Bolt Beranek and Newman Inc.

PLURIBUS DOCUMENT 5

Report No. 2931

advanced software

Report No. 2931 Bolt Beranek and Newman Inc.

PLURIBUS DOCUMENT 5: ADVANCED SOFTWARE

April 1975

Sponsored by:

Advanced Research Projects Agency
ARPA Order No. 2351
Contract No. F08606-73-C-0027 and
Contract No. F08606-75-C-0032

PLURIBUS DOCUMENT 5: ADVANCED SOFTWARE

PREFACE

"Pluribus Document 5: Advanced Software" is one of a set of nine which, taken together, provide complete documentation of the Pluribus line of computer systems. In the present document, Part 1, entitled "Lockheed System Software," is a brief overview of some of the programs available for the SUE* minicomputer, the processing element of the Pluribus. Parts 2 and 3 are manuals for two cross assemblers used to generate Pluribus code. Part 2 describes the version for the PDP-1d EXECIII system; Part 3 describes the version which runs on a PDP-10 TENEX system. Part 4, entitled "System Reliability Package," describes the standard software package which performs many of the Pluribus reliability functions.

Of the four parts of "Pluribus Document 5," parts 1, 2, and 3 are presently included here; part 4 is in production and will be added when it becomes ready.

*SUE is a trademark of the Lockheed Electronics Company.

TABLE OF CONTENTS

PREFACE

Part 1: Lockheed System Software . . . . . LEC

Part 2: Pluribus Assembly Language
and Operating Procedures
(PDP-1d Cross Assembler Version) . . . . . PDP-1 Assembler

Part 3: Pluribus Assembly Language
and Operating Procedures
(PDP-10 TENEX Cross Assembler Version) . . . . . PDP-10 Assembler

Part 4: System Reliability Package . . . . . Reliability

PLURIBUS DOCUMENT 5: ADVANCED SOFTWARE

PART 1: LOCKHEED SYSTEM SOFTWARE

Update History:

Lockheed material printed July 1973 (see footnotes on following page).

Following is a description* of some of the software written for the SUE** minicomputer available from Lockheed. While these programs were not written with the Pluribus architecture in mind, they can be used on Pluribus systems with at most minor modifications.

SUE software helps the user develop application programs. The programmer can write in assembly language, assemble, debug, and run on a variety of available machines. SUE system software features are:

Programs that run on SUE with 4K words of memory and an ASR-33 Teleprinter:

A comprehensive one-pass assembler that produces relocatable object code.

A relocating Link Loader that produces an executable translation of a main program and links external subroutines to it.

A Basic Loader that loads the output from the Link Loader into memory for execution.

A conversational debug program for on-line test and modification of assembled programs.

An I/O control system for communication between programs and peripheral devices.

Operator utility routines that interface between the program and the operator.

Test and maintenance programs for fast field analysis and repair of faults.

*Reproduced with the permission of the Lockheed Electronics Company from SUE Computer Handbook, edition of July 1973, copyright Lockheed Electronics Co., Inc.

**SUE is a trademark of Lockheed Electronics Company.

Programs that run on an IBM 360 or a Lockheed Electronics' MAC 16:

SUE Cross Assembler for listings and assembled code output identical to the SUE assembler.

SUE Link Loader that builds relocatable binary-formatted output for loading by the Basic Loader on the SUE processors.

Programs written in FORTRAN to run on a variety of machines:

SUE simulator for execution and testing of SUE-assembled object code on the IBM 360 computer or any large-scale computer with a ANSI-standard-FORTRAN Compiler.

SUE ASSEMBLER (LAP-2)

The assembler operates on a SUE computer with 4K words of memory and an ASR-33 Teletypewriter. An expanded version of the assembler that has additional features and operates additional peripherals can be used on machines of increased memory capacity. All assemblers for SUE are one pass, producing object code for the Link Loader. If additional peripherals are available an assembly listing is produced on the same pass; if not, then a listing pass is required. A Diagnostic Only option provides a listing of those statements in error.

Two cross assemblers are available for SUE. One operates on the Lockheed Electronics' MAC 16 Computer, the other on IBM 360 computers. Cross assemblers provide the user with assembly capability on readily-accessible processors having high-speed peripherals. These cross assemblers function identically to the SUE assembler and produce the same listing and object code.

An expanded assembler has many features not normally found in a minicomputer assembler. Some of these are

Full macro capability.
Fixed-point decimal conversion, single and double precision.
Floating-point decimal conversion, single and double precision.
Conditional assembly directives.
Listing formatting directives (EJECT, SPACE, etc.).
New operation definition capability (to allow assembling special op-codes implemented in a customized control ROM).

SUE LINK LOADERS

The SUE Link Loader is a relocating loader capable of building a core load by linking a main program and external subroutines. The loader accepts the output from the SUE assembler and generates output for loading by the Basic Loader. The operator may enter a relocation constant for changing the memory location of the linked program. Options include forcing the Link Loader to completion when external references remain undefined but are not necessary for the initial test runs; printing a memory map of the core load to provide the programmer with a reference for easy access of program modules; and defining externals not included in the subroutines.

The Cross Link Loaders that run on the MAC 16 and IBM 360 processors combine with the cross assemblers to provide a complete program generation system. The output can be loaded into the SUE computer for execution or loaded into the simulated memory of the SUE Simulator for execution and test.

SUE BASIC LOADER (BLOD-2)

The SUE Basic Loader loads the output generated by the Link Loaders into memory for execution. Record-by-record checking is performed with error detection causing an immediate halt to the system. Both Load and Go or Load and Halt operations are provided.

SUE BASIC OPERATING SYSTEM (BOS)

BOS serves as an off-line aid to the programmer when testing a new program. Some features included:

Change a word or byte in memory.
Execute a selected portion of the program.
Search the program for a key bit pattern.
Dump memory to the printer.
Dump memory in Basic Loader format to the punch.

SUE INPUT/OUTPUT CONTROL SYSTEM (IOCS)

IOCS provides a centralized I/O package that frees the user from details of dealing directly with peripheral devices. IOCS allows concurrent I/O operation of multiple devices and provides device independence to the user through assignment of device logical unit numbers to the various I/O devices at execution time.

The user calls IOCS from a calling sequence that uses a parameter list to define the requested operation. The parameter list offers several options to the programmer such as wait or no-wait for I/O completion and, upon device error, re-try or don't re-try the request. At the completion of any requested operation IOCS returns to the calling function for further processing.

SUE OPERATOR UTILITY INTERFACE PACKAGE (OUIP)

OUIP provides program-to-operator and operator-to-program communication. This package operates in conjunction with IOCS and provides the following functions to the user:

Input data from keyboard
Fetch name
Fetch numeric
Print message
Print numeric
Print carriage return/line feed
Print space
Print character
Input symbolic source line
Input binary formatted record
Output symbolic source line
Output binary formatted record
Program return

The user program can call any of these routines for ease in communicating with I/O devices. All symbolic and binary routines are interrupt driven and double-buffered. Each allows operator assignment of the desired peripheral for flexibility.

PLURIBUS DOCUMENT 5: ADVANCED SOFTWARE

PART 2: PLURIBUS ASSEMBLY LANGUAGE AND OPERATING PROCEDURES (PDP-1d Cross Assembler Version)

Update History:

Originally written as part of Hospital Computer Project memorandum Six-E, BBN Report No. 1422, May 1966, and extensively revised by W. Mann, S. Jeske, and D. Walden - January 1975.

TABLE OF CONTENTS

page

1. THE PLURIBUS MIDAS ASSEMBLY SYSTEM. 1
   1.1 The Midas Source Language. 2
      1.1.1 The Character Sets. 2
      1.1.2 Legal Strings 3
         1.1.2.1 Basic Strings. 3
         1.1.2.2 Complex Strings. 4
            1.1.2.2.1 Language Units. 4
            1.1.2.2.2 Combining Operators . . 5
   1.2 The Assembly Program 7
      1.2.1 The Current Location Counter. 7
      1.2.2 The Symbol Table. 8
      1.2.3 The Radix Indicator 8
   1.3 Defining Symbols 8
      1.3.1 Address Tags. 9
      1.3.2 Variable Names. 9
      1.3.3 Assigned Parameters 10
   1.4 The Use of Expressions 10
      1.4.1 Storage Words 10
      1.4.2 Constants 10
      1.4.3 Location Assignments. 11
   1.5 Instruction Format 11
      1.5.1 Arithmetic Instructions 12
      1.5.2 Jump Instructions 13
      1.5.3 Branch Instructions 13
      1.5.4 Shift Instructions. 14
      1.5.5 Operate Instructions. 15
         1.5.5.1 Special Functions. 15
         1.5.5.2 Memory Reference 16
   1.6 Source Program Format 16

TABLE OF CONTENTS (cont'd)

page

2. PSEUDO-INSTRUCTIONS. 19
   2.1 OCTAL and DECIMAL 19
   2.2 .ASCII. 19
   2.3 CONSTANTS 20
   2.4 VARIABLES 21
   2.5 DIMENSION 22
   2.6 EQUALS and OPSYN. 22
   2.7 NULL. 23
   2.8 OFFSET. 23
   2.9 REPEAT. 24
   2.10 START 25
   2.11 EXPUNGE 26
   2.12 WORD 26
   2.13 ØIF and 1IF 26
   2.14 PRINT, PRINTX, PNTNUM 28
   2.15 STOP. 28
   2.16 VERNUM. 29
   2.17 BT, BF, RET, STM, MTS, RTM, MTR 29
   2.18 Other Pseudo-Instructions 29
   2.19 Sample Program Section 29
      2.19.1 Sample Page of Program - Octal Listing . . 30
      2.19.2 Sample Page of Program - Hexadecimal Listing 31
3. MACRO-INSTRUCTIONS
   3.1 Macro-Definitions. 32
      3.1.1 Basic Format. 32
      3.1.2 Dummy Arguments 33
   3.2 Macro Calls. 35
   3.3 Storage of Macro-Instructions. 36
   3.4 Nested Macros. 36

TABLE OF CONTENTS (cont'd)

page

3.5 The Pseudo-Instructions IRP and IRPC. 39
4. OPERATION OF THE MIDAS ASSEMBLY SYSTEM 43
4.1 Preparation of a Source-Language Program 43
4.2 Performing an Assembly. 43
4.2.1 Initial Procedure. 43
4.2.2 The Control Language 43
5. BINARY OUTPUT FORMAT 49
6. ERROR CHECKING. 50

APPENDIX A. Midas Character Set 55
A.1 Alphabetic 55
A.2 Punctuation 55
A.3 Combining Operators. 56
A.4 Illegal. 56
A.4.1 Generally Illegal 56
A.4.2 Illegal Except Within a Macro-instruction or an IRP. 57
A.5 Ignored Except Within a Macro-instruction or an IRP 58
APPENDIX B. Symbols in Permanent Midas Vocabulary. . . . 59
B.1 Pluribus Instruction Symbols 59
B.1.1 Symbols Associated with Memory Reference Instructions 59
B.1.2 Symbols Associated with Branch Condition, Shifts and Jumps 60
B.1.3 Symbols Associated with Control (Class Ø) Instructions 61
B.1.4 Symbols Associated with Register Selection
B.1.5 Symbols Associated with the Scientific Instruction Set 62

TABLE OF CONTENTS (cont'd)

page

B.1.6 SUE to Pluribus Instruction Equivalence. 64

B.2 Pseudo-Instructions. 65

APPENDIX C. Teletype Code Conversion. 69

C.1 Characters with 6-bit Internal Representation . . 69

C.2 Characters with 12-bit Internal Representation. . 71

1. THE PLURIBUS MIDAS ASSEMBLY SYSTEM

The assembly system described in this report was adapted from the Bolt Beranek and Newman PDP-1 Midas assembly system which in turn was adapted from the Midas Assembler originally written for the PDP-1 at Massachusetts Institute of Technology. Throughout the remainder of this report we use "Midas" alone to stand for "Pluribus Midas".

The Midas Assembly Program, while offering all the normal assembly features, belongs to a class of extended assembly programs referred to as macro-assemblers. Macro-assemblers such as Midas provide an extensive set of control operations (called pseudo-instructions) which, in principle, make it possible for the assembly program to perform computations analogous to those of any object program it can produce.

Most notable in this respect are the Midas macro-instruction features, which permit the programmer to define a special purpose abbreviative language to suit his own needs. Using pseudo-instructions provided for that purpose, a user can name a complex coding sequence and provide for varying parameters. Other pseudo-instructions available in Midas provide means for performing assembly-time list processing, symbol manipulation, and loop termination as well as for changing the course of assembly in response to certain conditions.

Formal constraints on the construction and manipulation of symbols are few, so the programmer may, within the range of processor capabilities, vary formats to suit particular programs. The programmer is free to ignore any of the special features and use Midas as a simple mnemonic code translator.

The formal rules of the Midas source language and basic processor references are described in this Section. Section 2 describes the functions and formats of all system pseudo-instructions. Section 3 explains the use of macro-instructions. Section 4 provides instructions for performing an assembly. Error conditions that are detected during an assembly and associated error messages are listed in Section 5.

The notation used in this volume includes some special symbols. ↵ represents carriage-return/line-feed. →| represents a tab. Quotation marks indicate the pressing of the Control key on the Teletype keyboard. The symbols < > are used to enclose text that might normally be set off by quotation marks. Unless otherwise noted, all integers appearing in the text are octal integers.

1.1 The Midas Source Language

A Midas source program is a string of alphanumeric and operational characters. From this string the Midas assembly program produces the words that make up the object program, located properly in memory. In order to accomplish this, the assembler must interpret the source program as a series of meaningful strings. In most cases, a string that is meaningful to the assembler represents a word in the object program. In other cases, the string may direct the assembler to produce several or no words in the object program.

This section describes the mechanics of creating legal Midas character strings, the references that the assembly program uses in associating character strings with binary values, the conventions that instruct Midas as to the type of value or actual value a string is to represent, and the overall source program format requirements.

The source program, described above as a single string of characters is, more precisely, a system of arbitrary strings, each of which consists of individual characters juxtaposed according to formal conventions. The construction of legal strings is hierarchical in nature. The lowest level constituent strings are formed from the alphabetic members of the character set. Higher level constituent strings are constructed from previously defined constituent strings using those members of the character set which function as combining operators. The type of object a constituent string represents is indicated by punctuation characters.

1.1.1 The Character Set

The complete character set from which the source language is constructed is included in Appendix A. It consists of all characters on the Teletype keyboard not specified as illegal.

The character set is generally divided into the following categories:

alphanumeric characters: The letters A-Z and the character, <.> (period)*, which may be constituents of symbols; and the digits Ø-9, which may be constituents of symbols or integers.

*The character <.> also denotes a decimal number (as in <1Ø.>) and may also be used to represent the value of the current location.

combining operators:	Single characters representing fixed arithmetical or logical operations to be performed by Midas.
punctuation characters:	These serve as string delimiters. A string-delimiting character may serve a variety of purposes depending on its use. String delimiters in general identify individual strings and often indicate the manner in which a string is to be interpreted. See Appendix A for a complete listing. The most generally used delimiters are space, tab, and carriage-return/line-feed.

1.1.2 Legal Strings

1.1.2.1 Basic Strings

The minimum character strings required to represent values in the source program are symbols or integers, formed as follows:

Integers

An integer is a string of digits (Ø,1,....9) that is evaluated as octal or decimal according to the radix prevailing at its appearance. The integer value is its representation as an 18.-bit binary number which restricts integer values to 777777 if the radix is set for octal and to 262143 for decimal. An integer above these limits will be evaluated modulo $(2^{18.}-1)$.*

---

* The Pluribus Midas assembler described here uses 18.-bit quantities internally as a result of its derivation from an assembler for the PDP-1, an 18.-bit machine. At load time the 18.-bit quantities are reduced to 16.-bit quantities suitable for the Pluribus according to rules given in a later section.

Symbols

A symbol is defined as a string of characters, the first six of which must distinguish it from all other symbols.

Letters, numbers, and periods may be used as symbol constituents, but at least one must be a letter, or the first must be a period.

Longer symbols, useful for mnemonic or documentary purposes, may be used, since Midas ignores any character except a terminator in excess of six.

Symbols for macro names and pseudo-instructions are subject to the same restrictions as symbols for numerical values.

Note that the symbols <READIN> and <READINTAPE> are both legal symbols; if used in the same source program, they both will appear to the assembler as <READIN> and will be used interchangeably. If distinct symbols are desired, care must be taken to differentiate symbols within their first six characters.

