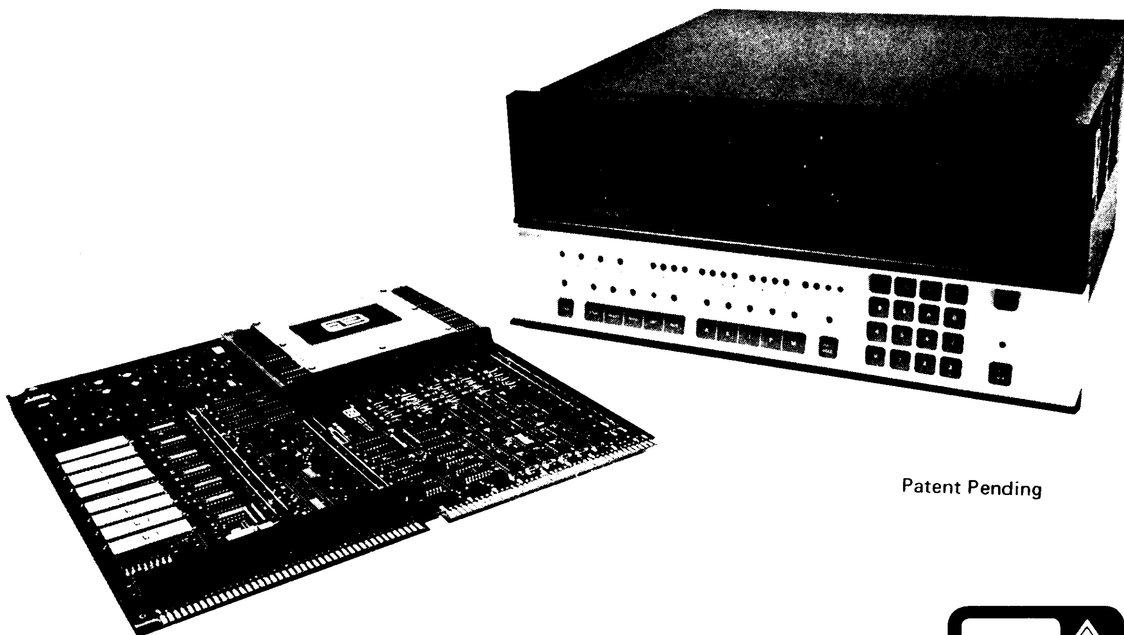
20025-00A0
SEPTEMBER 1973

NAKED MINI® LSI/ALPHA LSI
SOFTWARE MANUAL



Patent Pending

COMPUTER AUTOMATION, INC.
the NAKED MINI company
18651 Von Karman, Irvine, Calif. 92664
tel. 714-833-8830 TWX 910-595-1767

COPYRIGHT 1973, COMPUTER AUTOMATION, INC.

ALPHA LSI SOFTWARE MANUAL

TABLE OF CONTENTS

Page No.

CHAPTER 1. INTRODUCTION 1A-1

Software Problem Report Forms 1A-2

CHAPTER 2. ASSEMBLERS

BETA-4 Assembler 2A-1
BETA-8 Assembler 2B-1
OMEGA Assembler 2C-1

CHAPTER 3. UTILITIES

Bootstrap (BOOT) 3A-1
AUTOLOAD 3AA-1
Binary Loader (BLD) 3B-1
Object Loader (LAMBDA) 3C-1
Binary Dump/Verify Program (BDP/VER) 3D-1
Debug (DBG) 3E-1
Source Tape Preparation (STP) 3F-1
Concordance (CONC) 3G-1

CHAPTER 4. SUBROUTINES

Teletype Utility Package (TUP) 4A-1
Fixed Point Arithmetic Package 4B-1
Floating Point Arithmetic Package 4C-1
Floating Point Elementary Functions 4D-1

CHAPTER 5. I/O DRIVERS

Teletype I/O Drivers 5A-1

PROGRAMMING NOTES

NOTE 1.3 Beta Assembler Core Requirements
NOTE 2 Converting Software From CAI 216 to ALPHA Series Computer
NOTE 4 Recommended Load Locations of Standard Utilities
NOTE 5 Loader Extensions
NOTE 6 Console Register Instructions

APPENDIX A. DIAGNOSTICS

Quality Control Diagnostic (QCD) A1-1

CHAPTER 1. INTRODUCTION

GENERAL

The ALPHA Software Manual describes all the standard software which supports the NAKED MINI/ALPHA Series computers manufactured by Computer Automation, Inc.

ORGANIZATION AND CONTENT OF THE MANUAL

This manual is organized into five chapters, a programming note section and one appendix. The purpose of each chapter of this manual is as follows:

CHAPTER 1.	INTRODUCTION - Describes the purpose and organization of this manual.
CHAPTER 2.	ASSEMBLERS - Describes all assemblers and provides operating procedures. Programs in this category include BETA 4, BETA 8, etc.
CHAPTER 3.	UTILITIES - Describes all utility programs. Provides operating instructions and listings for programs such as bootstrap, object loaders, binary loaders, etc.
CHAPTER 4.	SUBROUTINES - Describes subroutines such as: the multiprecision arithmetic package, trigonometry functions, elementary functions, etc. Includes operating instructions and listings.
CHAPTER 5.	I/O DRIVERS - Describes various I/O driver programs which drive peripheral devices such as line printers, punches, readers, etc.
PROGRAMMING NOTES -	Programming Notes are published periodically whenever a useful programming technique is discovered. These "helpful hints" are made available to all users of an ALPHA Series computer.
APPENDIX A.	DIAGNOSTICS - All diagnostics are assembled in this Appendix. Each diagnostic is fully described in terms of capabilities, loading instructions and operating instructions. The diagnostics described in this Appendix include the Quality Control Diagnostic, I/O diagnostics, etc.

MAINTENANCE OF THIS MANUAL

Users will be notified of new programs and their descriptions. These documents are designed to be conveniently added to this manual at your option.

SOFTWARE PROBLEM REPORT

If you have encountered a problem while using our software, or just have a suggestion for future enhancements, please note them below. Send the completed form to:

Manager, Technical Applications
COMPUTER AUTOMATION, INC.
18651 Von Karman
Irvine, California 92664

Please complete all the blanks. Your sending us as much information as possible will be helpful.

PROGRAM NAME: ______________________________

PROGRAM NUMBER: ______________________________

Computer configuration when problem occurred:

ALPHA-16__________ LSI-1__________ LSI-2__________ OTHER__________

Memory was__________K words of CORE__________MOS__________

Options are:

AUTOLOAD __________ POWER FAIL RESTART__________

REAL TIME CLOCK__________ TELETYPE__________

PERIPHERALS:______________________________

DESCRIPTION OF PROBLEM OR SUGGESTIONS: ______________________________

____

____

____

____

____

YOUR NAME______________________________ TITLE__________

COMPANY NAME ______________________________

ADDRESS______________________________

PHONE ______________________________
(Area Code) (Extension)

Please include any printout, tapes, etc. that might help us to investigate your problem at our facility.

