/W:n Specifies characters per line, where n specifies the number of characters.

Page length options (default is 60 lines):

/L	Long listing (same as /L:80).
/L:n	Specifies lines per page, where n specifies the number of lines.

Each switch must begin with a slash. For example:

```
.GLOBAL/W/L MAIN,SUB1,SUB2 RET
```

Index

)FTWARE PUBLICATIONS FILE REFERENCE NUMBER: _____

SOFTWARE DOCUMENTATION READER'S COMMENTS

e appreciate your help in evaluating our documentation efforts. Please feel free to attach additional comments. If you require a written response, che

NOTE: This form is for comments on software documentation only. To submit reports on software problems, use Software Performance Reports (SPRs), available from Alpha Micro.

ease comment on the usefulness, organization, and clarity of this manual: AMOS Assembly Language Programmer's Manual

_____

_____

_____

_____

_____

_____

_____

id you find errors in this manual? If so, please specify the error and the number of the page on which it occurred.

_____

_____

_____

_____

_____

hat kinds of manuals would you like to see in the future?

_____

_____

_____

_____

ease indicate the type of reader that you represent (check all that apply):

- ☐ Alpha Micro Dealer or OEM
- ☐ Non-programmer, using Alpha Micro computer for:
  - ☐ Business applications
  - ☐ Education applications
  - ☐ Scientific applications
  - ☐ Other (please specify):

    _____
- ☐ Programmer:
  - ☐ Assembly language
  - ☐ Higher-level language
  - ☐ Experienced programmer
  - ☐ Little programming experience
  - ☐ Student
  - ☐ Other (please specify):

    _____

AME: _____ DATE: _____

TLE: _____ PHONE NUMBER: _____

RGANIZATION: _____

DDRESS: _____

TY: _____ STATE: _____ ZIP OR COUNTRY: _____

STAPLE STAPLE

FOLD FOLD

PLACE
STAMP
HERE

alpha
micro

17881 Sky Park North
Irvine, California
92714

ATTN: SOFTWARE DEPARTMENT

FOLD FOLD