1.1.2.2 Complex Strings

1.1.2.2.1 Language Units

Complex strings may be formed from basic strings by use of characters that are provided as combining operators. Although integers and symbols are the only basic strings in the Midas language that are formed by purely alphanumeric concatenations, a complex string may be bracketed and function in the same way as a basic string in a new combination. In discussing the construction of complex strings, language units will be called either syllables or expressions as defined below:

Syllable

A syllable is any component string of an expression whose value is independent of its use in the expression. An expression enclosed in brackets may be used as a syllable to form other expressions. In addition, the pseudo-instruction <.> is used to represent the current value of the location counter modulo (2^{16}·) and functions as a syllable.

Expression

An expression is a string consisting of one or more syllables separated by combining operators.

1.1.2.2.2 Combining Operators

The characters listed below according to function, are the Midas combining operators. Quotation marks denote that the Control key must be pressed while typing the character.

Product Operators	Function
"T"	Folder integer multiplication
"X"	Logical disjunction (exclusive OR)
"U"	Logical union (inclusive OR)
"A"	Logical intersection (AND)
"Q"	Quotient
"R"	Remainder

Additive Operators	Function
+ or space	Addition, mod $2^{18}-1$
- (minus)	Addition of the one's complement

Note that <A"T"B> results in an 18.-bit quantity equal to the sum of the unsigned magnitudes of the high and low order halves of the 36.-bit product produced by regular multiplication of the two 18.-bit quantities A and B. If both A and B are small enough, this function produces the ordinary product. When evaluating expressions, Midas performs operations from left to right, all product operations preceding ordinary additive ones. An additive operator which occurs with no syllable before it is processed first. Minus complements the syllable to its right, while plus and space are ignored.

When an expression is punched on paper tape to be loaded into the PLURIBUS, it is converted to 16.-bit two's complement as follows: If negative, one is added to the expression, then it is truncated to 16.-bits. (Note: -Ø is converted to zero in two's complement.)

Although this conversion process is ordinarily transparent to the programmer the following pitfall exists: When writing expressions which include logical operators, the programmer must bear in mind that Midas performs its internal arithmetic in one's-complement, then converts. For example, if TBITS=5

```
<AND A1 L   TBITS"X"-1>
```

will not work as might be expected. Minus one (-1) is 777776 in one's complement, and 5"X"777776=777773; as this is negative (in 18.-bit notation) it is ultimately converted to 177774. What was expected was probably TBITS"X"177777=177772. Using -Ø will not help either: 5"X"-Ø=777772 which is converted to 177773. While

```
<AND A1 L   -TBITS-1>
```

works, it is not recommended since it relies on an obscure characteristic of the assembler; instead set ONES=177777 and write

```
    <AND A1 L     TBITS"X"ONES>
or  <AND A1 L     TBITS"X"FFFF!>
```

Examples: In the following examples, various equivalent expressions are shown, and their component syllables listed:

a)*	Expressions:	ENB 1Ø	1Ø ENB	4Ø1Ø
	Syllables:	<ENB>,<1Ø>	<1Ø>,<ENB>	<4Ø1Ø>
b)	Expressions:	ENB I	INH	42ØØ
	Syllables:	<ENB>,<I>	<INH>	<42ØØ>

---

* Assume that ENB=4ØØØ, I=2ØØ, and INH=42ØØ

c)	Expressions:	7-2"U"3	6-2	4
	Syllables:	<7>,<2>,<3>	<6>,<2>	<4>
d)	Expressions:	+A	A	2"T"A-A
	Syllables:	<A>	<A>	<2>,<A>
e)	Expressions:	A+B"T"C	A+[B"T"C]	
	Syllables	<A>,<B>,<C>	<A>,<[B"T"C]>	

Note that the expression [A+B]"T"C is not equivalent to those of (e).

1.2 The Assembly Program

In order to interpret symbols and integers and to assign them to memory locations, Midas must make references to the Current-Location Counter, the Symbol Table, and the Radix Indicator, which are described below.

1.2.1 The Current Location Counter

The assembler assigns assembled words to be stored in sequential locations, 2 bytes per location, starting from any given even location. A register in the assembly program, referred to as the Current Location Counter, is incremented by 2 whenever a word is assigned, and indicates the location which will be assigned to the next word assembled. It is initially set at 11Ø and counts upward modulo (2^{16}•). Conventions are provided in the source language so that the programmer may assign a numerical value to the current location counter, thus specifying the first location in which a sequence will be stored.

A symbol may be defined as the current, specific value of the location counter at any time during the assembly and used throughout the source program. In addition, the pseudo-instruction <.> always has as its value the current location counter. The pseudo-instruction OFFSET, described below, allows code assembled to run at one location to be loaded into a different location. In this case an offset is added to the value of the current location counter before it is used for purposes other than actually assigning the word to a location in core.

Some programmers might tend to think of symbols for addresses and symbols for words as quite different entities. However, in the Pluribus Assembler, there is no difference. For example, if a computation to be performed requires the number 1ØØ in accumulator A1 we might <LDA A1 L> the number rather than provide a register containing it, and if the number 1ØØ is represented by the symbol <A>, we can <LDA A1 L→A> whenever we need 1ØØ in accumulator A1 whether <A> derived its value as an address tag or as a parameter.

1.2.2 The Symbol Table

Symbols, when associated with a value, are entered in the Midas Symbol Table, which is used as a reference by the assembler. Mnemonic symbols for Pluribus instruction codes are part of the initial contents of the symbol table. (A list of these is included in Appendix B). These, and programmer defined symbols which stand for numerical values, are "substantive" symbols. Two other symbol types are included in the symbol table: pseudo-instruction names and macro names. These are referred to as "operational" symbols; there is no single address or single 18.-bit number with which they are synonymous. All pseudo-instruction names are included initially in the symbol table, defined by a reference to an assembler routine. A macro name is entered in the symbol table when a macro-instruction is defined.

Midas assembles a source program in two passes; that is, two complete scans of the source program are required to produce an object program. This allows symbols to be associated with values at any point in the source program, without restricting their use prior to definition.

1.2.3 The Radix Indicator

Integers are interpreted as octal or decimal according to the radix prevailing when they are encountered by Midas. A register in Midas, called the current radix indicator, is initially set to accept octal integers. The pseudo-instructions DECIMAL and OCTAL may be used to alter and reset this indicator.

1.3 Defining Symbols

The Midas symbol table, described earlier, functions as a dictionary during the translation process. Each symbol introduced by the programmer is inserted together with its value

into the symbol table during Pass 1. The value of a symbol is referred to as its definition. Numerical definition of a symbol may be accomplished by its appearance as an address tag, a variable name, or in a parameter assignment. Symbols may also be defined as macro-instruction names or in terms of other symbols. Macro name symbols are discussed in the section on macro-instructions. The establishment of symbol synonyms is discussed in Section 2 in connection with the pseudo-instructions EQUALS and OPSYN.

The three basic formats that Midas allows for assigning a numerical value to a symbol are described below.

1.3.1 Address Tags

A symbol is identified as an address tag if it is terminated by a comma or a colon. An address tag is equated with the value of the location counter plus the offset. The value is entered into the symbol table modulo (2^{16}•). A symbolic address tag identifies a line in the source program text and is required only for lines referenced within the text.

An expression, such as <A+4>, terminated by a comma or colon, is checked by Midas to make sure that it is equal to the current location counter plus the offset. Since relative addressing using address arithmetic is possible, any untagged word might be referenced by such an expression. If a previously defined symbol occurs as an address tag, the symbol is not redefined; its value is checked and must agree with that of the current location counter plus the offset. These checks provide the programmer a facility for verifying that the location counter is properly set or that a symbol is properly defined. They also detect multiple uses of a single symbol.

1.3.2 Variable Names

A programmer may direct the Midas assembler to reserve a sequence of storage words for variable quantities produced by a computation. Symbols may be substituted for these locations before they are known. The assembly program classifies such symbols as "undefined variables" and assigns them provisional relative values. A string is classified as an undefined variable by the inclusion of the character <#> in some occurrence of the identifying string itself. The symbol is stored in the symbol table without the <#> and may be referred to with or without it.

Actual values are assigned for all variables still undefined at the appearance of the pseudo-instruction VARIABLES. The following are legal variable names: <ABC#>, <#ABC>, <AB#C>. The second form listed is preferred.

1.3.3 Assigned Parameters

A programmer may assign any numerical value to a symbol with a parameter assignment statement. The format is

SYMBOL=EXPRESSION

The symbol to the left of the equals sign is entered in the symbol table, and the value of the expression to the right is entered as its definition. If no value can be obtained for the expression, the symbol is not defined.

1.4 The Use of Expressions

The rules for forming expressions were given earlier. The evaluation of an expression depends on its context in the source program as well as on the value of its component syllables. Contexts in which Midas evaluates expressions are described below.

1.4.1 Storage Words

An expression terminated by a tab or carriage return is a storage word. A storage word, when encountered by Midas, is evaluated and assigned to the memory location equal to the value of the current location counter. The contents of a storage word may ultimately be used as an instruction and/or operated on by an instruction, depending on the use of the word in the object program.

1.4.2 Constants*

Constant values required by a program need not be introduced as storage words in the source program. The constant expression desired, enclosed in parentheses, may appear literally as the operand of an instruction.

For example, an instruction to subtract 1ØØ. from accumulator A1 could be written as <SUB A1→(1ØØ.)>. Midas will

---

* This feature is rarely used.

generate a word containing the value 1ØØ. The string <(1ØØ.)> is called a constant syllable, and the address of the word containing <1ØØ.> is substituted for it. Note that (1ØØ.), (5Ø.+5Ø.), or (A+25.), where A=75. are equivalent constant syllables and will all refer to the same location.

Constants may appear within constants to any depth, as in

LDA A1 I→((AB)) (1)

The right parenthesis of a constant syllable may be omitted if the constant is followed by a word terminator. For example,

LDA A1 I→((AB ↵ (2)

is equivalent to (1) above. Omission of the right parenthesis in

LDA A1 I→((AB)-Z ↵ (3)

would, however, change the meaning.

1.4.3 Location Assignments

A location assignment is an expression immediately followed by a slash. When Midas encounters a location assignment, the expression is evaluated, and the location counter is set to this value (rounded down if odd). If an expression that is used to assign a location contains any undefined symbols when encountered by Midas on Pass 1, the current location becomes indefinite. This means that the definition of address tags is inhibited, and the value of <.> is undefined, until a defined location assignment occurs, and at that time the counter again becomes definite. On Pass 2, an undefined symbol in a location assignment will cause an error message (USL). The undefined symbol is taken as zero, and the location remains definite. The Midas command "E", discussed in Section 4, permits the programmer to arrange for Midas to type a message if the location becomes indefinite on Pass 1.

1.5 Instruction Format

Basic Pluribus instructions come in five major classes (Arithmetic, Jump, Branch, Shift, Operate), each of which will be considered separately below. The scientific processor instruction set is listed in appendix B.

1.5.1 Arithmetic Instructions:

Typical Example	FOO:	ADD A3 X1	→| DOG
Complex Examples	FOO:	ADD A3 X1 A B I M	→| DOG
	FOO:	ADD A4 X7 D B X	

This type of instruction features an op code (one of 9), an Accumulator (one of 8), an Index register (one of 8), five separate mode switches (2-5 options each) and an address. The recommended listing format is in the following order: <op code> space <Accumulator> space <Index register> space <MODE 1 switch> space <MODE 2 switch> space <MODE 3 switch> space <MODE 4 switch> space <MODE 5 switch> Tab <address>.

Of course, it doesn't actually matter what order is used, since Midas is just adding up predefined values.
OP codes: LDA, SUB, ADD, AND, IOR, EOR, CMP, TST, STA

ACC : AØ, A1, A2, A3, A4, A5, A6, A7

Index : XØ, X1, X2, X3, X4, X5, X6, X7

MODE 1 : W, A, D

MODE 2 : B

MODE 3 : I

MODE 4 : M, L, E, R

MODE 5 : X

The Op codes, ACC, and index are pretty straightforward after reading the SUE computer handbook. (STA is the same as LDA M.) The modes are a little more complex. W (for Word) can be used anywhere to make listing formats line up in columns (W=Ø). A and D specify Auto Increment and Auto Decrement, respectively. B specifies halfword (byte) operation. I specifies Indirect addressing. M specifies "To Memory". L specifies the address field is to be used as a Literal. E specifies the 4 bit field where the index usually lives is a literal. R means the operand is the contents of the specified index register. X specifies a single word instruction--what the SUE handbook calls "indexed" (not "direct address"), i.e., addressed through an index register only. Note: not all possible

combinations of modes are legal Pluribus instructions; in particular, L, E and R cannot have other mode options.

More Examples

```
LDA A3 L     →| 77        /put 77 in AC 3
LDA A3 B 7                /put  7 in AC 3
ADD A3 M     →| DOG       /add AC3 to DOG, result in DOG
```

1.5.2 Jump Instructions:

```
Examples:   JMP              →| FOO
            JSB  A7          →| FOO
            JMP  X1          →| FOOTAB
            JSB  A7  X1      →| FOOTAB
            JMP  I           →| FOO
            JMP  X3  X
            JSB  A2  X3 X
```

1.5.3 Branch Instructions:

```
Examples:   BT EQ, DOG
            BF EQ, DOG
```

BT and BF are "Branch" pseudo-instructions. EQ is one of a set of thirteen branch conditions. DOG is the location to branch to. BT stands for Branch if True. BF stands for Branch if False. The Branch Conditions, their meanings, and how they are set are listed in the following table:

Condition	Meaning	Set By
TR	True	Lockheed (always)
EQ	Equal	last CMP
GT	Greater Than	last CMP
OV	Overflow	ADD, SUB, some shifts
CY	Carry	ADD, SUB, some shifts

Condition	Meaning	Set By
F1	Flag 1	
F2	Flag 2	special instructions
F3	Flag 3	
LP	Loop Complete	last AUTO INC/DEC
OD	Odd	last Arithmetic (except CMP)
ZE	Zero	last Arithmetic (except CMP)
NG	Negative	last Arithmetic (except CMP)

If an undefined branch condition is used, a "UBR" error is generated by the assembler. If the displacement is out of range (D>127. or D<-128.) a "DOR" error is generated.

1.5.4 Shift Instructions

The Pluribus basically has 16 shift instructions. The sixteen are highly bit coded, and can be formed by combining two symbols, each of which can take four forms.

```
Examples:    LI  AO  A3   6

             LX  LL  A3   X2

             RI  LO  A3   5

             RX  LC  A3   X1
```

The first symbol specifies shift direction and size. The second specifies one of four shift types.

```
AO  Arithmetic Open    :   use carry - off end - sign extend

LL  Logical Linked     :   use carry - rotate

LO  Logical Open       :   no carry - off end

LC  Logical Closed     :   no carry - rotate
```

LI Left by IMMEDIATE

LX Left by INDEX Register (last 4 bits)

RI Right by IMMEDIATE

RX Right by INDEX Register (last 4 bits)

1.5.5 Operate Instructions

These come in two varieties, called special function and memory reference.

1.5.5.1 Special Functions

Defined instructions are:

HLT	halt
RST	reset status bits
SST	set status bits
ENB	enable interrupt levels
INH	inhibit interrupt levels
ENW	enable and wait
INW	inhibit and wait

In the inhibit/enable instructions, bits 3-Ø indicate levels. E.g., to enable levels 1 and 3 use the instruction

```
ENB 5
```

or to inhibit level 4 and wait for an interrupt use the instruction

```
INW 8
```

In the status bit instructions, bits 6-Ø are set for bits in the status word to set or reset; e.g., to clear the three program flags, use the instruction

```
RST 16Ø
```

Bit	Octal Coding	Hex Coding	Status Bit
Ø	1	1	E = Equal
1	2	2	G = Greater than
2	4	4	V = Overflow
3	1Ø	8	C = Carry
4	2Ø	1Ø	F1 = Program flag 1
5	4Ø	2Ø	F2 = Program flag 2
6	1ØØ	4Ø	F3 = Program flag 3

1.5.5.2 Memory reference

Defined instructions are:

RET Return from interrupt

STM Status to memory

MTS Memory to status

RTM Registers to memory

MTR Memory to registers

These are implemented as pseudo-instructions. If the value of the argument is 255.or less, a reference to "executive core" is generated; otherwise a displacement is computed. A "DOR" error is possible here.

1.6 Source Program Format

Midas begins processing a source program after it encounters a title. A title may be any string of characters terminated by a carriage return. Initial carriage returns are ignored. The end of the source program is indicated by the appearance of the START pseudo-instruction.

The portion of the source program which is to be assembled, referred to as the body, is composed of character strings. These strings are processed by Midas sequentially.

Comments may be entered throughout the source program. Any string of text characterized as a comment will be ignored by the assembler. A comment is introduced by a slash and must be preceded and terminated by either a tab or carriage return.

Two sample programs follow for solving the equation:

Z=7Q+V/2 for Q=4Ø, V=6Ø.