 Form 99-20357-00A0

COMPUTER AUTOMATION, INC.

the NAKED MINI[R] company

18651 Von Karman, Irvine, Calif. 92664
tel. 714-833-8830 TWX 910-595-1767

BETA 4 ASSEMBLER REFERENCE MANUAL

96018-00D0 September, 1973

Copyright 1973, Computer Automation, Inc.

TABLE OF CONTENTS

Section			Page
1.	GENERAL DESCRIPTION		1-1
	1.1 Introduction		1-1
		1.1.1 General	1-1
		1.1.2 Scope	1-2
		1.1.3 Cross Reference Documents	1-2
	1.2 Operating Environment		1-2
		1.2.1 General Software Requirements	1-2
		1.2.2 Hardware Configurations	1-7
	1.3 Design Features		1-7
		1.3.1 General	1-7
		1.3.2 I/O Device	1-7
		1.3.3 Symbolic Source Save	1-9
		1.3.4 Operation Code Definition	1-9
		1.3.5 Free Form Input	1-9
		1.3.6 Formatted Output	1-9
		1.3.7 Error Only Listings	1-9
	1.4 Performance Features		1-9
		1.4.1 General	1-9
		1.4.2 System Resource Usage	1-9
		1.4.3 Automatic Multi-pass Assembly	1-11
	1.5 Definition of Terms		1-12
		1.5.1 Technical Terms	1-12
		1.5.2 Definitions	1-12
3.	SOURCE LANGUAGE DEFINITION		2-1
	2.1 Introduction		2-1
		2.1.1 General	2-1
		2.1.2 Assembler Processing	2-1
	2.2 Types of Source Statements		2-1
		2.2.1 General	2-1
		2.2.2 Comment Statement	2-2
		2.2.3 Pause Statement	2-2
		2.2.4 Eject Statement	2-2
		2.2.5 Instruction Statement	2-2

TABLE OF CONTENTS (Continued)

Section			Page
2.3	Expressions		2-4
	2.3.1	General	2-4
	2.3.2	Definitions	2-5
	2.3.3	Currency Symbol	2-5
	2.3.4	Numeric Terms	2-6
	2.3.5	Symbolic Terms	2-7
	2.3.6	Text String	2-9
	2.3.7	Expression Operators	2-10
	2.3.8	Evaluation of Expressions	2-10
	2.3.9	Absolute and Relocatable Expressions	2-11
	2.3.10	External Reference Expressions	2-15
	2.3.11	Literals	2-15
	2.3.12	Indirect Addressing	2-16
	2.3.13	Indexing	2-16
3.1	Introduction		3-1
	3.1.1	General	3-1
	3.1.2	Instruction Classes	3-1
	3.1.3	Syntax Description	3-1
3.2	Class 1 - Memory Reference, Word Operand		3-1
	3.2.1	General	3-1
	3.2.2	Assembly Format	3-2
	3.2.3	Memory Addressing Modes	3-3
3.3	Class 8 - Memory Reference, Byte Operand		3-9
	3.3.1	General	3-9
	3.3.2	Assembly Format	3-9
	3.3.3	Memory Addressing Modes	3-9
3.4	Class 9 - Memory Reference, Double Word		3-16
	3.4.1	General	3-16
	3.4.2	Assembly Format	3-16
3.5	Class 2 - Immediate Instructions		3-18
	3.5.1	General	3-19
	3.5.2	Assembly Format	3-19
	3.5.3	Class 2 Examples	3-20

TABLE OF CONTENTS (Continued)

Section			Page
	3.6	Class 3 - Conditional Jump Instructions	3-20
	3.6.1	General	3-20
	3.6.2	Assembly Format	3-21
	3.6.3	Defined Jump Conditions	3-22
	3.6.4	Microcoded Jump Conditions	3-23
	3.7	Class 4 - Shift, Single Register	3-24
	3.7.1	General	3-24
	3.7.2	Assembly Format	3-24
	3.7.3	Single Register Shift Instructions	3-25
	3.7.4	Bit Manipulation Intructions	3-26
	3.8	Class 7 - Long Shift	3-27
	3.8.1	General	3-27
	3.8.2	Assembly Format	3-27
	3.9	Class 5 - Register Change and Control	3-28
	3.9.1	General	3-28
	3.9.2	Assembly Format	3-29
	3.10	Class 6 - Input/Output	3-31
	3.10.1	General	3-31
	3.10.2	Assembly Format	3-31
	3.10.3	Multi-word Instructions	3-33
4.	4.1	Introduction	4-1
	4.2	Assembler Control Directives	4-1
	4.2.1	Absolute Assembly (ABS)	4-1
	4.2.2	Relocatable Assembly (REL)	4-2
	4.2.3	Origin (ORG)	4-2
	4.2.4	End of Assembly (END)	4-2
	4.2.5	Conditional Assembly (IFT, IFF, ENDC)	4-3
	4.2.6	Machine Direction (MACH)	4-4
	4.2.7	Title Directive (TITL)	4-5
	4.3	Data and Symbol Definition Directives	4-5
	4.3.1	Reserve Storage (RES)	4-5
	4.3.2	Equate (EQU)	4-6
	4.3.3	Set (SET)	4-7
	4.3.4	Test String (TEXT)	4-7
	4.3.5	Data Definition (DATA)	4-8
	4.3.6	Byte Address Constant (BAC)	4-9

TABLE OF CONTENTS (Continued)

Section			Page
	4.4	Program Linkage Directives	4-9
		4.4.1 External Name Definition (NAM)	4-9
		4.4.2 External Reference (EXTR)	4-10
		4.4.3 External Reference (REF)	4-11
	4.5	Pseudo Instructions	4-11
		4.5.1 Subroutine Call (CALL)	4-11
		4.5.2 Subroutine Entry (ENT)	4-12
		4.5.3 Subroutine Return (RTN)	4-12
		4.5.4 Coded Halt (STOP)	4-13
		4.5.5 Wait for Interrupt (WAIT)	4-13
5.		User-Defined Operation Codes	5-1
	5.1	Introduction	5-1
	5.2	Format	5-1
		5.2.1 Label Field	5-1
		5.2.2 Operation Code Field	5-1
		5.2.3 Expression or Operand Field	5-1
		5.2.4 Comments Field	5-1
	5.3	Restrictions	5-2
	5.4	Instruction Type Code Table	5-2
	5.5	Usage	5-2
		5.5.1 Example 1	5-2
		5.5.2 Example 2	5-3
6.		Assembler Outputs	6-1
	6.1	Introduction	6-1
	6.2	Source Input	6-1
		6.2.1 Source Program	6-1
		6.2.2 Source Tape	6-1
	6.3	Assembly Listing	6-1
		6.3.1 General	6-1
		6.3.2 Listing Format	6-1
		6.3.3 Source Program Errors	6-5

TABLE OF CONTENTS (Continued)

Section		Page
6.4	Object Output	6-6
7.	Operating Procedures	7-1
7.1	Introduction	7-1
7.1.1	General	7-1
7.1.2	Assumptions	7-1
7.2	Step-by-Step Procedures	7-1
7.3	Alteration of Assembler Variables	7-3
Appendix A:	INSTRUCTION MNEMONICS	A-1
Appendix B:	DIRECTIVE MNEMONICS	B-1

LIST OF ILLUSTRATIONS

Figure		Page
1-1	BETA Assembler Translation	1-3
1-2	Software Configuration	1-3
1-3	Source Statement Input: First Pass	1-5
1-4	Source Statement Input: Second Pass	1-6
1-5	Minimum Hardware Configuration	1-8
1-6	Optional Hardware Configuration	1-8
1-7	Memory Allocation	1-10
3-1	Class 1 Machine Language Format	3-2
3-2	Byte Address	3-10
3-3	Class 8 Machine Language Format	3-10
3-4	Class 9 Machine Language Format	3-16
3-5	Class 2 Machine Language Format	3-21
3-6	Class 3 Machine Language Format	3-23
3-7	Class 4 Machine Language Format	3-26
3-8	Class 6 Machine Language Format	3-33
6-1	Source Code	6-2
6-2	Assembly Listing	6-3

LIST OF TABLES

Table		Page
7-1	Assembly Variables	7-3

Section 1

GENERAL DESCRIPTION

1.1 INTRODUCTION

1.1.1 General

The Beta Assember translates symbolic language programs into easily understood object language which may be loaded into an ALPHA series computer (ALPHA LSI or ALPHA 16).

1.1.1.1 Symbolic Source Code

The symbolic source code which is the input to the Beta Assembler consists of individual symbolic statements. All of the statements together make up a program which is to be translated.

Each individual statement represents either a computer instruction or an assembler directive. Computer instructions are translated into an object code for eventual execution by an ALPHA series computer. Directives, however, are instructions to the Beta Assembler itself. They give the assembler directions concerning the translation process.

1.1.1.2 Object Code

The primary output from the Beta Assembler is a program which can be loaded into an ALPHA series computer for execution. The program output is in a language which is called "object code". The physical medium normally used for outputting object code is punched paper tape.

Object code is loaded into an ALPHA computer by the Object Loader (LAMBDA).

The primary function of the Beta Assembler, therefore, is the translation of symbolic source code into object code.

1.1.1.3 Assembly Listing

In addition to an object code tape, the assembler generates a hard copy listing of the assembled program.

1.1.2 Scope

This manual describes the Beta Assembler. Input formats for all classes of ALPHA LSI and ALPHA 16 instructions are explained, but the functions of each instruction are not covered in this manual.

1.1.3 Cross Reference Documents

Information which is not explained in this manual is contained in related documents.

1.1.3.1 NAKED MINI LSI/ALPHA LSI Programming Reference Manual

The usage and functions of all instructions recognized by the ALPHA LSI computers are contained in this document.

1.1.3.2 NAKED MINI 16/ALPHA 16 Computer Reference Manual

The functions of all instructions recognized by the ALPHA 16 computers are contained in the NAKED MINI 16/ALPHA 16 Computer Reference Manual.

1.1.3.3 ALPHA Object Loader (LAMBDA) Description

The Object Loader is described in a separate document. Refer to that document for a description of object tape load procedures, the functions of the Object Loader, and a detailed description of the object tape format.

1.2 OPERATING ENVIRONMENT

1.2.1 General Software Requirements

The Beta Assembler is an I/O independent program which translates symbolic source statements into object code and listing information. It requires I/O driver modules which will perform all necessary I/O functions. The drivers required are:

a. Source Input Module
b. Object Output Module
c. Listing Output Module
d. Source Save/Retrieve Module

Figure 1-2 illustrates the required software configuration.

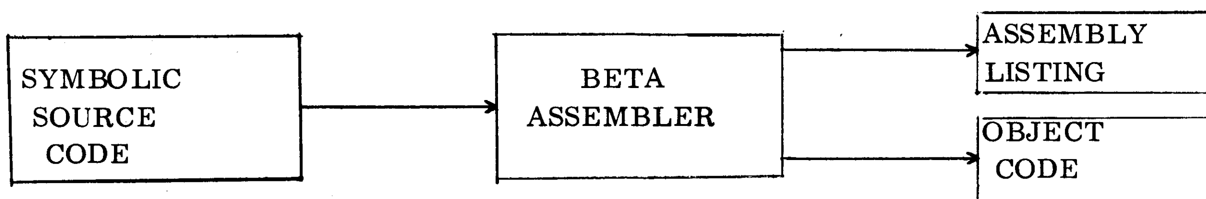


Figure 1-1. BETA Assembler Translation

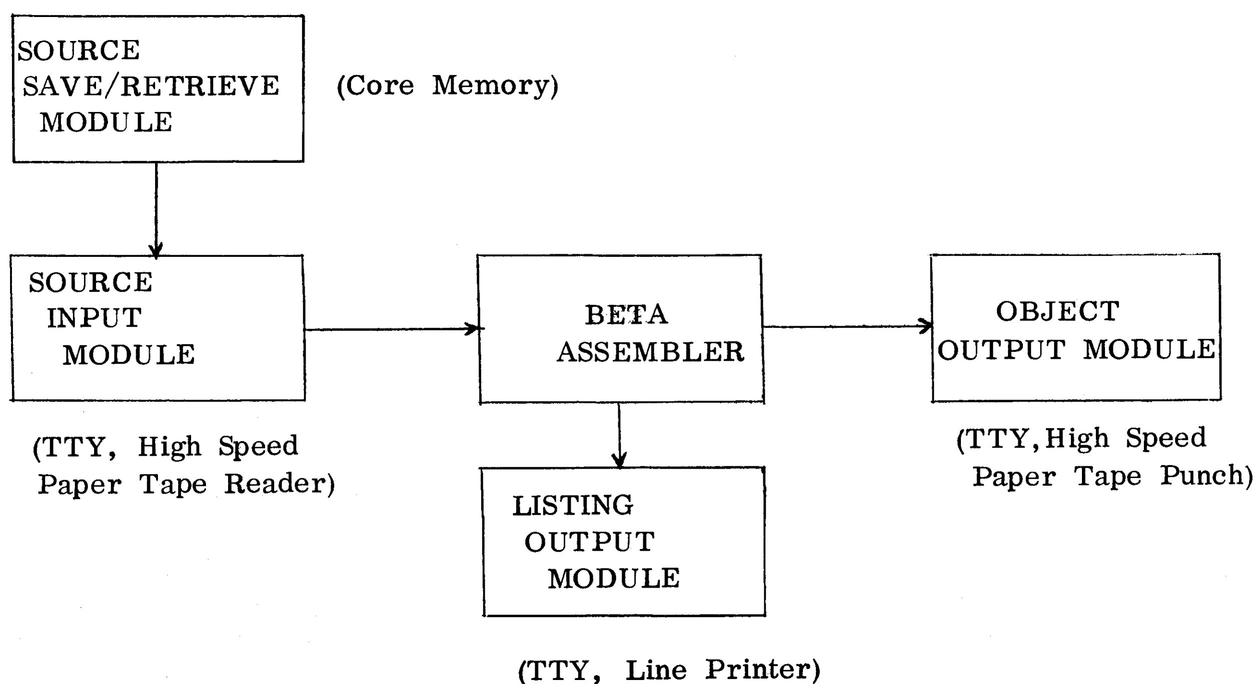


Figure 1-2. Software Configuration

1.2.1.1 Source Input Module

The source input module provides an interface between the assembler and the source input device. The assembler calls this module whenever it is ready to process a new source statement.