The two symbolic programs produce identical machine language programs as given below:

Sample Program 1

```
1ØØ!/→LDA A1   →Q
      LI AO A1 3
      SUB A1   →Q
      LDA A2   →V
      RI AO A2 1          /Good Thing V is even
      ADD A1 R X2
      STA A1   →Z
      HLT

Z,  →Ø9
Q:  →4Ø                   /either : or , works
V,  →6Ø

      START 1ØØ!
```

Sample Program 2

```
1ØØ!/→LDA A1   →(4Ø)
      LI AO A1 3
      SUB A1   →(4Ø)
      LDA A2   →(6Ø)
      RI AO A2 1
      ADD A1 R X2
      STA A1   →#Z
      HLT


      VARIABLES
      CONSTANTS
      START 1ØØ!
```

Resulting Machine Language

```
4ØØ    Ø7ØØ3Ø
4Ø2    ØØØ432
4Ø4    121223
4Ø6    Ø7Ø43Ø
41Ø    ØØØ432
412    Ø7ØØ5Ø
414    ØØØ434
416    123241
42Ø    Ø45Ø22
422    Ø3ØØ3Ø
424    ØØØ43Ø
426    ØØØØØØ
43Ø    ØØØØØØ
432    ØØØØ4Ø
434    ØØØØ6Ø
```

It should be noted here that the above examples were primarily pedagogical in nature. The second example, in particular, totally ignored the ability to have literal fields as part of the instruction. The strings <CONSTANTS>, <VARIABLES>, <START> in the sample programs are pseudo-instructions, which will be discussed in the following section.

2. PSEUDO-INSTRUCTIONS

Pseudo-instructions are source-language expressions that serve to direct the assembly process. A pseudo-instruction statement consists of a pseudo-instruction symbol terminated by a delimiter and followed by arguments as required. Unless otherwise noted, a pseudo-instruction statement is terminated by a tab or a carriage return. Pseudo-instruction symbols, like all other symbols, are identified by no more than six characters. Thus, pseudo-instruction symbols composed of more than six characters may always be abbreviated. For example, DIMENSION may be shortened in use to DIMENS. The pseudo-instruction repertoire is described below with regard to format and function.

2.1 Octal and Decimal

When integers are encountered, they are interpreted as octal or decimal according to the value of the prevailing radix indicator. The pseudo-instructions OCTAL and DECIMAL reset the radix indicator, which is set by Midas to OCTAL at the beginning of each pass. An integer syllable followed directly by a period will be interpreted as a decimal number regardless of the current radix and will not change the radix value. Likewise, any syllable followed directly by an exclamation point will be interpreted as a hexadecimal number, regardless of radix.

2.2 .ASCII

The pseudo-instruction .ASCII is used to assemble a string of characters, two per word, into successive words in the object program. .ASCII may not be used in constants or in other contexts where an expression is required.

An .ASCII statement consists of the symbol <.ASCII> terminated by a delimiter and followed by a string of characters. The first character in the string is used as a delimiter of the text string itself; it is not stored as part of the text string, and its reappearance terminates code storage. Thus, if given the string <.ASCII /MESSAGE/>, Midas will store character code for the word, MESSAGE. A 12.-bit code character* should not be used as a text delimiter. Only six bits at either end of a string are interpreted as delimiters, so the remaining six bits of a 12.-bit character would be included in the stored text somehow.

---

* See Appendix C.

If the character <#> is used in the argument of .ASCII, it is handled in an exceptional way; it is stored as end-of-message (ASCII Ø) rather than as its own ASCII Code Configuration, 243. Consequently, there is no provision for entering <#> as actual text. Note that adding <#> to the end of an even number of characters will force a full word (two bytes) of Ø after the last character. An odd number of characters will cause the unused (odd) byte of the last word (after the last character) to be Ø.

Three pseudo-instructions--CONSTANTS, VARIABLES, and DIMENSION--are provided to direct automatic storage assignment of words for constant and variable data. Variable data words may be generated individually by reference or as fixed length arrays obtained with the DIMENSION pseudo-instruction. Constant data words are generated to accommodate literal references.

2.3 CONSTANTS

The pseudo-instruction CONSTANTS effects the allocation of constant syllables to storage words containing constant values, beginning at a location whose value is equal to that of the current location counter at the appearance of CONSTANTS. When a constant syllable is allocated, its address (plus its offset) is substituted for it at all references. If different expressions enclosed within parentheses have the same value, they are considered to be the same constant syllable and are associated with only one location. The pseudo-instruction CONSTANTS may be used no more than 8. times in the same program. The number of distinct constants on pass 1 in any one constants area is limited to 64..

Since storage space for constants is allocated on Pass 1 when some expressions may not be definite, the number of registers reserved in the constants area may exceed the number Midas needs when all references have been consolidated; thus, a gap of unused registers may arise between a constants area and any subsequent portion of the object program. The pseudo-instruction UNCON has as its value the size of the most recent of these gaps. It is zero on pass 1.

The following examples show symbol prints following Pass 1 and Pass 2 for the same program.

CONSTANT AREA RESERVED, INCLUSIVE

FROM	TO
326	332

indicates registers reserved during Pass 1. At the completion of Pass 2, the printout

```
CONSTANTS AREA, INCLUSIVE
FROM        TO
 326        332
```

indicates those registers actually containing constants.

2.4 VARIABLES

The value of the current location counter at the appearance of the pseudo-instruction VARIABLES on Pass 1 marks the beginning of the storage area allocated to variables. All variable names classified as undefined are assigned locations at this time. The relative value assigned to an undefined variable is added to the value of the first location in the sequence (plus the offset) and the result obtained entered as its address. Each variable is assigned by Midas to a storage word, whose initial contents are unspecified.

When a variables area has been completely allocated, the value of the current-location counter is that of the next location at which a storage word will be assembled. If VARIABLES appears when the location counter is indefinite, it is inadmissible. The Midas command "E" (described in Section 4) can be used to help locate the problem if the location is indefinite on Pass 1 when VARIABLES is used.

In the current version of Midas the pseudo-instruction VARIABLES may be used no more than 8. times. If the maximum is exceeded, the error comment TMV (too many variables) is typed. The number of defined variables, however, is limited only by the capacity of the symbol table.

At the occurrence of VARIABLES on Pass 2, Midas compares the value of the current location counter with the value which was associated with that variables area on Pass 1. A disagreement is noted by the error message VLD (variables location disagrees), which indicates that errant symbol definitions or macroexpansions have altered the sequence of assembled words.

2.5 DIMENSION

The pseudo-instruction DIMENSION reserves space in the variables storage area, which may be referenced relative to a symbolic address. DIMENSION is used to set up fixed-length arrays. The pseudo-instruction and the name and extent of any number of arrays constitute a DIMENSION statement, according to the following format:

```
DIMENSION NAME1(LGTH1),NAME2(LGTH2)
```

The entire statement is terminated by a tab or carriage return and requested blocks are separated from one another by commas. The array name must be a legal symbol that has not been previously defined. The extent must be stated as an expression whose syllable values are known when the DIMENSION statement is encountered on Pass 1. It is given in bytes.

2.6 EQUALS AND OPSYN

The pseudo-instructions EQUALS and OPSYN permit a user to establish symbol synonyms, representing the same value. The format is

```
EQUALS SYNONYM,SYMBOL
```

Or

```
OPSYN SYNONYM,SYMBOL
```

where <SYNONYM> must be a legal symbol string and the <SYMBOL> with which it is identified must be previously defined. <SYNONYM> is assigned the same numerical or operational value as <SYMBOL>, and the two are thereafter synonymous. OPSYN operates on Pass 1 only; EQUALS, on both passes. The following example illustrates the difference between the two.

Let us say, for example, that a programmer wanted to use the macro-instruction facilities to redefine a pseudo-instruction such that the new instruction was a function of the old. The original pseudo-instruction would have to be represented by a different symbol; otherwise, its appearance in the definition would act as a macro call, resulting in a closed loop.

For example, if one writes

```
EQUALS OCT,OCTAL
```

and then defines a macro-instruction OCTAL in terms of OCT,

which now calls the pseudo-instruction, on Pass 2 OCT will again be made equivalent to OCTAL, which has been redefined. OCT then no longer references the pseudo-instruction, and the loop avoided on Pass 1 will occur anyway on Pass 2. If one uses OPSYN, however, OCT will be associated with OCTAL as desired on Pass 1 only and retain its identity with the original pseudo-instruction on Pass 2.

2.7 NULL

The NULL pseudo-instruction performs no action, but can be used as a substitute for symbols no longer needed in a program.

Some programs are required to be compatible with various environments (different machines, data bases, etc.), and a function performed frequently in one usage may not be performed at all in another. For example, a symbolic program may contain complex macro-definitions that need not be assembled in one instance but must be available for other processings. In this case, the macro names may simply be equated with NULL, as in

```
EQUALS MACRO,NULL
```

Another case in which NULL is useful arises in connection with macro-table storage space and the "garbage collector." When the amount of space available for the storage of macro-definitions is exhausted, the garbage collector will search the table for definitions which no longer have a reference in the symbol table and will recover such space by consolidating the remaining table entries. The symbol-table reference to a macro that has been redefined is automatically transferred to the latest definition. In the case of macros that have not been redefined but are simply no longer needed, the reference must be suppressed in order to notify the garbage collector of the available space. EXPUNGE can also be used for this function.

2.8 OFFSET

The pseudo-instruction OFFSET is used to set the value of the offset count, whose relation to the current location counter was described in connection with symbol definition. The pseudo-instruction format is

```
OFFSET EXPRESSION
```

where the value of the expression (positive or negative) is stored as the offset count. When the offset count is any value

other than zero, a symbol value derived as an address tag will not equal the core location of the storage word at which it is loaded. For example, the coding:

```
        OFFSET 6
ABC,→|LDA A1 L    →|1ØØ
     |  JMP       →|ABC
```

occurring when the current location counter contains 1ØØ will be assembled as:

```
1ØØ/ →|LDA A1 L →|1ØØ
     |  JMP     →|1Ø6
```

A portion of the object program that was assembled under these conditions is not executable at the location it occupies if storage word expressions use these symbols as referents. The offset capability is, however, useful in creating a body of data independent of its core location, yet internally consistent.

The effect of one OFFSET declaration is terminated by the appearance of another. If a return to the normal sequence is desired, the programmer must set the offset count to zero. OFFSET is used in connection with memory "renaming" and in constructing item maps.

2.9 REPEAT

REPEAT instructs Midas to assemble a specified portion of the source program a specified number of times and thus relieves the programmer of the necessity for source language repetition of a repetitive object program sequence. The format is

```
REPEAT EXPRESSION, TEXT
```

where EXPRESSION is the count of the REPEAT, specifying the number of iterations desired, and the TEXT is the source program section to be iterated, called the range of the REPEAT. The count must be defined when Midas encounters the REPEAT on Pass 1; otherwise, the range is ignored and the error print <USR> occurs.

A carriage return is used to terminate the entire instruction; tabs may be used to denote storage words. Brackets may be used to enclose portions of the range or the entire range; this allows carriage returns or nested brackets to be included.

Since a REPEAT merely serves to reproduce a string, the range may include any elements of the source language, including other REPEATS and macro calls. (An internal REPEAT, unless it is at the end of the range, must be bracketed; otherwise its terminating carriage return would also terminate the first REPEAT.)

In the following example, the statement

REPEAT 2, LDA A1 →|A →|ADD A1 →|B →|STA A1 →|C

will generate for assembly the coding equivalent to

LDA A1 →|A
ADD A1 →|B
STA A1 →|C
LDA A1 →|A
ADD A1 →|B
STA A1 →|C

If the count of a REPEAT is zero or negative the range is not processed.

2.10 START

The START pseudo-instruction directs Midas to stop reading characters. START must appear at the end of every source-language file and may take as an argument an expression denoting the starting address of the object program. The format is

START EXPRESSION

At the end of Pass 2 in response to command "J", (Section 4), Midas appends to the binary output a one word Jump Block containing the argument of the last START processed.

2.11 EXPUNGE

The pseudo-instruction EXPUNGE removes symbols from the symbol table. The format is

```
EXPUNGE SYM1,SYM2,SYMN
```

where the argument is a list of symbols, separated by commas and terminated by a tab or carriage return. Any type of symbol may be expunged. Midas ignores undefined symbols in the list. If any member of the list is not a legal symbol, Midas ignores the rest of the list. An expunged variable will not be defined unless it appears again with <#> after the EXPUNGE; <#> itself may not appear in the argument list.

2.12 WORD

The pseudo-instruction WORD appends 18.-bit computer words, specified by the argument(s) of the pseudo-instruction, to a block of binary output. The format is

```
WORD EXPRESSION
```

or

```
WORD EXPR1,EXPR2,...EXPRN
```

The appended words are not necessarily part of the object program; their values are selected to produce special binary formats when needed. For example, words might be appended in order to accommodate a particular loader or to insert jump blocks before the end of assembly. Normal binary output format is discussed in Section 4.

2.13 ØIF and 1IF

A programmer may find it useful, particularly when handling complex macro-instructions, to be able to test the value of an expression and to condition part of the assembly on the result. Such testing is effected by the pseudo-instructions ØIF and 1IF, in conjunction with symbols called qualifiers, which represent tests available. The tests are as follows:

Qualifier	Condition is true if:
VP	the evaluated expression is greater than or equal to ±Ø

VZ	the evaluated expression is equal to ±Ø
P	Pass 2 is being performed
D	the expression tested is completely defined
N	the argument contains no characters (usually a dummy symbol of a macro or IRP)

The format of conditional statements is

```
ØIF VP EXPRESSION
1IF VZ EXPRESSION
ØIF D  EXPRESSION
1IF N  SYMBOL
```

where the test requires an argument, and otherwise, ØIF P. The value of 1IF is 1 if the condition is true, Ø if false; the value of ØIF is Ø if the condition is true, 1 if false.

A conditional statement may be terminated by tab, carriage return,],), or comma. A conditional value may be used as a syllable; in this case the conditional must be terminated by a slash. For example:

```
LDA A1 L →|ØIF VP X/+3
```

is equivalent to

```
LDA A1 L →|3 or LDA A1 L →|4
```

while

```
LDA A1 L →|ØIF VP X+3
```

is equivalent to

```
LDA A1 L →|Ø or LDA A1 L →|1
```

ØIF and 1IF are often used to obtain a zero or one as the count of a REPEAT. For example:

```
END: Ø
REPEAT ØIF VP 7776-END, PRINTX /OVERFLOWED CORE/        (1)
```

The address assigned to END is subtracted from 7776. If 7776 is greater, the test is true, and the value of ØIF will be Ø; thus, the count of the REPEAT will be Ø and the message will not be printed.

```
REPEAT 1IF P, EXPUNGE TY0,TY1,ONE          (2)
```

Example (2) will on Pass 2 direct Midas to expunge the listed symbols.

2.14 PRINT, PRINTX, PNTNUM

The psuedo-instructions PRINT, PRINTX, and PNTNUM effect an on-line printout by Midas during assembly. These instructions are particularly useful for obtaining information during the processing of complex macro-instructions. The format of the first two is

```
PRINT TEXT
```

or

```
PRINTX TEXT
```

where the argument may be text of the form used with the pseudo-instruction .ASCII or, if used in a macro-instruction, dummy symbols.

PRINT will cause Midas to type a line in the same format as the first three columns of an error listing (described in the section on error checking). The code PNT is substituted for an error code in the first column and is followed by the argument and a terminal line feed.

PRINTX cause Midas to type only the argument. Since both pseudo-instructions are effective on both passes, a repetitive printout can be avoided only if conditioned, using ØIF or 1IF, with the qualifier P. For example, in response to

```
REPEAT ØIF P, PRINT /TEXT/
```

Midas will print only on Pass 1.

PNTNUM causes the value of an expression to be typed as an octal number. The format is

```
PNTNUM EXPRESSION
```

2.15 STOP

The pseudo-instruction STOP is used when the programmer wishes to arrest the expansion of a macro-instruction, an IRP,

or the range of a REPEAT. In any other context, STOP is ignored by Midas.

When used within a macro or an IRP, STOP will suppress subsequent coding until the occurrence of the next TERMINATE or ENDIRP. Within the range of a REPEAT, STOP will halt the expansion of all subsequent text in the repeat. In addition if the expansion of the REPEAT occurs within a macro-instruction or an IRP, the expansion will be stopped as well.

STOP may be used conditionally as in the following:

```
REPEAT 3, [REPEAT 1IF VZ A-B,STOP
A=A-B]
```

The text A=A-B will appear for processing up to three times. However, if A=2 and B=1 at the start, the count of the inner REPEAT, which generates the STOP, will have the value one before the second appearance, and the expansion of the first REPEAT will be arrested. STOP may also be supplied as an argument for an IRP or a macro call.