In the standard configuration, this module drives a TTY paper tape reader or a high-speed paper tape reader.

1.2.1.2 Object Output Module

The object output module collects and outputs the object code to the selected I/O device.

In the standard configuration this module drives either a TTY paper tape punch or a high speed paper tape punch.

1.2.1.3 Listing Output Module

The assembler calls the listing output module when listing information is ready for printing on a hard-copy device.

In the standard configuration this module drives either a TTY printer or a line printer.

1.2.1.4 Source Save/Retrieve Module

Beta is a two-pass assembler; i.e. the source code must be read twice by Beta in order for the complete translation process to be performed.

When the source code is read the first time, the source code is saved in memory for the second pass. The source input module passes each source statement to the source save/retrieve module when the statements are read during the first assembly pass. Figure 1-3 illustrates a source statement input during the first pass.

When Beta requires a source input statement during the second assembly pass, the source input module checks to see if source statements were saved during the first pass. If they were saved, the source input module calls the source save/retrieve module for the next source statement. If they were not saved, the source input module goes to the source input I/O device for the next source statement. Figure 1-4 illustrates a source statement input during the second assembly pass.

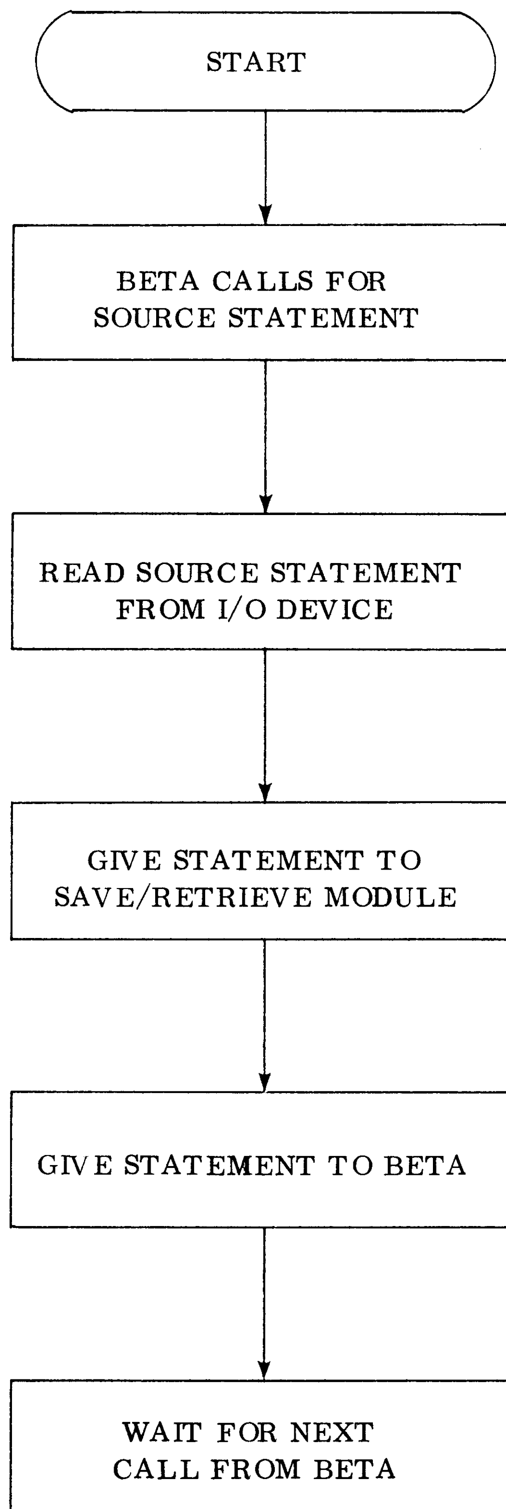


Figure 1-3. Source Statement Input: First Pass

START

BETA CALLS
FOR SOURCE STATEMENT

WAS SOURCE SAVED
DURING FIRST PASS?

YES

NO

GET SOURCE STATEMENT
FROM SAVE/RETRIEVE
MODULE

GET SOURCE STATEMENT
FROM SOURCE INPUT
DEVICE

GIVE SOURCE STATEMENT
TO BETA

WAIT FOR NEXT CALL
FROM BETA

Figure 1-4. Source Statement Input: Second Pass

In the standard system, source statements are saved in memory. The source save/retrieve module will save source statements as long as there is memory space available (no conflict with symbol table memory requirements). If the source save/retrieve module runs out of memory, the source program will be re-read from the source input I/O device.

1.2.2 Hardware Configurations

The following paragraphs describe the hardware configurations which may be used with a 4K ALPHA LSI or ALPHA 16 during assembly.

1.2.2.1 Minimum Configuration

Figure 1-5 illustrates the minimum hardware configuration which may be used to assemble programs. The minimum configuration consists of an ALPHA series computer with 4K words of memory and an ASR-33 or ASR-35 Teletypewriter (TTY). The I/O functions provided by the TTY are:

a. Source code input is provided by the paper tape reader.
b. Object code output is provided by the paper tape punch.
c. Assembly listing hard-copy is provided by the printer.

1.2.2.2 Optional Configurations

Figure 1-6 illustrates the additional peripheral devices which may be supported in a basic 4K system:

a. A High Speed Paper Tape Reader may be used for source code input.
b. Object code may be output on a High Speed Paper Tape Punch.
c. Assembly listings may be generated on a Line Printer.

1.3 DESIGN FEATURES

1.3.1 General

The following paragraphs describe some of the design features of the Beta Assembler. These features increase the power and flexibility of the assembler.

1.3.2 I/O Device Independent

The Beta Assembler is not restricted to any particular I/O device configuration. It will operate with any I/O devices for which driver modules are provided.

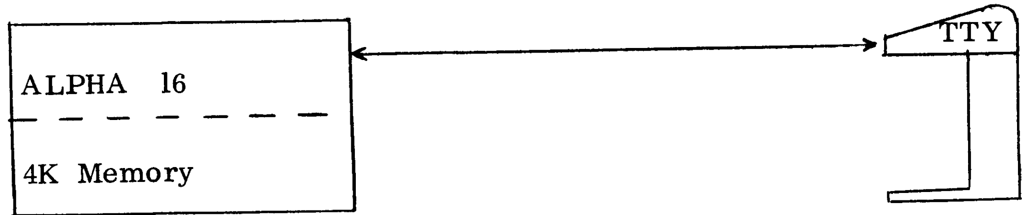


Figure 1-5. Minimum Hardware Configuration

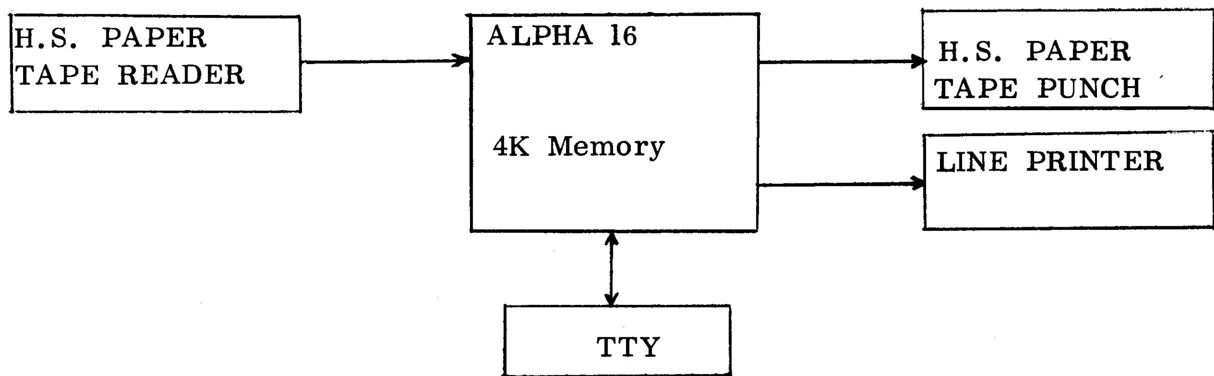


Figure 1-6. Optional Hardware Configuration

1.3.3 Symbolic Source Save

The Beta Assembler system will save source input code during Pass 1 and re-read the source from memory during Pass 2 whenever sufficient memory is available.

1.3.4 Operation Code Definition

The operation code definition feature allows the programmer to re-name existing instructions, or to name previously unnamed instructions.

1.3.5 Free Form Input

Since many users use paper tape as their input medium, requiring a fixed number of character positions for a field makes source preparation extremely time consuming. The Beta Assembler therefore recognizes a space as a separator between fields; i.e., one or more spaces define the end of a field. Source code tapes need not be rigidly formatted.

1.3.6 Formatted Output

The Beta Assembler generates formatted assembly listings even though the source code is not formatted.

1.3.7 Error Only Listings

The programmer has the option of selecting an error only listing at assembly time. Only lines containing an error will be listed.

1.4 PERFORMANCE FEATURES

1.4.1 General

The following paragraphs describe the general methods used by the Beta Assembler in its translation function.

1.4.2 System Resource Usage

In a basic configuration, the computer memory must hold the following when doing an assembly: (See Figure 1-7)

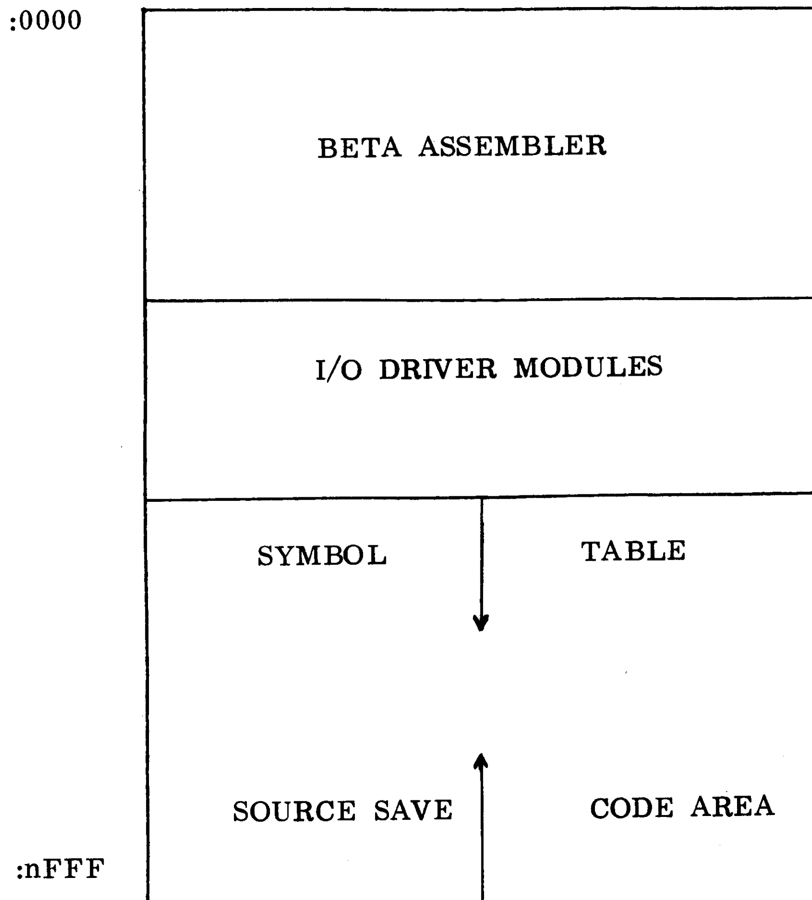


Figure 1-7. Memory Allocation

a. The Beta Assembler and its associated I/O modules.
b. A Symbol Table, which holds the names and assigned values of all symbols used by the programmer.
c. Source symbolic code for re-reading during Pass 2.

1.4.2.1 Allocation of Space

The Beta Assembler will occupy a fixed area of memory. The memory space that remains is used to store the symbol table and source code. The symbol table starts at the first available location following the I/O drivers and works forward toward the end of memory. Four words of memory are required for each symbol.

Source code is saved beginning with the last available word of memory and working backward toward the symbol table. Each source statement requires one word for each two characters, including internal blanks, in the statement. The maximum size of a source statement is seventy-two characters, therefore the maximum number of words that may be required to save a single statement is thirty-six.

1.4.2.2 Space Conflicts

If there is not enough space in memory to save both the symbol table and source code, the symbol table takes precedence. Source code which was saved before memory was exhausted is erased as each new symbol is stored in the symbol table. The source will then have to be re-read from the I/O device selected during Pass 2.

1.4.2.3 Computation of Symbol Table Capacity

The number of symbols which may be stored in the symbol table may be determined as follows:

Number of symbols = [(Upper limit of memory) - (last location occupied by BETA)] /4

1.4.3 Automatic Multi-pass Assembly

The Beta Assembler must read source code twice in order to completely translate source statements into object code. The first pass generates the symbol table, and the second pass completes the assembly.

If source code is saved, Beta executes the second pass automatically without operator intervention.

1.5 DEFINITION OF TERMS

1.5.1 Technical Terms

Certain technical terms used in this manual are defined in the following subsection.