2.16 VERNUM

The VERNUM pseudo-instruction has as its value the version number of the text file currently being assembled. It is used to identify which particular version of the program was used to assemble a binary tape. (See Section 4.1)

2.17 BT, BF, RET, STM, MTS, RTM, MTR

These pseudo-instructions are used to generate two byte SUE instructions as described in Sections 1.5.3 and 1.5.5.2. They are terminated by tab, carriage return,],), /, or comma. They may be used as syllables by enclosing them in brackets.

2.18 OTHER PSEUDO-INSTRUCTIONS

The remaining pseudo-instructions--DEFINE, TERMINATE, IRP, IRPC and ENDIRP--are described in the section on macro-instructions.

2.19 SAMPLE PROGRAM SECTION

A page of listing is included in both hexadecimal and octal. It pretends to be neither complete nor correct.

2.19.1 Sample Page of Program - Octal Listing

```
                    1000/
                    BOFI2M=72
001000  070030  LOOP:     LDA A1          PID
001002  001376
001004  040211            JMP X1 I        BASE
001006  001400
001010  044430  I2M:      SUB A1 L        BOFI2M
001012  000110
001014  070051            LDA A2 X1       HOLDI2M
001016  001402
001020  115376            BT ZE,.-2
001022  070151            LDA A6 X1       I2MNXT
001024  001404
001026  070051            LDA A2 X1       PLEASEFLUSH
001030  001406
001032  115024            BT ZE, I2MNOF
001034  040170            JSB A7          FLUSH
001036  001410
001040  070050            LDA A2          NSF
001042  001412
001044  115376            BT ZE,.-2
001046  044641            SUB A2 E 1
001050  030050            STA A2          NSF
001052  001412
001054  110012            BT TR, I2MTRY
001056  030351  I2MNOF:    STA A6 X1 I     ERETQ
001060  001414
001062  030151            STA A6 X1       ERETQ
001064  001414
001066  070051  I2MTRY:    LDA A2 X1       SLT
001070  001416
001072  115726            BT NG, I2MQUT   /HOLDING LINE DEAD
```

2.19.2 Sample Page of Program - Hexadecimal Listing

```
                1ØØØ/
                BOFI2M=72
Ø2ØØ  7Ø18      LOOP:           LDA Al          PID
Ø2Ø2  Ø2FE
Ø2Ø4  4Ø89                      JMP Xl I        BASE
Ø2Ø6  Ø3ØØ
Ø2Ø8  4918      I2M:            SUB Al L        BOFI2M
Ø2ØA  ØØ48
Ø2ØC  7Ø29                      LDA A2 Xl       HOLDI2M
Ø2ØE  Ø3Ø2
Ø21Ø  9AFE                      BT ZE,.-2
Ø212  7Ø69                      LDA A6 Xl       I2MNXT
Ø214  Ø3Ø4
Ø216  7Ø29                      LDA A2 Xl       PLEASEFLUSH
Ø218  Ø3Ø6
Ø21A  9A14                      BT ZE, I2MNOF
Ø21C  4Ø78                      JSB A7          FLUSH
Ø21E  Ø3Ø8
Ø22Ø  7Ø28                      LDA A2          NSF
Ø222  Ø3ØA
Ø224  9AFE                      BT ZE,.-2
Ø226  49Al                      SUB A2 E 1
Ø228  3Ø28                      STA A2          NSF
Ø22A  Ø3ØA
Ø22C  9ØØA                      BT TR, I2MTRY
Ø22E  3ØE9      I2MNOF:         STA A6 Xl I     ERETQ
Ø23Ø  Ø3ØC
Ø232  3Ø69                      STA A6 Xl       ERETQ
Ø234  Ø3ØC
Ø236  7Ø29      I2MTRY:         LDA A2 Xl       SLT
Ø238  Ø3ØE
Ø23A  9BD6                      BT NG, I2MQUT   /HOLDING LINE DEAD
```

3. MACRO-INSTRUCTIONS

A macro-instruction is any legitimate source-language text that a programmer names and sets up so that when the name appears in the subsequent source program, Midas will assemble the text. The text and macro name are established by a macro-definition, whose format is described below. Where the text includes parameters that may differ with each occurrence of the macro-instruction, these parameters may be represented by dummy symbols.

3.1 Macro-Definitions

A macro-definition is initiated by the pseudo-instruction DEFINE, delimited by any terminator. DEFINE is followed by a macro name. A macro name must be a legal symbol, which, if previously defined, will be redefined. The macro name is followed by a list of dummy symbols, if needed, and terminated by a tab or carriage return. If symbols that appear in the text of the macro-instruction are not listed after the macro name, Midas will treat them as ordinary symbols. After a macro name Midas interprets the first character other than space as the first member of the argument list. The argument list is discussed in greater detail later. Midas considers all text following the name and argument line to be the body of the macro-instruction. Midas stores this text until the appearance of the pseudo-instruction TERMINATE, which signals the end of the definition. The body of the macro may include any element of the source language, including other macro-definitions or calls. Any dummy symbol from the list may appear as a syllable in the body of a macro-definition.

3.1.1 Basic Format

The basic format of a macro-definition is illustrated by the following examples.

```
DEFINE
    NEGATE                          (MACRO NAME)
    EOR A1 L →|-1                   (BODY OF THE MACRO)
    ADD A1 E 1
TERMINATE
```

(1)

The macro name, <NEGATE>, subsequently serves as a macro call in the source program. Midas will assemble the body of the macro (<EOR A1 L→-1> and <ADD A1 E 1>) into the object program at each appearance of the macro call.

```
DEFINE

     SUM A, B, C

     LDA A1 →| A                                    (2)

     ADD A1 →| B

     STA A1 →| C

TERMINATE
```

(The character # must be the first character if it is used in a dummy symbol string.)

The macro call <SUM ZORG,ZINC,XMAX> will cause the following sequence to be assembled.

```
     LDA A1 →| ZORG

     ADD A1 →| ZINC

     STA A1 →| #XMAX
```

3.1.2 Dummy Arguments

A programmer may use up to 20. distinct symbols as dummy arguments in a macro-definition as long as each appears in the dummy argument list. Members of the argument list are usually separated from one another by commas. The position of an argument in the list is the model for the order of arguments supplied at a macro call.

Some syllables, although they are referenced only within the body of the macro, will represent a different value at each call. Such syllables may be represented by dummy symbols and specified in the argument list as generated arguments, for which Midas will automatically provide a symbol. A list of those dummy arguments for which Midas must generate symbols is preceded by a slash and follows the list of arguments which

the programmer must supply as shown below:

```
DEFINE MACROSYM A,B/C,D,E
```

or

```
DEFINE MACROSYM /A,B,C
```

where all symbols are to be generated.

Symbols generated and inserted by Midas are of the form <...AØ1>, <...AØ2>, <...AØ3>, etc. If at a macro call the programmer supplies a real argument in a list position corresponding to that of a generated symbol, Midas will accept the supplied symbol rather than generate one. A generated symbol may be used to define variables, address tags, etc.

The following examples give an idea of the use of generated arguments.

1) DEFINITION

```
DEFINE CLEAR A,N/B
        LDA A1 L     →|A
        LDA A2 L     →|N
        STA A1       →|B+2
        LDA A1 E Ø
B,  →|STA A1 X2 D →|Ø
        BF LP,B
TERMINATE
```

CALL: CLEAR TAB,1ØØ

EXPANSION

```
        LDA A1 L     →|TAB
        LDA A2 L     →|1ØØ
        STA A1       →|...AØ1+2
        LDA A1 E Ø
...AØ1→|STA A1 X2 D →|Ø
        BF LP,...AØ1
```

2) DEFINITION

```
DEFINE SAVEAC /A
        STA A1 →|#A
        JSB A1 →|SUBR
        LDA A1 →|A
TERMINATE
```

CALL: SAVEAC

EXPANSION

```
        STA A1 →|#...AØ1
        JSB A1 →|SUBR
        LDA A1 →|...AØ1
```

If an argument is supplied at the call, as in <SAVEAC TEMP>,

```
STA A1→#TEMP

JSB A1→SUBR

LDA A1→TEMP
```

Midas, when scanning the body of a macro-definition for dummy arguments, compares each legal symbol in the text with the symbols in the dummy argument list. Those symbols that correspond to any dummy symbol are stored in a special way, as described in Section 3.3, Storage of Macro-Instructions.

If the programmer wishes to represent only a part of a symbol by a dummy argument, he may use an apostrophe to denote this in the body of the macro definition.

```
<BT FA,.+6>
```

In the pseudo-instruction the string <FA> satisfies the requirements for a legal symbol. During the dummy symbol scan, Midas would interpret <FA> as a single symbol unless an apostrophe is used to indicate that the <A> alone is a dummy symbol, as in

```
DEFINE MACRO A

BT F'A,.+6
```

The apostrophe is deleted when the macro-instruction is defined. In the case of a nested macro-definition, apostrophes are also deleted at the time of definition; that is, when the higher level macro is called.

3.2 Macro Calls

A macro call consists of a macro name followed by a list of arguments separated by commas. The call is terminated by a tab or carriage return.

The arguments of a macro call may include any character string (including an empty string) with the following restrictions. Since comma terminates an argument and tab or carriage

return terminates the list, these may be included only in arguments enclosed by brackets. Brackets must be used in pairs and may be used within other brackets. Midas will consider all but the outermost pair to be part of the argument.

At the appearance of the macro call, Midas processes the body of the macro (stored in the macro table) as though it had appeared in sequence. At this time Midas substitutes for the corresponding dummy arguments and creates the correct number of generated arguments.

If the programmer supplies extra arguments at a macro call and the definition specified generated arguments, the extra supplied arguments will take the place of generated ones. If, however, arguments are supplied in excess of the total number (supplied and generated), the excess arguments are ignored. Note that Midas will not generate a symbol when a programmer fails to supply one that has been specified in the definition.

3.3 Storage of Macro-Instructions

After the occurrence of the DEFINE pseudo-instruction, Midas saves the name of the following macro-definition and scans the list of dummy arguments, keeping count both of the total number of arguments and the number of these arguments that are to be generated. While storing the body of the macro in the macro table, Midas scans the text for dummy symbols. When Midas encounters a symbol that matches a symbol from the dummy argument list, the list position of the corresponding dummy argument is stored in place of the symbol in the text and is distinguished by a code prefix.

Macro-definitions within the body of the macro are stored literally and defined only when the instruction containing them is called.

When Midas encounters the final TERMINATE, the number of words that were required to store the definition is deposited in the first macro-table register preceding the text. The macro name and the table location of the definition are entered in the symbol table.

3.4 Nested Macros

It is convenient when discussing nested macro-instructions to think of DEFINEs and TERMINATEs as if they were parentheses, the outermost pair constituting the highest level macro-definition.

When the programmer calls the highest level macro-definition, Midas stores the second level definition in the macro table, and so on. Internal macro-definitions may contain dummy arguments of higher level ones. These arguments will be replaced by supplied arguments when the higher level definition is called. Pairs of DEFINEs and TERMINATEs must count out. To ensure that they do, the programmer may use the macro name as the argument of a TERMINATE instruction. Then if the DEFINE associated with that TERMINATE refers to another macro name, the error message <MND> (macro name disagrees) will inform the user of a "mispairing" of DEFINEs and TERMINATEs.

A series of examples of nested macros follows. Note in example 1 the use of apostrophe and the insertion of a supplied argument into a nested definition.

EXAMPLE 1

```
DEFINE FLOAT INSTR
  OPSYN OLD'INSTR,INSTR
DEFINE INSTR X
  LDA A1 L  →|X
  JSB A2    →|F'INSTR
TERMINATE INSTR
TERMINATE FLOAT
```

If FLOAT MUL appears, the expansion will be

```
OPSYN OLDMUL,MUL
DEFINE MUL X
  LDA A1 L  →|X
  JSB A2    →|FMUL
TERMINATE MUL
```

This macro-instruction may be used to change Pluribus instructions to subroutine calls. Their original meanings could be restored by

EXAMPLE 2

```
DEFINE UNFLOAT INSTR
   INSTR=OLD'INSTR
TERMINATE
```

```
DEFINE MACRO X,Y
  LDA A1  →| X
DEFINE MAC2 Y
  ADD A1  →| Y
TERMIN
TERMIN
```

The call <MACRO ONE,TWO> will generate

```
  LDA A1  →|ONE
DEFINE MAC2 TWO
  ADD A1  →|TWO
TERMIN
```

The argument supplied for <Y> at the call of MACRO must be a symbol, since it will be inserted as a dummy argument in the definition of MAC2.

EXAMPLE 3. It is usually safer to use rather meaningless symbols as dummy arguments to avoid duplication of real arguments. For example,

```
DEFINE MACRO X
  LDA A1 L  →|X
DEFINE MAC2 COUNT
  ADD A1 L  →|X+3
  STA A1    →| COUNT
TERMIN
TERMIN
```

If COUNT is also a program symbol that the programmer inadvertently supplies at the call of MACRO, the result would be

```
  LDA A1    →|COUNT
DEFINE MAC2 COUNT
  ADD A1 L  →|COUNT+3
  STA A1    →| COUNT
TERMIN
```

EXAMPLE 4. The following example illustrates a macro-instruction that redefines itself when first called.

```
DEFINE INCREM
  LDA A1 E Ø
  STA A1      →|XYZ
DEFINE INCREM
  LDA A1 L    →|1ØØ
  ADD A1 M    →|XYZ
TERMIN
TERMIN
```

At the first call of INCREM the following text is generated:

```
  LDA A1 E Ø
  STA A1      →|XYZ
DEFINE INCREM
  LDA A1 L    →|1ØØ
  ADD A1 M    →|XYZ
TERMIN
```

Subsequent calls will generate

```
  LDA A1 L    →|1ØØ
  ADD A1 M    →|XYZ
```

3.5 The Pseudo-Instructions IRP and IRPC

The pseudo-instruction IRP (indefinite repeat) generates sequential iterations of text a number of times determined by analysis of its arguments. A different set of arguments is substituted at each iteration.

An IRP statement consists of the <IRP> symbol followed by a list of arguments, each enclosed in brackets, terminated by a tab or carriage return. Following the argument list is the body of the IRP. It, like the body of a macro-definition, may include any source language elements, including other IRP's

and macro calls or definitions. The body of an IRP is delimited by the pseudo-instruction ENDIRP.

Each argument of the IRP is itself a list of subarguments separated by commas or carriage returns. The first two members of a subargument list are dummy arguments, and each may be used in the body of the IRP. The remaining members of the list are the "real" arguments of the IRP. Upon encountering an IRP, Midas processes the body of the IRP repeatedly, with different symbolic equivalents substituted for the dummy arguments each time according to the following procedure. Midas begins by substituting the first member of the real argument list for the first dummy symbol and the remainder of the real argument list for the second dummy symbol. The remainder of the real argument list is then treated as the real argument list in subsequent processings, until all lists are exhausted.

IRPC operates exactly as does IRP but on a different type of list. Elements of an IRP subargument list are separated by commas and may include a text string or a bracketed expression. Real arguments of an IRPC subargument list are not separated by commas; each character in the string is treated as an individual list member.

The following examples illustrate the use of IRP and IRPC. Note that the dummy symbols may be omitted if not referenced. although their positions must be indicated by a comma. The second dummy symbol is omitted below:

```
IRP [NAME,,MAPØ,MAP2,MAP1,MAP3],[VALUE,Ø,4,2,6]
    NAME=176ØØØ+VALUE
ENDIRP
```

This IRP will create symbol table entries for the listed map symbols and their corresponding addresses.

The following example shows use of an IRPC within a macro-definition. The digit supplied at the macro call will, during the expansion process, be compared with each digit in the sub-argument list until its equal is found and printed.

```
DEFINE TYPE DIGIT
  IRPC [NUM,,0123456789]
  REPEAT 1IF VZ [DIGIT]-NUM, PRINTX /NUM/ →| STOP
  ENDIRP
TERMINATE
```

The next example illustrates how to use the second dummy symbol, which represents a list.

```
DEFINE MACRO LIST
  IRP [X,Y,LIST]
  REPEAT ØIF D X, [MAC2 [Y]]
  ENDIRP
TERMINATE
```

The list obtained for Y in the first IRP repetition is used as a supplied list for another macro-instruction.

The next example shows a series of nested IRP's used to define a macro-instruction that, given the list <X1,X2,...XN>, will set up a matrix of the form:

```
X1,X2,......XN
X2,X3,...XN,X1
X3,.........X2
.
.
.
XN

DEFINE MATRIX LIST
       LGTH=Ø
  IRP [,,LIST]
       LGTH=LGTH+1
  ENDIRP
  IRP [XØ,LIST2,LIST]
       COUNT=1
       XØ
       IRP [XN,,LIST2]
            COUNT=COUNT+1
            XN
       ENDIRP
  IRP [XØ2,,LIST]
```

```
                    REPEAT 1IF VZ COUNT-LGTH,STOP
                    XØ2
                ENDIRP
            ENDIRP
            TERMINATE
```

The first IRP gets the length of the list. The second gets the next (initially first) member. The third processes the remainder of the list. The fourth goes back to the beginning of the list and takes each element until COUNT = length of text.