1.5.2 Definitions

1.5.2.1 Directive

A Directive is a command to the Beta Assembler. The term Directive is synonymous with the term pseudo-operation.

1.5.2.2 Source Program

A Source Program is a sequential list of symbolic source statements as they appear on the programmer's coding form.

1.5.2.3 Source Code

Source Code is a source program which has been prepared for input to the Beta Assembler; e.g., when a source program has been punched into paper tape, the paper tape contains the source code. The terms Source Code and Source Program are sometimes used interchangeably.

1.5.2.4 Source Input

Synonymous with Source Code.

1.5.2.5 Object Code

Object Code is one of the outputs from the Beta Assembler. It is the code which can be loaded into the memory of an ALPHA series computer by the Object Loader.

1.5.2.6 Object Tape

An object Tape is a punched paper tape containing the object code output of the Beta Assembler. The Beta Assembler generates an object tape during its final assembly pass.

1.5.2.7 Object Loader

The Object Loader is the loader program which interprets object tapes and loads the assembled program into an ALPHA series computer memory. Refer to the ALPHA Object Loader (LAMBDA) Description for further details.

1.5.2.8 Relocatable Program

A Relocatable Program is one which can be loaded anywhere in memory for execution; i.e. memory addresses are not fixed. As a general rule, object language programs are relocatable.

Section 2

SOURCE LANGUAGE DEFINITION

2.1 INTRODUCTION

2.1.1 General

This section describes the source language format which is recognized by the Beta Assembler. The types of statements are defined, and the fields that make up each type of statement are discussed in detail.

2.1.2 Assembler Processing

The primary function of the Beta Assembler is the translation of symbolic source language statements into object code to be loaded into an ALPHA series computer.

Since the assembler is I/O independent, source statements may be input to the assembler from many different types of devices. The following description is not concerned with input devices, but rather with the form in which source statements must appear when presented to the assembler for processing.

2.2 TYPES OF SOURCE STATEMENTS

2.2.1 General

Four types of source statements are recognized by the assembler: Comment, Pause, Eject, and Instruction.

2.2.1.1 Identification of Type

The type of source statement is specified by the character that appears in the first character position of the statement.

2.2.1.2 Maximum Size

A source statement may contain a maximum of 72 characters, including spaces. It may contain less characters, but must contain at least one character.

2.2.2 Comment Statement

A Comment statement is identified by an asterisk (*) as the first character of the statement. The remainder of the statement is used for programmer comments which will appear on the assembly listing. Comment statements do not cause any object code to be generated.

2.2.3 Pause Statement

A Pause statement is identified by an up arrow (↑) as the first character of the statement. When a Pause statement is recognized, the remainder of the statement is ignored by the assembler.

The Pause statement is meaningful only when punched paper tape is used as the source input medium. The Pause statement causes the assembly process to be temporarily suspended so that the operator may mount another source input tape. This allows the source program to be punched into several small tapes rather than one large tape.

2.2.4 Eject Statement

An Eject statement is identified by a period (.) as the first character of the statement. The Eject statement causes the listing output device to go to the "top of form".

The physical action that is performed by the I/O device depends on how that device interprets a "top of form" command. If the listing device is a line printer, paper will be ejected to the top of the next physical page. If the listing device is a Teletype printer, sufficient line feeds are issued to generate an 11-inch page (66 lines).

2.2.5 Instruction Statement

An Instruction statement is identified by an alphabetic character or a space (letters A-Z or space) as the first character of the statement.

An Instruction statement represents either an instruction to be translated into object code, or a directive to the assembler itself. An Instruction statement contains four fields: Label field, Operation field, Operand field, and Comment field. Adjacent fields are separated by one or more spaces, and the statement is terminated by a carriage return.

2.2.5.1 Label Field

The Label field may contain a symbolic label, or tag, which may be used as a symbolic reference by other instruction statements. The label field is identified by an alphabetic character (letter A-Z) in the first character position of the instruction statement. A space in the first character position means that the statement has no label.

If a label is used, it consists of a symbolic code containing from one to six alphanumeric and colon (:) characters. (A colon is used in all labels in Computer Automation library programs to avoid mnemonic conflicts with user programs. Users should not use colons in labels.) The first character of the label must be alphabetic. For example:

AB15

is a legitimate symbolic label. But

1AB5

is not legitimate because the first character is not alphabetic. Also,

ABC*

is not a legitimate label because it contains a non-alphanumeric or colon character.

At assembly time the label is assigned a value which is determined by the location of the instruction statement with which it is associated. The label may then be used in the operand field of other instruction statements. For example, the statement

```
TEMP    DATA    100
```

causes the binary equivalent of the decimal number 100 to be stored somewhere in the computer memory when the program is assembled and loaded for execution. The constant 100 may be accessed by referring to the label TEMP. The statement

```
        LDA     TEMP
```

when executed as an instruction causes the A Register to be loaded with the value stored in the location identified by the label TEMP. The symbolic terms in the label field are the terms that are stored, along with assigned values, in the symbol table during pass one of the assembly process.

One or more spaces (blanks) terminate the label field.

2.2.5.2 Operation Field

The Operation field contains a legally defined symbolic instruction operation code or directive code. In addition, operation codes defined by the programmer using the Operation Code Definition feature will be recognized.

Directives are defined in Section 4 of this manual. The Operation field consists of not less than one or more than four characters. The Operation field of an instruction statement must be present.

2.2.5.3 Operand Field

The Operand field follows the Operation field and is separated from it by at least one space. It is terminated by a space.

The syntax of the Operand field depends on the type of instruction or directive with which it is associated. The Operand field syntax description is contained in the descriptions of the instruction classes and the directive descriptions. If the Operand field is used, it will contain an expression consisting of one of the following:

a. The currency symbol ($).
b. A single symbolic term.
c. A single numeric term.
d. A combination of symbolic terms, numeric terms, and/or the currency symbol joined by the arithmetic operators plus (+) and minus (-).
e. A text string.

The formation and evaluation of Operand field expressions is treated separately in subsection 2.3.

2.2.6.4 Comments Field

The Comments field follows the operand field and is separated from it by one or more spaces. The Comments field may contain programmer's notes.

Comments are listed on the assembly listing but do not cause the generation of object code.

2.3 EXPRESSIONS

2.3.1 General

The programmer has considerable latitude in writing the expression which appears in the operand field of instruction statements.

2.3.1.1 Syntax

The specific syntax of the operand field is defined by the class of instruction in which it appears. For example, the operand field of an immediate class instruction is limited to eight binary bits; therefore, any expression which generates a value that requires more than eight bits is invalid even though the expression is valid in all other respects.

2.3.1.2 Formation Rules

This subsection describes the general rules for writing expressions. Specific rules for Operand fields which differ from these general rules are explained in the instruction class descriptions to which they apply.