4. OPERATION OF THE MIDAS ASSEMBLY SYSTEM

4.1 Preparation of a Source-Language Program

The programmer prepares his source-language program on-line via a Teletype terminal, using TECO, for example, to type in and edit text and to store the program in a file. The file is accessible to Midas by name, version number, and index.*

4.2 Performing an Assembly

4.2.1 Initial Procedure

English files to be processed must be on the PDP-1d's Fastran drum for access by Midas. Midas runs under DDT control and is called in the following way. First, the user types C"F"PMIDAS to start the version of Midas which assembles Pluribus programs. When called, Midas responds with a <#> to signal that it is ready to accept a typed command. The condition of Midas when it is first brought into core is as follows. The current location counter is set to 11Ø and the radix indicator to OCTAL. The macro table is empty. The symbol table contains all pseudo-instructions and a minimal list of Pluribus instructions (these are listed in Appendix B).

4.2.2 The Control Language

A command to Midas is a character string terminated by an Altmode. The first character of the string designates the function to be performed, such as <1> to do Pass 1, and <C> to continue the present pass. Some of the commands require arguments, such as the name and version number of the file to be processed. Arguments, when required, are typed after the command character. Spaces, line feeds, and "control" characters are ignored; and rubout and backslash have their usual meanings.

*Some of this section assumes knowledge of the EXEC III operating system on BBN's PDP-1d computer.

Midas performs each command immediately upon receiving the Altmode terminating the input string. If the command is syntactically incorrect, or if an argument contains a file name and version that is not found in the current index, a question mark is typed and the command is completely ignored.

It is possible to type in a list of commands and have the assembler perform them one after the other without further attention from the user. The command <L> sets the assembler to listen to a series of commands, separated by carriage-returns, and is finally terminated by an Altmode. When typing in commands, rubout rubs out the current line, backslash rubs out the last character, and rubout at the beginning of a line terminates the whole type-in unsuccessfully, causing an error printout. When the Altmode is typed Midas leaves the type-in mode and begins processing the commands that have been entered.

The command characters and argument requirements are listed on the following pages.

Control Characters	Function	Required Arguments
1	Begin Pass 1	Name of English file followed by a comma and the version number
2	Begin Pass 2	Same as for 1
C	Continue present pass on additional file.	Name and version number of additional file, as for 1
I	Initialize symbol and macro tables.	None
E	The argument of E represents a bit setting. Certain bit settings inform Midas to perform a special function during assembly. The bit settings and their associated functions are listed below. Bit 17--continue processing commands after a fatal error. <E1> Bit 16--print all characters processed. <E2> Bit 15--print an error comment on Pass 1 if the location goes indefinite. <E4> Bit 14--define undefined symbols as Ø on Pass 2. <E1Ø>	An octal number

Control Characters	Function	Required Arguments
	Bit 13--suppress DOR errors. <E2Ø>	
J	Add a jump block to assembled binary program. Selects address following last START encountered.	None
H	Halt Midas.	None
X	Connect to an index.	Name of programmer's index
A	List constants areas.	None
L	Listen for a series of commands.	None. Commands are typed after altmode following the <L>
W	Type EOT and wait a specified (octal) number of minutes before proceeding.	Octal number
B	Set up an indexed file of binary output in programmer's index.	Name and version number as required for file access
S	Set up indexed file of symbol table and macro table in programmer's index.	Name and version number as required by general filing
T	Load symbol and macro table into Midas.	Name and version number under which the table is filed

A simple assembly of a symbolic program consisting of a file in the Index IMP would be accomplished by the following sequence of commands.

```
X  IMP

1  PROGRAMX,1

2

J

B

H
```

A more complex example might be —

```
#X IMP                                  Connect to IMP index

#1 PARAMS,412                           Start Pass 1
IMPJOB  4.12  PARAMS - PASS 1           (Underlined part is title line
                                        of file)

#C PART1,413                            Continue (Pass 1)
IMPJOB  4.13  PART 1 - PASS 1

#C PART2,413                            Continue (Pass 1)
IMPJOB  4.13  PART 2 - PASS 1

#C PART3,413                            Continue (Pass 1)
IMPJOB  4.13  PART 3 - PASS 1

#2 PART1,413                            Start Pass 2*
IMPJOB  4.13  PART 1 - PASS 2

#C PART2,413                            Continue (Pass 2)
IMPJOB  4.13  PART 2 - PASS 2

#C PART3,413                            Continue (Pass 2)
IMPJOB  4.13  PART 3 - PASS 2
PNT    5546      FOO                    (Result of the PNT
                                        pseudo-instruction -
                                        see language descrip-
                                        tion.)

#J                                      Terminate binary with a
                                        jump block
```

*The parameters file generates no binary code; therefore it does not require a second pass.

```
#B BIG413,1                          File binary as BIG413,
                                     version 1

#S BIG413,1                          File symbol table

#A                                   Give alphabetic symbol
CONSTANTS AREA, INCLUSIVE            print
FROM        TO
5424        554Ø

#H                                   Halt Midas
```

4.3 Order of Operations

The commands 1, 2, C, and J represent functions that must be performed in a certain order. The other commands represent functions that the programmer may select at various times during an assembly.

5. BINARY OUTPUT FORMAT

All blocks, with the exception of the single-word jump block, begin with two words that indicate position and length and end with a checksum word. The maximum block length is $1\emptyset3_8$ words. The number of data words in a block is derived by subtracting the first word from the second. The checksum word contains the sum modulo $(2^{18}-1)$ of all other words in the block, including the first two.

The first word contains, in addition to the type-indicating bits, the address in core where the first data word is to be stored; the second, the address following storage of the last word in a block.

The pseudo-instruction WORD may be used to fabricate special formats or to insert jump blocks without stopping the assembly. When Midas encounters a WORD pseudo-instruction, it terminates the current block with a checksum. The arguments of WORD are appended directly to the binary output.

6. ERROR CHECKING

If Midas encounters an error in source-language coding, the assembly is interrupted and a descriptive error message printed. Depending on the severity of the error, assembly may or may not continue. The format of an error message is exemplified as follows:

(1)	(2)	(3)	(4)	(5)	(6)
USW	1ØØØ	14ØØØØ	ALPHA+2	REPEAT	GAMMA

Column (1) contains a descriptive error code; (2), the octal address at which the error occurred; column (3), the (non-zero) offset count; (4), the symbolic address stated in terms of the last address tag seen. Column (5) contains the last pseudo-instruction symbol or macro name Midas encountered. Column (6), used only in errors involving symbol definition, contains the offending symbol. Midas omits any column that is not pertinent.

The error codes and the conditions with which they are associated are listed on the following pages, indicating what action Midas takes.

Error Designation	Condition Causing Error	Action on Continuation
Undefined symbols		
USα	Undefined symbol (α indicates where found):	All undefined symbols are evaluated as zero.
B	Branch address	
C	in a constant	
D	in size of dimension array	
F	in OFFSET count	
I	in argument of ØIF or 1IF	
L	in a location assignment	
O	in argument of EQUALS or OPSYN	
P	in a parameter assignment	
R	in the count of a REPEAT	
S	in the argument of a start	
T	in a multi-syllabic address tag	
W	in a storage word	
UWD	undefined symbol in argument of a WORD pseudo-instruction	

Error Designation	Condition Causing Error	Action on Continuation
Undefined symbols		
UBR	Undefined branch condition BT, BF	assumes TR
UDP	Undefined displacement to RET,STM,etc.	assumes Ø
UPA	Undefined PNTNUM argument	no print occurs
UNC	Constant is undefined, as no valid CONSTANTS area was defined	tries anyway
Multiple Definitions		
MDT	Multiply defined tag	Original definition retained
MDV	Multiply defined variable (a symbol previously defined as other than a variable appears with a #)	Original definition retained
MDD	Multiply defined dimension (a previously defined symbol used as an array name)	Original definition retained
Other Errors		
MND	Macro name disagrees (the argument of a TERMINATE disagrees with the name being defined)	First name used

Error Designation	Condition Causing Error	Action on Continuation
Other Errors		
ICH	Illegal character	The character is ignored
ILF	Illegal format	Characters are ignored until the next tab or carriage return
IPA	Improper parameter assignment. (The expression to the right of the equals sign is inadmissible.)	The assignment is ignored
VLD	Variables location disagrees. (The pseudo-instruction VARIABLES has appeared on Pass 2 at a different location than on Pass 1.)	Condition ignored
LGI	Location gone indefinite	If the appropriate bit is set (by Midas control <E>), LGI is printed on Pass 1
DOR	Destination out of range - BT, BF, etc.	tries anyway
PNT	Not an error. Result of PRINT pseudo-instruction	

In the event of the following error conditions, assembly cannot continue.

CLD	Constants location disagrees. The pseudo-instruction CONSTANTS

Error Designation	Condition Causing Error	Action on Continuation
Other Errors cont.		
	has appeared on Pass 2 in a location different from Pass 1. All constants syllables have been assigned incorrect values.	
TMC	Too many constants (The pseudo-instruction CONSTANTS has been used too many times in one program)	
TMP	Too many parameters (The storage reserved for macro-instruction arguments has been exceeded)	
TMV	Too many variables (the pseudo-instruction VARIABLES has been used more than 8 times in one program)	
SCE	Storage capacity exceeded (symbol table or macro table full, or too many constant words used)	
EOF	No START pseudo-instruction	

APPENDIX A. Midas Character Set

A.1 Alphabetic

Letters (A-Z)
Digits (Ø-9)

A.2 Punctuation

Character	Function(s)
, (comma)	a) indicates address tag mod (2^{16}) b) separates elements of a List c) terminates count of a REPEAT
: (colon)	indicates address tag mod (2^{16}•)
=	equates symbol to the left with expression to the right
/ (slash)	a) terminates location assignment b) introduces comment c) introduces list of macro-instruction arguments to be generated d) terminates a conditional
()	enclose a literal
[]	expression enclosed specified for syllable function
#	denotes symbol as a variable
carriage-return-line-feed, tab, and line feed	a) word terminators b) varying meanings according to context
. (period)	a) as first character of a symbol is a letter b) as an entire symbol is a pseudo-instruction giving the value of the location counter c) in a number has the value of the digits to its left taken in base 12 (decimal)

A.2 Punctuation - Cont'd

Character	Functions(s)
! (explanation)	has the value of the letters and digits to its left in base 20 (hexadecimal)

A.3 Combining Operators

Product Operators

"T"	folded integer multiplication
"X"	logical disjunction (exclusive OR)
"U"	logical union (inclusive OR)
"A"	logical intersection (AND)
"Q"	quotient
"R"	remainder

Additive Operators

+ or space	Addition, mod $2^{18.}-1$
- (minus)	Addition of the one's complement

A.4 Illegal

A.4.1 Generally Illegal

break

rubout

\ (backslash)

Altmode

A.4.2 Illegal Except Within a Macro-instruction or an IRP

"

"B"

"C"

%

wru

&

"F"

"G"

"H"

*

;

<

>

?

@

↑ (or ^ (hat))

← (or - (underscore))

vert. tab

"K"

"N"

"O"

A.4.2 Cont'd

"P"

"S"

"V"

"W"

"Y"

"Z"

A.5 Ignored Except Within a Macro-instruction or an IRP

$

'

EOT

formfeed

carriage return

APPENDIX B. Symbols in Permanent Midas Vocabulary

B.1 Pluribus Instruction Symbols

B.1.1 Symbols Associated with Memory Reference Instructions

```
LDA = 7ØØ1Ø   (7ØØ8!)
SUB = 7Ø41Ø   (71Ø8!)
AND = 7141Ø   (73Ø8!)
EOR = 7241Ø   (75Ø8!)
TST = 7341Ø   (77Ø8!)
STA = 3ØØ1Ø   (3ØØ8!)
ADD = 71Ø1Ø   (72Ø8!)
IOR = 72Ø1Ø   (74Ø8!)
CMP = 73Ø1Ø   (76Ø8!)
  W = Ø
  M = -4ØØØØ  (-4ØØØ!)
  E = -2361Ø  (-2788!)
  X = -1Ø     (-8!)
  B = 4ØØØ    (8ØØ!)
  A = -1ØØØØ  (-1ØØØ!)
  L = -24ØØØ  (-28ØØ!)
  R = -24Ø1Ø  (-28Ø8!)
  I = 2ØØ     (8Ø!)
  D = -2ØØØØ  (-2ØØØ!)
FCL = 3Ø1Ø    (6Ø8!)
```

B.1.2 Symbols Associated with Branch Condition, Shifts and Jumps

TR = Ø

EQ = 4ØØ (1ØØ!)

GT = 1ØØØ (2ØØ!)

OV = 14ØØ (3ØØ!)

CY = 2ØØØ (4ØØ!)

F1 = 24ØØ (5ØØ!)

F2 = 3ØØØ (6ØØ!)

F3 = 34ØØ (7ØØ!)

LP = 4ØØØ (8ØØ!)

OD = 44ØØ (9ØØ!)

ZE = 5ØØØ (AØØ!)

NG = 54ØØ (BØØ!)

LT = 6ØØØ (CØØ!)

BF = 1ØØØØØ (8ØØØ!) / effected with a pseudo-instruction

BT = 11ØØØØ (9ØØØ!) / effected with a pseudo-instruction

LI = 12Ø2ØØ (AØ8Ø!)

RI = 1222ØØ (A48Ø!)

LX = 12ØØØØ (AØØØ!)

RX = 122ØØØ (A4ØØ!)

AO = Ø

LL = 4ØØ (1ØØ!)

LO = 1ØØØ (2ØØ!)

LC = 14ØØ (3ØØ!)

Symbols Associated with Branch Conditions, Shifts and Jumps - Cont'd

```
JMP = 4ØØ1Ø  (4ØØ8!)
JSB = 4ØØ1Ø  (4ØØ8!)
NOP = 1ØØØØØ (8ØØØ!)
```

B.1.3 Symbols Associated with Control (Class Ø) Instructions

```
HLT = Ø
RST = 1ØØØ   (2ØØ!)
SST = 12ØØ   (28Ø!)
ENB = 4ØØØ   (8ØØ!)
INH = 42ØØ   (88Ø!)
ENW = 41ØØ   (84Ø!)
INW = 43ØØ   (8CØ!)
KEY = 4Ø2Ø   (81Ø!)
SKEY = 4Ø2Ø  (81Ø!)
RET = 2ØØØ   (4ØØ!) /effected with a pseudo-instruction
STM =  4ØØ   (1ØØ!) /effected with a pseudo-instruction
MTS = 24ØØ   (5ØØ!) /effected with a pseudo-instruction
MTR = 34ØØ   (7ØØ!) /effected with a pseudo-instruction
RTM   14ØØ   (3ØØ!) /effected with a pseudo-instruction
```

B.1.4 Symbols Associated with Register Selection

```
XØ = Ø
X1 = 1
X2 = 2
X3 = 3
X4 = 4
X5 = 5
X6 = 6
X7 = 7

AØ = Ø
A1 = 2Ø      (1Ø!)
A2 = 4Ø      (2Ø!)
A3 = 6Ø      (3Ø!)
A4 = 1ØØ     (4Ø!)
A5 = 12Ø     (5Ø!)
A6 = 14Ø     (6Ø!)
A7 = 16Ø     (7Ø!)
```

B.1.5 Symbols Associated with the Scientific Instruction Set

```
RBIT = 4Ø4ØØ (41ØØ!)
SBIT = 41ØØØ (42ØØ!)
CBIT = 414ØØ (43ØØ!)
```

Symbols Associated with the Scientific Instruction Set (cont'd)

```
IBIT =  42ØØØ   (44ØØ!)
TSBT =  424ØØ   (45ØØ!)
TBIT =  43ØØØ   (46ØØ')
MOVT =  434ØØ   (47ØØ!)
MOVO =  4341Ø   (47Ø8!)
MOVP =  436ØØ   (478Ø!)
MOVN =  4361Ø   (4788!)
SLAN =  12ØØ1Ø  (AØØ8!)
SLLN =  121Ø1Ø  (A2Ø8!)
SRAN =  122Ø1Ø  (A4Ø8!)
SRLN =  123Ø1Ø  (A6Ø8!)
DLAN =  124Ø1Ø  (A8Ø8!)
DLLN =  125Ø1Ø  (AAØ8!)
DRAN =  126Ø1Ø  (ACØ8!)
DRLN =  127Ø1Ø  (AEØ8!)
 DRX =  126ØØØ  (ACØØ!)
 DRI =  1262ØØ  (AC8Ø!)
 DLX =  124ØØØ  (A8ØØ!)
 DLI =  1242ØØ  (A88Ø!)
 MUL =  13141Ø  (B3Ø8!)
```

Symbols Associated with the Scientific Instruction Set (cont'd)

```
 DIV =  132Ø1Ø  (B4Ø8!)
DLDA =  13241Ø  (B5Ø8!)
DSTA =  13ØØ1Ø  (BØØ8!)
DADD =  131Ø1Ø  (B2Ø8!)
DSUB =  13Ø41Ø  (B1Ø8!)
JKEY =  4Ø3Ø     (818!)
LCPU =  4Ø4Ø     (82Ø!)
LKEY =  4Ø6Ø     (83Ø!)
  EE =  2ØØ      (8Ø!)
  EL =  1Ø        (8!)
  ER =  Ø
```

B.1.6 SUE to Pluribus Instruction Equivalence

```
BRUN =  BT  TR,
DSBL =  INH
DSBW =  INW
HALT =  HLT
JUMP =  JMP
JSBR =  JSB
MREG =  MTR
```

SUE to Pluribus Instruction Equivalence
(cont'd)

MSTS = MTS

NOPR = NOP

REGM = RTM

RETN = RET

RSTS = RST

SETS = SST

ENBL = ENB

ENBW = ENW

STSM = STM

B.2 Pseudo-Instructions

Symbol	Function
BF and BT	compile branch instructions
CONSTANTS	specifies storage areas for constant words
DECIMAL	classifies integers as decimal numbers
DEFINE	initiates macro-definition
DIMENSION	allocates storage area for arrays
ENDIRP	end an indefinite repeat
EQUALS	establishes symbol equivalence
EXPUNGE	erases symbols from symbol table
IRP and IRPC	initiates indefinite repeat
MTR	
MTS	
NULL	no operation
OCTAL	classifies integers as octal numbers
OFFSET	assigns address tags as current location counter and sets an expression whose value is the offset count
OPSYN	same as EQUALS; Pass 1 only
PNTNUM	prints its argument's value in octal during assembly

B.2 Pseudo-Instructions (cont'd)

PRINT	generates symbolic location print-out and prints comment during assembly
PRINTX	prints comment during assembly
REPEAT	generates iterative source-language text
RET	
RTM	
START	denotes end of source program and specifies starting address
STM	
STOP	ends expansion of IRP's, macro's and REPEAT's
TERMINATE	ends macro-definition
UNCON	has the value of the number of unused constants in the previous constants area (Pass two only)
VARIABLES	reserves space for variables and arrays
VERNUM	has the version number of the English input file being processed
WORD	appends word(s) to binary output block
ØIF	tests an expression; if true, value if zero; if false, one

Pseudo-Instructions (cont'd)

lIF	if true, value is one; if false, zero
. (period)	has the value of the current location; is "undefined" if the location or offset is indefinite
.ASCII	inserts ASCII code for character string

APPENDIX C. Teletype Code Conversion ("X" means control-X)

C.1 Characters with 6-bit internal representation

ASCII	CHARACTER
Ø4Ø	SPACE
Ø41	!
Ø42	"
Ø43	#
Ø44	$
Ø45	%
Ø46	&
Ø47	'
Ø5Ø	(
Ø51	)
Ø52	*
Ø53	+
Ø54	,
Ø55	-
Ø56	.
Ø57	/
Ø6Ø	Ø
Ø61	1
Ø62	2
Ø63	3
Ø64	4
Ø65	5
Ø66	6
Ø67	7
Ø7Ø	8
Ø71	9
Ø72	:
Ø73	;
Ø74	<
Ø75	=
Ø76	>
Ø77	?

APPENDIX C - Cont'd

ASCII	CHARACTER
1ØØ	@
1Ø1	A
1Ø2	B
1Ø3	C
1Ø4	D
1Ø5	E
1Ø6	F
1Ø7	G
11Ø	H
111	I
112	J
113	K
114	L
115	M
116	N
117	O (OH)
12Ø	P
121	Q
122	R
123	S
124	T
125	U
126	V
127	W
13Ø	X
131	Y
132	Z
133	[
175,176,Ø33	EOM
135	]
Ø15-Ø12	CARRIAGE RETURN-LINE FEED

C.2 Characters with 12-bit Internal Representation

ASCII	CHARACTER
ØØØ	NULL or BREAK or "@"
ØØ1	"A"
ØØ2	"B"
ØØ3	"C"
ØØ4	EOT
ØØ5	"E" or WRU
ØØ6	"F" or RU
ØØ7	"G" or BELL
Ø1Ø	"H"
Ø11	TAB
Ø12	LINE FEED
Ø13	"K" or VT
Ø14	"L" or FORM FEED
Ø15	CARRIAGE RETURN (Output Only)
Ø16	"N"
Ø17	"O"
Ø2Ø	"P"
Ø21	"Q"
Ø22	"R" or TAPE
Ø23	"S" or RDR OFF
Ø24	"T"
Ø25	"U"
Ø26	"V"
Ø27	"W"
Ø3Ø	"X"
Ø31	"Y"
Ø32	"Z"
Ø33	"["
Ø34	SHIFT "L"
Ø35	"]"
Ø36	"↑"
Ø37	"←"
134	BACKSLASH
136	↑ (or hat)
137	← (or underscore)
177	RUBOUT

PLURIBUS DOCUMENT 5: ADVANCED SOFTWARE

PART 3: PLURIBUS ASSEMBLY LANGUAGE AND OPERATING PROCEDURES (PDP-10 TENEX Cross Assembler Version)

The Pluribus Assembler executes under TENEX and assembles code for the processors of the BBN Pluribus multiprocessor. This program is a modified version of the PAL11X assembler which was authored by L. McGowan and submitted as DECUS 10-31. It has subsequently been rewritten by M.I.T. Project MAC, and modified by BBN. The present modification was made by C.R.Morgan at BBN.

PDP-10 Assembler

Update History:
Originally written by C.R.Morgan - December 1974

PREFACE

This document describes the Pluribus Assembly Language and how a source program written in this language can be assembled on TENEX and loaded and run on a BBN Pluribus multiprocessor.

Part 1 contains a comprehensive description of the Pluribus language. The processors for the Pluribus are Lockheed SUE processors. It is assumed that the reader is familiar with the contents of the SUE Computer Handbook.

Part 2 describes the complete operation of assembling a Pluribus source program on TENEX. The assembled program can then be punched on paper tape and run on a Pluribus. It is assumed that the reader is familiar with TENEX.

The following terms and symbols are used throughout this document.

Term or Symbol	Meaning	7-bit Octal ASCII Code
byte	An 8-bit quantity	
word	A 16-bit quantity	
	Blank or space	040
tab	Horizontal tab	011
	Line feed	012
FF	Form feed	014
'	Apostrophe	047
"	Quote	042
:	Colon	072
;	Semicolon	073
R0 - R7	Registers 0 - 7	
PC	Program counter, register 0	

CONTENTS

		Page
PART 1.	THE PLURIBUS ASSEMBLY LANGUAGE	1
1.1	A SOURCE STATEMENT	1
1.1.1	Labels	1
1.1.2	Operators	2
1.1.3	Operands	3
1.1.4	Comments	3
1.1.5	Format Control	3
1.2	SYMBOLS	5
1.2.1	Permanent Symbols	5
1.2.2	User-Defined Symbols	5
1.2.3	Direct Assignment	6
1.2.4	Register Symbols	7
1.3	EXPRESSIONS	9
1.3.1	Expression Operators	9
1.3.2	Numbers	10
1.3.3	ASCII Text Generation	11
1.4	THE LOCATION COUNTER	12
1.4.1	Reserving Blocks of Storage	12
1.5	ADDRESSING MODES	14
1.5.1	Control Class with 8 Bit Field	14
1.5.2	Control Class with 4 Bit Field	15
1.5.3	Control Class with Address	15
1.5.4	Rotate and Shifts	15
1.5.5	Branch Instructions	16
1.5.6	General Class Instructions	17
1.5.7	Subroutine Call Instruction	17
1.5.8	Jump	17

		Page
1.6	ASSEMBLER DIRECTIVES	18
1.6.1	.BYTE	18
1.6.2	.WORD	19
1.6.3	.BLKB and .BLKW	20
1.6.4	.ASCII and .ASCIZ	20
1.6.5	.EVEN and .ODD	21
1.6.6	.END	22
1.6.7	.EOT	22
1.6.8	.TITLE and .STITL	22
1.6.9	.RAD50	23
1.6.10	.IF	24
1.6.11	.IIF and .LIF	26
1.6.12	Special Listing and Output Actions	27
1.6.13	.OFFSET	28
1.6.14	.INSRT	28
1.6.15	.REPT and .IRP and .IRPC	28
1.6.16	.EXTRN	29
1.6.17	.ENTRY	30
1.6.18	.RADIX	31
1.7	MACROS	32
1.7.1	Defining a Macro	32
1.7.2	Calling a Macro	32
1.7.3	Concatenation	32
1.7.4	MACRO-Argument Scan	33
1.7.5	.MEXIT	34
1.7.6	.TTYMA	34
1.8	VALUE-RETURNING PSEUDO-OPS	35
1.8.1	.ADRMD	35
1.8.2	.LENGTH	35
1.8.3	.FIRST	35
1.8.4	.ADDRE	35
1.9	SPECIAL SYMBOLS	36
1.9.1	Pre-Defined Symbols	37
1.10	RELOCATION	40

Page

PART 2. OPERATING PROCEDURES 42

2.1 RUNNING THE ASSEMBLER 42
2.1.1 Initial Dialogue 42
2 1.1.1 Command String 42
2.1.1.2 Command Files 44
2.1.2 Closing Dialogue 44
2.1.3 Cross Reference Listing 45

2.2 OUTPUT 46
2.2.1 Listing Format 46
2.2.1.1 Binary Output 47
2.2.1.2 Loading Programs 48
2.2.2 Error Messages 48

2.3 CHARACTER SET 49

APPENDICES

A Special Characters 50

B General Class Address Mode Syntax 53

C Instructions 55

D Assembler Directives 58

E Initial Symbol Table 63

F Lockheed SUE Opcode Equivalents 68

G Additions and Limitations 70

THE PLURIBUS ASSEMBLY LANGUAGE

The Pluribus Assembly Language, described herein, is a machine language for the BBN Pluribus multiprocessor. A source program written in Pluribus can, however, be assembled on TENEX as explained in Part 2 of this document.

1.1 A SOURCE STATEMENT

A source program consists of a series of source statements. Each statement is terminated by either a carriage-return/line-feed or carriage-return/form-feed sequence. A source statement may contain only printable characters (ASCII* values 040 - 175 inclusively) plus the blank, tab, carriage return, line-feed and form-feed characters. Null (000) and rubout (177) characters are ignored by the assembler. Lower-case alphabetics (141 - 172) are converted to upper-case except in .ASCII and .ASCIZ statements, and after " or '.

A source statement may have up to four fields. These fields, if present, must appear in the following order:

label operator operands comments

Each field is defined primarily by its order of appearance within the source statement, and secondarily by a specific delimiting or terminating character (see Appendix A).

1.1.1 Labels

A label defines a symbolic address in the program being assembled, and the assembler equates that label with the current value and relocation of the assembler's location counter at the point where the label is encountered. Thus, a symbol defined in a label field may be used to refer to the address of the associated memory location.

A storage word is never generated for a label. (Binary output in symbol table is generated.)

The following rules apply to the label field of a source statement:

a. A label must be terminated by a colon(:). (If the label is terminated by two colons then it is said to be "half-killed", or suppressed - i.e. not available for DDT typeout, as with == in MACRO.)

- - - - - - - - - - -

*ASCII stands for American Standard Code for Information Interchange.

b. A label, if present, must be encountered within the first field of a source statement.

c. A source statement may contain no labels, one, or multiple labels. If multiple labels appear, each label is defined as being equivalent to the current value and relocation of the location counter.

d. Blanks and/or tabs may precede a label or follow a label, even to the extent of separating the label from the label terminator (:).

e. The rules for the formation of a label are as specified for symbols; see section 1.2.2.

f. Therefore, embedded blanks and/or tabs within a label are not permitted. e.g. AB CD:

Example:

If the location counter currently contains 100 , the source statement

```
ABCD:    LDA     S,N
```

will equate the label ABCD with the address 100 .

Under the same conditions, the statement

```
ABC:     DDD:     $777:     LDA A,B
```

will equate each of the three labels ABC, DDD, and $777 with address 100 .

1.1.2 Operators

The operator field normally contains a mnemonic belonging to any one of the following classes:

a. A machine instruction mnemonic contained in the permanent symbol table.

Example:	label	operator	operands
	ABCD:	LDA	X,Y

b. An assembler directive (see section 1.6).

Example:	label	operator	operands

```
ABCD:           .WORD       10
```

All instruction mnemonics are listed in Appendix C and assembler directives are listed in Appendix D.

The following rules apply to operators:

a. The operator may be preceded by a label.

b. Leading blanks and/or tabs in the operator field are ignored.

c. The operator is terminated by a space, tab, or any delimiter which can start the operand field, if no other legal terminator exists. If there are no operands, the terminator may be CR/LF, CR/FF or semicolon. All terminators are listed in Appendix A.

1.1.3 Operands

The contents of the operand field are dependent on the contents of the operator field.

1. The operand field must contain only symbols, expressions, address specifications, or data.

2. Multiple operands are separated by commas. Blanks and/or tabs adjacent to terminators in the operand field are ignored.

3. The operand field is terminated by either a semicolon (indicating the beginning of the comment field) or a source statement terminator (carriage return/line-feed or carriage return/form feed characters).

1.1.4 Comments

The comments field is optional. If it is present, it must be preceded by a semicolon character, contain any valid ASCII characters, and terminate with a source statement terminator.

Example

```
Label       Operator       Operand       Comment
LABEL:          LDA            A,B       ;THIS IS A COMMENT.<CR/LF>
```

1.1.5 Format Control

Formatting of the source program is controlled by the space and tab characters. They have no effect on the assembling process of the source program unless they are embedded within a symbol, number,

or ASCII text; or are used as the operator field terminator. Thus, they can be used to provide a neat, readable program. A statement can be written:

```
LABEL:LDA A,TAG;GET TAG
```

or, using formatting characters, it can be written:

```
LABEL: LDA A,TAG     ;GET TAG
```

which is much easier to read.

Page size of the assembly listing is controlled by the form feed character. A page of n lines is created by inserting a form feed (CTRL/FORM keys on the keyboard) after the nth line. If no form feed is present, a page is terminated after 56 lines. The number of lines on a page can be changed by the /V option in the command line to the assembler.

1.2 SYMBOLS

A symbol represents a numerical quantity or memory address.

1. A symbol defined by a label represents a memory address.
2. A symbol defined by a direct assignment statement may represent either a numerical constant or a memory address.

The assembler recognizes two categories of symbols, permanent and user-defined. Permanent symbols are the operator mnemonics and assembler directive names (LDA, ADD, JSB, .ASCII, etc.). These symbols are a permanent part of the assembler's symbol table and need not be defined by the user (see Appendix E). User-defined symbols are those symbols defined by the user via labels or direct assignment statements. These symbols are added to the symbol table as they are encountered during the assembly process. A user-defined symbol may be the same as a permanent symbol. In this case, the selection of value for the duplicated symbol is as described below.

1.2.1 Permanent Symbols

The value associated with a permanent symbol is dependent upon its usage.

1. A permanent symbol encountered in the operator field is associated with its corresponding machine op code.

2. If a permanent symbol in the operand field is also user-defined, then the user's value is associated with the symbol. If the symbol is not found to be user-defined, then the corresponding permanent symbol value is associated with the symbol.

1.2.2 User-Defined Symbols

User symbols are defined by using them as labels or in direct assignment statements.

The rules for the formation of a symbol are (same for predefined):

1. A symbol is composed of from one to six characters. A symbol can be longer than six characters, but only the first six characters are considered by the assembler.

2. The first character within a symbol must be either a letter A through Z, or a letter a through z, or one of the characters $. or %.

3. The remaining characters in a symbol may be upper or lower case letters, digits, or the characters $. %.

Caution should be taken when using the $ and . since they are in system reserved names (e.g., .ASCII, .END). Lower and upper case letters are equivalents (e.g. table and TABLE are equivalent).

Example

Valid Symbols would be:

A

$34

SYS...