2.3.2 Definitions

2.3.2.1 Term

A term is a single number or symbol. It may have an absolute value, or its value may depend on its location within a program. Some examples of terms are:

25
ABC

2.3.2.2 Expressions

An expression is a grouping of one or more terms connected by the arithmetic operators plus (+) and minus (-). An expression may also contain a text string consisting of one or more ASCII characters. Examples of expressions are:

A	(single symbolic term)
$-4	(symbolic and numeric terms)
ABC-DEF	(symbolic terms connected by arithmetic operator)
'ABCD'	(text string)

2.3.3 Currency Symbol

The Currency symbol ($) is one term which may be used within an expression. It stands for the current value of the location counter; i.e., the location of the instruction or directive in which it is used.

2.3.3.1 Function

The Currency symbol allows the programmer to reference memory locations relative to the instructions which he is writing rather than assigning labels to the referenced locations. For example, without the Currency symbol:

```
LOOP      SEN       7,1
          JMP       LOOP
```

With the currency symbol this code is

```
          SEN       7,1
          JMP       $-1
```

Using the currency symbol eliminates the need for a label, thus saving space in the symbol table.

2.3.3.2 Value

The value assigned to the currency symbol by the assembler is the current value of the location counter at the time the symbol is encountered. This value is absolute if an absolute assembly is being performed, and it is relative if a relocatable assembly is being performed.

2.3.4 Numeric Terms

Numeric terms are used when absolute numeric values are required within an expression. Numeric terms may consist of decimal, octal, or hexadecimal numbers. Numeric terms are not affected by the type of assembly being performed. Their values are always absolute, even if a relocatable assembly is being performed.

2.3.4.1 Decimal Numbers

A decimal number contains from one to five decimal digits. The range for decimal numbers is 0 thru 32767. The first digit of the number must be non-zero.

A decimal number is interpreted as a signed positive integer when interpreted by the assembler. It may be prefixed with a minus sign (-) in order to generate a negative number. Examples of decimal numbers are:

```
5
296
7594
```

2.3.4.2 Octal Numbers

An octal number contains from one to seven octal digits. The first digit must be a zero. The leading zero is an indicator which identifies an octal number. The range for octal numbers is from 0 thru 0177777.

An octal number is generally used to generate a bit pattern rather than a signed numeric value. It may be prefixed with a minus (-) sign in order to generate a negative (two-complement) value. Examples of octal numbers are:

016
0370

2.3.4.3 Hexadecimal Numbers

A hexadecimal number contains from one to four hexadecimal digits preceeded by a colon (:). The range for hexadecimal numbers is from :0 thru :FFFF.

A hexadecimal number is generally used to generate a bit pattern or reference a specific memory location. It may be prefixed with a minus sign (-) in order to generate a negative (two-complement) value.

:A
:B5F4
:123

2.3.5 Symbolic Terms

Symbolic terms are symbols made up of alphanumeric and colon (:) characters. A symbolic term may contain from one to six characters. The first character must be alphabetic.

2.3.5.1 Function

A symbolic term is used to reference a location within the current program, or an entry point to an external program. For example, a memory reference instruction may use a symbolic term to specify the location of the data which is being referenced:

```
LDA     ABC
```

A Jump instruction may use a symbolic term to specify the branch location:

```
JMP     LOOP
```

2.3.5.2 Symbol Definition

A symbol which is referenced in the operand field of an instruction must be defined in some manner if the assembler is to properly assemble the instruction. A symbol may be defined in one of two ways:

a. By appearing in the label field of the current program.
b. By appearing in the operand field of an EXTR directive.

The second method is a special case associated with references to external programs and will be considered separately. This discussion applies to the first case only.

2.3.5.3 Value

A symbolic term is assigned the value of the location counter when the value is defined. The first assembly pass reads the label field of the program and assigns values to all symbolic terms which appear there. These terms, with their assigned values, are then stored in the symbol table for future reference.

The following example illustrates symbol definition and value assignment:

```
Line  1.              REL      :100
      2.    START     LDA      ABC
      3.              ADD      DEF
      4.              STA      GHI
      5.    ABC       DATA     250
      6.    DEF       DATA     500
      7.    GHI       DATA     0
      8.              END
```

The statement in line 1 is a directive which sets the starting value of the location counter to hexadecimal 100 and the relocation attribute to relative at the start of the assembly.

The statement at line 2 is the first instruction to be stored in memory. The symbol START is given the current value of the location counter, or :100, and the location counter is incremented to :101.

The instruction statements in lines 3 and 4 do not have symbols in their label fields, but they do cause the location counter to be incremented. The location counter has a value of :103 when the instruction statement in line 5 is read, therefore, the symbol ABC is given the value :103. Values are assigned to symbols DEF and GHI in the same manner.

If this program were assembled, the symbol table would show that:

LABEL	VALUE
START	:100
ABC	:103
DEF	:104
GHI	:105

When these symbols are encountered in the operand fields of instructions during the second assembly pass, the values in the symbol table are then substituted for the symbols for final evaluation of the operand field expression.

2.3.5.4 Absolute and Relocatable Assemblies

If an absolute assembly is performed, the value assigned to each symbol is fixed. If a relocatable assembly is performed, the value may be modified by a relocation bias when the program is loaded by the object loader.

2.3.6 Text String

A text string consists of one or more ASCII characters bounded by quote ('). The number of ASCII characters which may appear in the operand field of an instruction statement depends on the class of instruction which is being assembled.

2.3.6.1 Character Size

An ASCII character is represented by an eight-bit binary pattern, therefore one character can be stored in one byte location, and two characters can be stored in one computer word.

By this definition, an immediate class instruction can legitimately hold only one ASCII character in its D field:

```
          CAI          'B'
```

A text string using a TEXT directive can contain an unlimited number of characters:

```
          TEXT         'ABCDEFGHIJ'
```

Remember, however, that a source statement is limited to a maximum of 72 characters.

2.3.6.2 Recognized Characters

All printable ASCII characters, including the ASCII space, may be used in a text string. Special characters, such as carriage return and line feed, may not be used.

2.3.6.3 Quote Sign

A quote sign (') is used to mark the beginning and end of a text string.

```
'ABCDEF'
```

If the quote sign itself is to be used as a printable character within the text string, a double quote is used:

```
'Don"t'
```

2.3.7 Expression Operators

Terms with an expression are connected by the arithmetic operators plus (+) for addition and minus (-) for substraction. Any valid terms may be connected by + and -.

Examples of terms connected by arithmetic operators are:

```
$-4
ABC+5
4+5
ABC+DEF-GHI
```

2.3.8 Evaluation of Expressions

The expression in the operand field of an instruction must be reduced to a single value. This subsection explains the rules by which expressions are evaluated.

2.3.8.1 Single Term Expressions

An expression consisting of a single term has the value of that term. For example, consider the symbol values listed in paragraph 2.3.5.3. The symbol ABC has the value :103. The instruction

```
          LDA          ABC
```

then means "load the A Register with the contents of location :103". The value of the symbol is submitted for the symbol in evaluating the expression.