Invalid Symbols would be:

6$	First character must be A-Z
AB CD	Embedded blank
AA(I)	() are invalid characters
TABLE[1	[is an invalid character

1.2.3 Direct Assignment

A direct assignment statement associates a symbol with a value, relocation, externalness (see Relocation 1.12) and register-ness (see 1.2.4). When a direct assignment statement introduces a symbol for the first time, that symbol is added to the assembler's symbol table and the specified value, relocation and externalness is associated with that symbol. When a direct assignment statement specifies a symbol that has previously been introduced, then the specified value, relocation and externalness replaces the previous value associated with the symbol except if a permanent symbol, in which case both are retained. The general format for a direct assignment statement is:

Symbol = Expression

1. An equal sign (=) must separate the symbol from the expression defining the value to be associated with the symbol. (If two equal signs separate the symbol from the defining expression, the symbol is said to be "half-killed" or suppressed, i.e. not available for DDT typeout, as in MACRO).

2. A direct assignment statement may be preceded by a label and may be followed by comments.

3. Only one symbol can be defined within a direct assignment statement.

4. The symbol itself must conform to the rules given under "Symbols" (section 1.2.2).

5. Blanks and/or tabs are ignored.

6. Only one level of forward referencing is allowed.

   Example

```
X=Y          ;Unresolved on pass 2
Y=Z          ;Resolved on pass 2
Z=1          ;Resolved on pass 1
```

   Since X will still be undefined at the end of pass 2 of the assembler, all references to X during pass 2 will be flagged as an error and all references to Y before it on pass 2 will also be flagged as an error.

A storage word is never allocated for a direct assignment statement.

Examples

```
A = 1                ;The symbol A is equated with
                     ;the value 1
B = 'A&MASKLOW       ;The symbol B is equated with the value
                     ;of the expression (see section 1.3)
C: D = 3             ;The labels C and E are equated
                     ;with the numerical memory address
E: LDA A,D           ;where the LDA command will be
                     ;stored.  The symbol D is equated
                     ;with 3.
```

1.2.4 Register Symbols

The eight general registers of the processor are numbered 0 through 7. These registers may be referenced by use of a register symbol; that is, a symbolic name for a register. A register symbol is one of the symbols %0 through %7 or is defined by means of a direct assignment, where the defining expression contains at least one of the symbols %0 through %7 or at least one term previously defined as a register symbol. In addition, the defining expression of a register symbol must be absolute. For example:

```
R0=%0       ;DEFINE R0 AS REGISTER 0

R3=R0+3     ;DEFINE R3 AS REGISTER 3

R4=1+%3     ;DEFINE R4 AS REGISTER 4

THERE=%2
```

It is important to note that all register symbols must be defined before they are referenced. A forward reference to a register symbol will generally cause phase errors (Section 2.2.3).

The % may be used in any expression, thereby indicating a reference to a register. Such an expression is a register expression. Thus, the statement:

```
LDA     %6,=0
```

will clear register 6 while the statement:

```
LDA     6,=0
```

is illegal and will generate an error message.

1.3 EXPRESSIONS

An expression is a combination of terms joined together by a specified set of operators. Expressions are evaluated using 16-bit quantities and 16-bit operators, and produce a 16-bit result. (See Section 1.12 on relocation for rules regarding combination of relocatable and external values.)

A term may be any of the following:

a. A symbol, permanent or user-defined (see section 1.2).
b. A number (see section 1.3.2).
c. ASCII text (see section 1.3.3).
d. A special symbol (see section 1.8).
e. A register symbol (see section 1.2.4).

1.3.1 Operators

An operator may be any of the following:

a. + Arithmetic addition or unary plus
b. - Arithmetic subtraction or unary minus
c. ½ Logical OR
d. ? Logical exclusive OR
e. & Logical AND
f. _ Logical SHIFT
g. * Signed Multiplication
h. / Signed Division
i. \ Remainder

The evaluation of an expression proceeds from left to right. All operaters have equal precedence. Grouping is allowed using "<" and ">" as parentheses.

Examples

```
STA %1,A+2
```

The contents of register 1 will be moved to the location whose address is A+2. (See sect. 1.2.4 for explanation of %).

```
BR .-2
```

Branch to the location 2 bytes before the current location. (See sect. 1.4 for explanation of .).

```
STA %1,A+<2*140>
```

The contents of register 1 will be moved to the location whose address is A+300 (A+(2*140)).

1.3.2 Numbers

All unsigned numbers are treated as positive values; all negative numbers must be preceded by a unary minus (-) character. Numbers must be composed of a radix specifier followed by digits and letters in the range specified by the radix specifier. The possible radix specifiers are ^D, ^O, ^H, and nothing (literally the character ^ followed by D, O, or H). These specifiers indicate decimal, octal, hexadecimal, or the current radix as specified by the .RADIX assembly directive. All illegal numbers are set to value of zero.

Octal Numbers

All numbers preceded by a ^O are treated as octal numbers. Such a number can be composed only of the digits 0 through 7.

Decimal Numbers

All numbers preceded by a ^D are treated as decimal numbers. Such a number can only be composed of the digits 0 through 9.

Hexadecimal Numbers

All numbers preceded by a ^H are treated as hexadecimal numbers. Such a number can only be composed of the digits 0 through 9 and the letters A through F.

Numbers without specifier

Any number not preceded by a ^D, ^O, or ^H is a number in the current radix specified by the .RADIX assembly directive. This radix is octal at the beginning of each pass of the assembler. The number must begin with a digit 0 through 9. Subsequent digits may be in the range 0 through 9 or A through Z depending on the radix being used. Only those characters less than the radix may be used in either the first or subsequent positions of the number. The characters are ordered by 0 being the 0th digit and A being the immediate successor of 9.

Numbers exceeding the storage space provided them will be truncated and an error will be generated.

Example

```
.WORD 10, ^D10, 3, ^HF
```

Will result in octal 10, octal 12, 3, octal 17 being stored in consecutive memory words. (See 1.6.2 for a description of .WORD)

1.3.3 ASCII Text Generation

ASCII text may be generated three ways. Strings of text consisting of any number of characters are generated with the .ASCII or .ASCIZ assembly directive (see section 1.6.4). Single and double ASCII characters may be generated for use as operands with the apostrophe and quote operators. The apostrophe causes the next physically encountered character to be considered as text. The quote causes the next two characters to be text.

Examples (see Sect. 1.5.5 for explanation of #)

```
CMPB    R1,#'M          ;Compare R1 with the octal value
                        ;for the ASCII M character

CMP     R1,#"MN         ;Compare R1 with the octal value
                        ;for the ASCII pair MN.

.BYTE   'A, 'B, 'C, 'D  ;Equivalent to .ASCII /ABCD/

.WORD   "AB, "CD        ;Equivalent to .ASCII /ABCD/

LDA     R1,'A           ;Load location 301
```

1.4 THE LOCATION COUNTER

The Location Counter, referenced symbolically by "." , represents the address of the assembled object code. When used in the operand field of a machine instruction, "." represents the address of the first word of the instruction. When used in the operand field of an assembly directive it represents the address of the current byte or word. The value of the location counter can be relocatable (see below).

Examples

```
A:  BR  .              ;Equivalent to A: BR A

LDA R1,#.              ;Move the address of this LDA
                       ;instruction to register 1.

;Assume the location counter contains 200

 .WORD .,.,.+2         ;Generates 3 words containing
                       ;the values 200, 202, 206.
```

The assembler sets the location counter to relocatable zero before each pass. Throughout a pass, consecutive memory locations are assigned to each byte of object code generated for the user's program. The location counter is updated as the object data is generated.

The value of the location counter, and hence the location where the object data is stored, may be changed by a direct assignment of the form

```
. = Expression
```

This sets the location counter to the value and relocatability of the expression. The expression must contain no forward reference, undefined symbols, or external references. Any such statement will be flagged as an error and the statement will be printed on the command device (usually on pass 1).

1.4.1 Reserving Blocks of Storage

Blocks of storage for data areas or buffers may be reserved by altering the location counter. For example, if 100 bytes of storage are required, the location counter should be advanced 100 locations from its current location.

Examples

```
        .=1000          ;Set the location counter to 1000

BUFA:   .=.+100         ;BUFA is assigned the value 1000
                        ;and then 100 bytes are reserved.
                        ; . now represents 1100.

BUFB:   .=BUFB+10
                        ;BUFB is assigned the value 1100
                        ;and 10 bytes are reserved.
```

Assembler directives are also available for reserving storage. (See section 1.6.3).

1.5 ADDRESSING MODES

Instructions may have no, one, or two operand fields specifying operands, operand addresses, or special fields within the instruction itself. All allowable operand fields will be defined in this section. The legal usage of these fields will be specified in the following sections.

Definitions and Conventions

a. Let E be a Simple expression as defined in section 1.3.

b. Let R be a Register expression. This is a simple expression containing a term preceded by a % symbol or a term previously equated to a register expression (see section 1.2.3).

   Examples

```
R0=%0           ;Register 0
R1=R0+1         ;Register 1
R2=1+%1         ;Register 2
```

c. Let ER be a register expression or a simple expression in the range 0 to 7.

The processor instruction set is divided into 8 groups of instructions:

1. Control class instructions with 8 bit field
2. Control class instructions with 4 bit field
3. Control class instructions with absolute or relative addressing
4. Rotate and shift instructions
5. Branch instructions
6. General class instructions
7. Subroutine call instruction
8. Jump instruction

Each of these classes has its own addressing structure. Each class will be discussed in order.

1.5.1 Control Class with 8 Bit Field

The control class instructions which have an eight bit field are the HLT, RST, and SST instructions. Each of these instructions uses the first byte of the instruction for the opcode. The second byte specifies the status bits (in RST and SST) to be changed or a signal to the programmer (in HLT). The instruction format is the opcode followed by one or more spaces or tabs followed by an expression E.

```
        RST     E           ; RESET FLAGS SPECIFIED BY E
```

If the expression is absent 0 is assumed. If the value of the expression requires more than 8 bits, it is truncated to eight bits and an error message is generated.

1.5.2 Control Class with 4 Bit Field

The control class instructions which have a 4 bit field are the KEY, ENB, ENW, INH, and INW instructions. Except for the KEY instruction each of these instructions interrogates or changes the interrupt system of the processor. The instruction format is the opcode followed by one or more spaces or tabs followed by an expression E.

```
        INH     E           ; INHIBIT INTERRUPTS GIVEN BY E
```

If the expression is absent 0 is assumed. If the value of the expression requires more than 4 bits, it is truncated to four bits and an error message is generated.

1.5.3 Control Class with Absolute or Relative Addressing

This group of control class instructions includes RET, STM, MST, MTR, and RTM. Each of these instructions specifies an even address for one of several reasons. The address is encoded into 8 bits plus a flag by the following device. If the address is even and less than (octal) 1000, then bit 11 of the instruction is cleared to indicate absolute addressing and the address is divided by 2 and stored in the low order byte of the instruction. If this fails, the present location is subtracted from the desired address. If the result is even and can be stored as a 9 bit two's complement number then the result is divided by 2 and stored in the low order byte of the instruction. Bit 11 is set to indicate relative addressing.

The form of the instruction is opcode followed by a sequence of spaces or tabs followed by an expression E.

```
        RET     E           ; RETURN FROM INTERRUPT VECTOR E
```

1.5.4 Rotate and Shift Instructions

The rotate and shift instructions are SLA, SRA, RLA, RRA, SLL, SRL, RLL, and RRL. These instructions function to move bits in the registers. The instruction names can be remembered by the following device. Those instructions which move bits off the end of a register and forget them are called shifts and begin with an S. Those instructions which never lose bits are called rotates and begin with an R. The second letter specifies right (R) or left (L). If the carry bit is involved in the instruction the instruction is called arithmetic and ends with an A. If the carry bit is not involved the instruction is called logical and ends with an L. The

rotate and shift instructions allow the programmer to specify either an absolute number of bits to move or to specify a register whose low four bits will indicate the number of bits to move. The assembler will recognize which type of instruction is desired by whether the count expression is a register expression or not and will assemble the correct version of the rotate or shift.

The form of the instruction is opcode followed by a sequence of spaces or tabs followed by a register expression R followed by a comma and an expression E.

```
RRL     R,E         ; ROTATE RIGHT E BITS
```

The register expression R and the comma must be present. If E is absent a 0 is assumed. If the expression E is not a register and requires more than 4 bits it is truncated to four bits and an error message is generated.

1.5.5 Branch Instructions

The branch instructions check some specified condition and transfer either on the truth or falsity of that condition. The address to transfer control to is specified by the low order byte of the instruction. The programmer specifies the instruction as an opcode (indicating condition and whether to branch on truth or falsity) followed by a sequence of spaces or tabs followed by an expression E indicating the address to transfer to.

```
BR      E           ; BRANCH UNCONDITIONALLY TO E
```

The instruction names can be remembered by the following device. Unconditional branch is BR and Unconditional don't branch is NOP. All other branch opcodes begin with a B. If the branch is on the falsity of the condition the B is followed by an N. The opcode is completed by appending the condition name to the end. The condition names are as follows:

C	CARRY
E	EQUALS
F1	FLAG 1
F2	FLAG 2
F3	FLAG 3
G	GREATER THAN
L	LESS THAN
LP	LOOP COMPLETE
M	MINUS
O	ODD
OV	OVERFLOW
Z	ZERO

The address is encoded into the instruction as follows. The current location is subtracted from the address E. If the result is even and can be expressed as a 9 bit two's complement number then the result is divided by two and stored in the low order byte of the instruction. Otherwise, an error message is generated. If the expression E is absent, the current location is assumed for E.

1.5.6 General Class Instructions

The most important group of processor operations is this class. It includes the LDA, ADD, SUB, IOR, EOR, AND, CMP, and TST instructions together with their variations. The programmer specifies the instruction by giving an opcode followed by a sequence of spaces or tabs, followed by a register expression R, followed by an address.

```
LDA     R,ADDR     ; LOAD R WITH CONTENTS OF ADDR
```

The opcodes are specified by giving the root opcode as above with the following characters following if desired. If the instruction is to be a byte instruction a B immediately follows the root opcode. If the direction of the instruction is to memory an M immediately follows the opcode. Thus LDAB is a load byte instruction and ADDBM is add register to byte of memory. The various forms of addressing include all the various forms of addressing allowable on the processor. They are summarized with exact formats and code generated in APPENDIX B.

1.5.7 Subroutine Call Instruction

The subroutine call or JSB instruction has format exactly like the general class instructions. The difference is that only the following addressing forms are allowed:

```
JSB     R,E
JSB     R,E(ER)
JSB     R,(ER)
JSB     R,@E
JSB     R,@E(ER)
JSB     R,@(ER)
```

1.5.8 Jump Instruction

The absolute jump instruction or JMP again has form similar to the general class instructions and the JSB. It uses no register specification and can only use those addressing modes that JSB can. Thus it can have only the following forms:

```
JMP     E
JMP     E(ER)
JMP     (ER)
JMP     @E
JMP     @E(ER)
JMP     @(ER)
```

1.6 ASSEMBLER DIRECTIVES

Assembler directives (sometimes called pseudo-ops) direct the assembly process and may generate data.

Assembler directives may be preceded by a label and followed by a comment. The assembler directive occupies the operator field. Only one directive may be placed in any one statement. One or more operands may occupy the operand field or it may be void -- allowable operands vary from directive to directive.

1.6.1 .BYTE

The .BYTE assembler directive is used to generate bytes of data. The forms of the .BYTE directive are:

```
Label     Operator      Operand

LABEL:    .BYTE         E       ;fills one byte

          .BYTE         E  , E  ;fills consecutive bytes
```

a. The mnemonic .BYTE occupies the operator field. A label field can appear preceding the operator field.

b. The expression must conform to the rules given in Section 1.3, and must be absolute.

c. Multiple expressions (generating multiple bytes) are separated by commas.

d. Only the low-order eight bits of the expression are stored. If the high order byte is not all 0's or all 1's, an error is generated.

Example

```
.BYTE   3,4,5
```

will store 3,4,5 in consecutive bytes.

```
.BYTE   1000,4,5
```

will store 0,4,5 in consecutive bytes and an error is generated for the first byte.

.BYTE , , is equivalent to .BYTE 0,0,0

Note that the statement .BYTE 100000+100000 will generate a byte of all 0's and will not generate an error because

the resulting expression, 000000, fits in one byte.

1.6.2 .WORD

The .WORD assembler directive is used to generate words of data. The forms of the .WORD directive are:

```
Label     Operator   Operand

LABEL:    .WORD      E              ;Fills one word

LABEL:    .WORD      E, E ...       ;Fills consecutive words

LABEL:               E              ;Fills one word

LABEL:               E,E, ...       ;Fills consecutive words
```

1. A label field is permitted.

2. The .WORD directive may optionally appear in the operator field.

3. The expression must conform to the rules given in section 1.3.

4. Multiple expressions are separated by commas.

5. Only the low order 16 bits of the resultant of the expression are stored.

6. If the operator field is absent, the first encountered term (other than the label field) in the first expression must not be a recognizable machine mnemonic or assembler directive unless it is preceded by an expression operator (+ - ± ? &).