Text strings are considered to be single term expressions.

2.3.8.2 Multiple Term Expressions

Expressions which consist of two or more terms connected by operators are reduced to a single value. Evaluation proceeds from left to right, and groupings of terms using parentheses or brackets is not permitted. For example, using the values given in paragraph 2.3.4.3, the following term will be evaluated:

```
LDA        DEF-ABC+2
```

The first step in evaluating the expression is to assign values to the symbolic terms:

```
LDA        :104-:103+2
```

The first two terms are evaluated:

```
:104 - :103=1
```

The value is substituted for the first two terms:

```
LDA        1 + 2
```

The third term is combined with the computed value:

```
1 + 2 = 3
```

This is now a single value, so the final statement is:

```
LDA        3
```

2.3.9 Absolute and Relocatable Expressions

An expression is absolute if its computed value is unaffected by program relocation when the assembled object code is loaded by the Object Loader. An expression is relocatable if its value changes according to the location in which the program is loaded.

2.3.9.1 Relocation Bias

A relocatable program is relocated by the Object Loader by displacing each instruction in the program to a new location.

At load time each relocatable instruction is offset by an amount called the Relocation Bias. The load address is computed as:

Load Address = Relocation Bias + Origin Address.

The Origin Address is the value to which the assembler location counter was set by an REL directive at assembly time. The Load Address is the address into which the program is loaded by the Object Loader at load time.

2.3.9.2 Relocatable Expressions

Symbolic terms are assigned values according to their location relative to the origin address of the program at assembly time. But the actual value of a symbolic term changes if the program is relocated.

The following example illustrates the values which are assigned to a program at assembly time:

```
Location
Counter

                        REL      :100
:100        START       LDA      ABC
:101                    STA      DEF
:102        ABC         DATA     ABC
:103        DEF         DATA     0

            START = :100
            ABC   = :102
            DEF   = :103
```

The location counter is set to a starting value by REL directive, and each symbolic term that appears in the label field is assigned a value according to its location relative to the origin address.

The following example illustrates the application of a relocation bias to the origin address:

Relocation bias = :500

Load Address	Content	Source Code		
			REL	:100
:600	LDA :602	START	LDA	ABC
:601	STA :603		STA	DEF
:602	:602	ABC	DATA	ABC
:603	0	DEF	DATA	0

Each instruction location, and thus the value of each symbolic term, is offset by the amount of relocation bias.

Each expression in the operand field whose value is changed by the changes in symbolic term values is a relocatable expression. For example, in the first example:

```
STA        DEF
```

The operand field is relocatable because the value of DEF changes from :103 to :603 because of program relocation.

In the first example, if the statement:

```
LDA        ABC
```

were changed to:

```
LDA        $+2
```

the expression would still be relocatable. The currency symbol would change in value from :101 to :601 because of relocation.

2.3.9.3 Absolute Expressions

An absolute expression is one whose value does not change because of relocation. For example, a numeric term is not subject to a relocation bias:

```
LDA        :F6
```

Also, if two symbolic terms are connected by a minus operator, the relocation biases cancel:

```
LDA        DEF-ABC
```

If the origin addresses in the above example are applied, the evaluation of the expression is:

```
:103 - :102 = 1
```

which is equivalent to:

: 603 - : 602 = 1

2.3.9.4 Rule for Relocatability

The relocatability of an expression may be determined as follows:

R = (Number of added relocatable terms) - (Number of subtracted relocatable terms)

a. If R = 1, the term is relocatable.
b. If R = 0, the term is absolute.
c. If R $\neq$ 1 or 0, then the expression cannot be handled by the assembler or the Object Loader.

A relocation bias is applied to the expression by the Object Loader at Load time if R = 1. The bias is not applied if R = 0. The Object Loader can apply one bias or no bias. For example, consider the expression:

$ + ABC - DEF

There are two added relocatable terms and one subtracted term:

R = 2-1 = 1

A relocation bias would be applied to the expression at load time.

Consider the following relative expression:

ABC + DEF

There are two added relocatable terms and no subtracted relocatable terms.

R = 2-0 = 2

The expression is illegal, since R $\neq$ 1 and R $\neq$ 0.

Relocatable terms may be combined with absolute terms:

$ + 5

There is one added relocatable term and no subtracted relocatable terms, therefore R = 1. The term +5 is absolute and is not a factor in computing relocatability.

2.3.10 External Reference Expressions

An external reference is a reference to a symbolic term that is defined in an external program. The term is declared external to the current program by an EXTR directive or a REF directive.

An external reference expression must be a single term expression. The symbolic term which is defined as being external to the current program cannot be combined with other terms in an expression. Indirect reference to external variables declared in an EXTR directive is not allowed.

2.3.11 Literals

A literal may be used with memory reference instructions to reference a specific value instead of a memory location.

2.3.11.1 Identification

A literal is identified by an equal sign (=) preceeding the expression in the operand field of a memory reference instruction. The equal sign makes the entire expression in the operand field a literal.

2.3.11.2 Function

When a literal is encountered at load time, a word is reserved in the scratchpad area of memory to hold the computed value of the expression in the operand field. Memory addressing is then generated to access that scratchpad location. The following is an example of a literal:

```
LDA     =5
```

This statement means: Place the number 5 in the A Register. This must be distinguished from the statement:

```
LDA     5
```

which means: load the A Register with the contents of memory location 5. In the first case, the A register is loaded with a 5. In the second case, the A register is loaded with the contents of memory location 5.

2.3.12 Indirect Addressing

Indirect addressing is specified by preceding the expression in the operand field of memory reference instructions with an asterisk (*).

2.3.12.1 Assembler Generated Indirect Addressing

When an expression references a memory location which is out of direct addressing range, the assembler/loader automatically generates an indirect address pointer and indirect addressing modes for the instruction. If the instruction also specifies indirect addressing, the multi-level indirect bit will be turned on in the assembler/loader generated pointer.

2.3.12.2 Byte Mode Indirect Addressing

Only one level of indirect addressing may be used with byte mode memory reference instructions. Therefore, there is a chance that addressing errors may be encountered when using indirect addressing in byte mode. If the programmer specifies indirect addressing with byte mode instructions, he must be sure that the indirect address pointer is within direct addressing range of the byte mode instruction. Otherwise, a second level of indirect addressing would be required, and only one level of indirect addressing is possible in byte mode. For example:

```
LDAB      *ABC
```

Location ABC must be within direct addressing range of the LDAB instruction. If it is not, the assembler will flag an operand field error.

2.3.13 Indexing

Indexing is denoted by preceding the expression in the operand field of memory reference instruction with the "at" symbol (@). For example:

```
LDA       @ABC
```

If indirect addressing and indexing are specified within the same instruction statement, both the "at" symbol (@) and the asterisk (*) are used. The order of their use is not important, as long as they both precede the expression in the operand field:

```
LDA       @*ABC
LDA       *@ABC
```

The above statements will generate the same memory addressing modes.