7. If the location counter is odd it is rounded up to the next higher even address before storing the data. However, if a label preceded the directive the value of the label will be the original odd value. Thus words of data will be stored at even addresses.

   Example

   ```
   .WORD     500,     1000
   ```

   Will store 500 and 1000 in consecutive words.

```
.WORD   LDA,    3000
```

Will store 40000, 3000 in consecutive words. Since .WORD is present, no leading operator is required before LDA.

```
+LDA, LDA
```

Will store 40000, 40000 in consecutive words. The leading + is required to keep the first LDA from being recognized as an operator.

Note that preceding the first term with & or ± or - is interpreted as follows:

```
   -LDA    ;Equivalent to 0-LDA=140000

   ±LDA    ;Equivalent to 0±LDA=040000

   &LDA    ;Equivalent to 0&LDA=000000
```

1.6.3 .BLKB and .BLKW

Two directives are available which cause uninitialized storage locations to be reserved. They are:

```
.BLKB N     ;reserves N bytes of storage
```

and

```
.BLKW N     ;advances location counter to next
            ;even location, then reserves N words
            ;of storage
```

1.6.4 .ASCII and .ASCIZ

The assembler directive .ASCII and .ASCIZ is used to generate 8-bit ASCII text. The form of this directive is:

```
.ASCII /xxx...x/
```

or

```
.ASCIZ /xxx...x/
```

Both these directives cause successive ASCII characters to be assembled into successive bytes of memory. The difference between the two is that ASCIZ causes an additional zero byte to be assembled beyond the end of the string.

1. The mnemonic .ASCII occupies the operator field. A label field can appear preceding the operator field. A space or tab must follow the .ASCII mnemonic.

2. ASCII text must not include nulls, rubouts, line or form feed.

3. The delimiting character (represented by / in the format above) may be any printable ASCII character except colon, equals or any character appearing within the text. This delimiter must appear both immediately preceding and immediately following the user's text.

4. In the operand field of .ASCII and .ASCIZ, lower case letters are distinguished from upper case letters.

Examples of the use of the .ASCII directives are:

```
A:  .ASCII /HELLO/ ;Stores the ASCII represen-
                   ;tations of H,E,L,L, and O
                   ;into consecutive memory
                   ;bytes.

B:  .ASCII AEND OF JOBA  ;Stores the ASCII represen-
                         ;tations of E,N,D, ,O,F, ,
                         ;J,O, and B into consecu-
                         ;tive memory bytes.
```

Note that the terminating character cannot appear within the text.

Example:

```
B: .ASCII AERROR ON TAPEA
```

("PEA" is ignored and error message given)

1.6.5 .EVEN and .ODD

The .EVEN assembler directive causes the assembler's location counter to be incremented by one if it is odd. If the location counter is already even, the .EVEN directive does nothing.

The .ODD assembler directive causes the assembler's location counter to be incremented by one if it is even. If the location counter is already odd, the .ODD directive does nothing.

1. The .EVEN or .ODD may be preceded by a label or followed by a comment. A label, if present, is assigned the value of the location counter before any modification takes place.

2. Any operand will be treated as a comment.

1.6.6 .END

The form of this directive is:

.END optional start address

The .END assembler directive performs two functions:

1. Indicates the logical end of the source program.

2. Optionally specifies the program's entry point. The absolute loader will transfer control to the entry point after a successful load. If no entry point is specified, the loader will halt.

The rules governing the .END directive are:

1. The mnemonic .END occupies the operator field. A label field may appear preceding the operator field. A space, or tab, must separate .END from the entry point specification.

2. The expression E must not contain any external references and must conform to the rules given in Section 1.3.

Example

```
.END A
```

identifies the starting address of the program as the value of symbol A.

1.6.7 .EOT

The .EOT assembler directive is used to indicate the physical end of the source input medium. For example, it might be used at the end of each strip of paper tape (except the last) when a source program resides on more than one strip of tape. The last strip would terminate with an .END directive.

Example

```
.EOT
```

Any operand will be treated as a comment.

1.6.8 .TITLE and .STITL

The .TITLE directive is used to name the object module. The name assigned is the first symbol following the directive. If there is no .TITLE statement the default name assigned is ".MAIN". If

there is more than one .TITLE directive, only the last one encountered is operative. The title is typed on the terminal and appears on the first line of each header.

.STITL sets the subtitle to the remainder of the line following the directive. The subtitle lists on the second line of each heading. If the .STITL is the first line of a page, it takes effect on that page; if not, it takes effect on the next page.

1.6.9 .RAD50

Systems programs can handle symbols in a specially coded form called RADIX 50 (this form is sometimes referred to as MOD40 or SQUOZE). This form allows successive groups of 3 characters to be packed into successive words; therefore, any 6-character symbol can be held in two words. The form of the directive is:

```
.RAD50 /CCC/
```

The single operand is of the form /CCC/ where the slash (the delimiter) can be any printable character except for "=" and ":". The delimiters enclose the characters to be converted which may be A through Z, 0 through 9, dollar ($), dot (.), percent (%), and space ( ). If there are fewer than 3 characters they are considered to be left justified and trailing spaces are assumed. If there are more than three characters, successive groups of 3 are assembled in successive words.

Examples:

```
.RAD50    /ABC/        ;PACK ABC INTO ONE WORD

.RAD50    /AB/         ;PACK AB (SPACE) INTO ONE WORD

.RAD50    //           ;PACK 3 SPACES INTO ONE WORD
```

The packing algorithm is as follows:

A. Each character is translated into its RADIX 50 equivalent as indicated in the following table:

Character	RADIX 50 Equivalent (octal)
(space)	0
A-Z	1-32
$	33
.	34
%	35
0-9	36-47

B. The RADIX 50 equivalents for characters 1 through 3 (C1,C2,C3) are combined as follows:

RESULT=((C1*50)+C2)*50+C3

NOTE: the character translation for Pluribus RADIX50 is not the same as that for PDP10 RADIX50.

1.6.10 .IF

Conditional assembly directives provide the programmer with the capability to conditionally include or not include portions of his source code in the assembly process. If the condition is met, all statements up to the matching .ENDC are assembled. Otherwise, the statements are ignored until the matching .ENDC is detected. The general form of the conditional directive is:

.IF Condition

The conditions fall into several classes:

1) Arithmetic Conditionals - .IF C EXPRESSION

Condition	True if
Z	Expression equal to Zero
LT	Expression less than Zero
LE	Expression less than or equal to Zero
GT	Expression greater than Zero
GE	Expression greater than or equal to Zero
NZ	Expression not equal to Zero

2) String Comparison - .IF C ARG1, ARG2

Condition	True if
DIF	ARG1 Different from (not equal to) ARG2
IDN	ARG1 Identical to (equal to) ARG2 (ARG1 and ARG2 are MACRO-TYPE args) (See Sect.1.7.4)

3) String Comparison with NULL - .IF C ARG

Condition	True if
B	ARG (MACRO-TYPE ARG) is blank
NB	ARG is not blank

4) Testing Definition - .IF C EXPRESSION

Condition	True if
DF	All symbols in Expression are defined
NDF	Some symbols in Expression are not defined

5) Testing PASS Assembler is in

Condition	True if
P1	Assembler is in pass one
P2	Assembler is in pass two

All conditionals must end with the .ENDC directive. Anything in the operand field of .ENDC is ignored. Nesting is permitted. Labels are permitted on conditional directives, but the scan is purely left to right.

For example:

```
          .IF Z 1
A:        .ENDC
```

A is ignored.

```
A:         .IF Z 1
          .ENDC
```

A is entered in the symbol table.

If an .END is encountered while inside a satisfied conditional, the directive will still be processed normally. If more .ENDC's appear than are required, an error occurs on the extras.

In addition, there are three subconditional assembly directives which may appear only in a conditional. They are

```
.IFF        if conditional is false
.IFT        if conditional is true
.IFTF       assemble whether conditional is true or false
```

.IFF switches the sense of the conditional for the code that follows, so it will be assembled only if the conditional failed. .IFT restores the original sense of the conditional for the code that follows. .IFTF forces assembly of the following code independent of the state of the conditional. This is different from ending the conditional because a .IFT may appear later (but before the .ENDC) and make assembly conditional again.

Example:

```
.IF LT   A - B
       A = B
        .IFF
       A = A+1     ;If B ≤ A, increment A and assemble B
        .WORD B
        .IFT
        .WORD A     ;If B > A, set A to B and assemble it
.ENDC
```

1.6.11 .IIF and .LIF

Two alternative forms of conditional assembly directives are available which cause conditional assembly of a single statement only, and which therefore require no .ENDC terminator. They are the "IIF" group and "LIF" group. The conditions are treated in the same fashion as the "IF" group enumerated above.

The form is:

```
.IIF Condition , Statement
```

The statement is assembled under the condition specified. This is equivalent to:

```
      .IF Condition

           Statement

      .ENDC
```

The second form of single-statement condition assembly is the "LIF" group. The form is:

```
.LIF Condition

     Statement
```

This assembles the statement only if the specified condition is true. If the condition is false, the statement on the next line is listed but not assembled.

1.6.12 Special Listing and Output Actions

There are two assembler directives which control whether or not the following code appears on the output listing. They are:

```
.LIST
```

Decrements the "XLIST" count if it is greater than 0. This count is initially set to 0. Listing action takes place only if the XLIST count is 0. The ".LIST" directive itself is listed only if the XLIST count was 0.

```
.XLIST
```

Increments the XLIST count and thus prevents the following code from being listed.

Another directive causes the listing to skip to a new page:

```
.PAGE
```

Three other assembler directives are used to signal some special condition during assembly.

```
.PRINT /TEXT/
```

When the .PRINT directive is encountered the first character following it which is not a space or tab is located. All source characters following this character up to (not including) the next occurrence of the same character constitues the string to be printed. The string is printed on the terminal and inserted into the error file if the error file is open.

```
.ERROR MSG
.MSG MSG
```

The .ERROR and .MSG directives provide similar features. Each uses the whole line of source text as an error message. This error message is processed exactly like an error message generated by the

assembler. The PC, line, page, and error message are printed on the terminal and inserted in the error file. If the directive is .ERROR the error count is incremented. If the directive is .MSG the error count is not incremented.

Another directive affects the CREF listing produced:

```
.XCREF A, B, C
```

prevents CREF output from being generated for symbols A, B, and C.

1.6.13 .OFFSET

The directive .OFFSET EXP sets the offset to EXP. This causes the sum of this offset and the real location counter to be used for the symbol "." in relative addressing and label definition. The offset is zero at the beginning of an assembly pass. The offset must not be relocatable or external.

1.6.14 .INSRT

The directive .INSRT FILSPC pushes the current source file and/or repeat macro etc. and starts reading from the specified file. After the end of that file, the file or macro containing the INSRT will be resumed starting with the next line.

The characters "/@._" are passed as part of the file name. If the specified file is not found, a non-fatal error occurs and assembly of the file containing the .INSRT continues. %FNAM2 is set to the extension of the .INSRT'ed file and remains set after the file is finished (unless that file has done an .INSRT, of course).

This facility is used, for example, to permit a family of separately-assembled modules to include a common head file of global parameter and macro definitions. Each module merely declares ".INSRT Headfile" to perform this linkage.

1.6.15 .REPT and .IRP and .IRPC

Repeat statements are of the form:

```
.REPT EXP
TEXT
-
-
.ENDR
```

The text enclosed by .REPT and .ENDR may be of any length. The number of times it is repeated is determined by the value of the expression which follows .REPT: that is, 0, 1, or N times depending upon whether EXP is $\leq$ 0, 1, or >1. Repeats can be nested to any

level and enclose or be a part of macros.

Two other forms of repeat are available for iterating over a set of objects or over the characters in a string.

```
.IRP DUMMY,  <A1, A2, A3>
```

repeats all the following text up to the matching .ENDM substituting A1 for DUMMY the first time, A2 for DUMMY the second time, etc. The stuff following the comma is treated as a macro argument, and might be delimited by ¢/ ... / instead.

```
.IRPC DUMMY, STRING
```

repeats the text up to a matching .ENDM once for each character in STRING, each time substituting that character for DUMMY. DUMMY must be a symbol; STRING is read in as a macro argument.

1.6.16 .EXTRN

It is often convenient to reference subroutines or symbols defined in another assembly (a subroutine library, for instance). This can be done using the .EXTRN pseudo-op. The statement

```
.EXTRN SYM1,SYM2,...,SYMN
```

declares that the symbols SYM1,SYM2, etc. are defined in another assembly and hence that their values cannot be known at assembly time, only at load time. Because of this the use of external symbols has the following restrictions:

1) Two external symbols cannot be referenced in the same expression.

2) External symbols cannot be operated on by the operators *,/,_,&,?,\ , and ±.

3) External symbols cannot occur on the right-hand side of the - operator.

4) External symbols cannot be defined elswhere in the assembly.

5) Register symbols cannot be external.

6) External symbols cannot be used where a value must be known (REPT, .= for example) or with the .END pseudo-op.

Aside from the above restrictions external symbols can be used like any other symbols.

Examples:

```
        .EXTRN     TCO,A,B

        X = A + 7
        Y = X - 3          ;Y = A + 4
        z = B

        JSB   R7,TCO  ; TYPE A CHARACTER
```

1.6.17 .ENTRY

The statement

```
.ENTRY  SYM1,SYM2,...,SYMN
```

declares that the symbols SYM1,SYM2, etc. are defined in this assembly and that their values are to be made available to other assemblies (requested via .EXTRN) at loadtime. There are no restrictions on the use of .ENTRY symbols.

Example:

```
        .ENTRY A,B,C,TCO

        A = 100

        B: 123

        C: .ASCIZ "Title:"

        TCO:   LDA     R1,TPS   ;Teletype Ready?

               TST     R1,=TTFLAG

               BGE     TCO      ;No-wait for it.

               STAB    R2,TPB   ;Yes, type character.

               JMP     (R7)     ;Return
```

1.6.18 .RADIX

Numbers which are not preceded by a ^D, ^O, or ^H are converted as numbers in the radix specified by the assembler. This radix is initially octal at the beginning of each pass but can be changed by the directive .RADIX:

```
.RADIX   EXPRESSION
```

The expression must be greater than one and less than thirty-seven. For all numbers without radix specifiers following this directive until the completion of the next .RADIX directive or end of the pass, the number will be converted in the radix specified by the current expression. Such numbers must begin with a digit in the range allowable by this radix and be followed by a digit or letter in the range allowed by this radix. The letter A is the direct successor of the digit 9 in the counting order.

1.7 MACROS

A macro is a string of text which, at any time after its definition, can be invoked by the use of its associated symbol. In addition, varying arguments can be transmitted at invocation time.

1.7.1 Defining a Macro

```
.MACR    LOADGO    VALUE,ADDR
 LDA     %3,VALUE
 LDA     %0,ADDR
.ENDM
```

.MACR and .ENDM are assembly directives, LOADGO is the symbol which will invoke the macro, and VALUE and ADDR are dummy arguments. They serve no purpose other than to indicate where actual arguments are to be substituted.

The directive .MACRO is equivalent to .MACR.

1.7.2 Calling a Macro

```
      LOADGO   #100,#123
```

would generate:

```
      LDA    %3,#100
      LDA    %0,#123
```

1.7.3 Concatenation

Another feature of the Macro facility is concatenation, that is, the ability to join character strings into a single symbol or name. The single quote (or apostrophe) is used in a Macro to denote concatenation, which is used to concatenate a Macro argument with some other character string(s) by delimiting the Macro argument in single quotes. When the concatenation is to take place at one end of a string, only one of the quotes is required.

Example

```
;DEFINE THE MACRO

  .MACR  TEST A,B,C,E

LDA   MEM'A,MEM'B'

LDA'E   MEM'C'A,#C'MEM'B'P

 .ENDM

;CALL
```

```
        TEST X, Y, Z,B

;GENERATES

        LDA  MEMX,MEMY

        LDAB MEMZX,#ZMEMYP

;CALL

         TEST  AG1,AG2,AG3

;GENERATES

         LDA   MEMAG1,MEMAG2

         LDA   MEMAG3AG1,#AG3MEMAG2P
```

Macros and Repeats can be used in any configuration and to any depth of nesting provided they are properly nested.

1.7.4 MACRO-Argument Scan

If the first character read when a macro-arg is expected is ^, the next character is used as the delimiter and all successive characters up to the next appearance of the delimiter go in the macro arg. After that, spaces are skipped, and a comma, CR or ; should follow.

If the first character seen is <, all characters between it and the matching > (not including the <and> themselves) go in the macro arg. After the >, spaces are skipped and a comma, CR or should follow.

If the first character is \, an expression is read in and its value is converted to a string in base 8, which becomes the macro argument. After the expression, spaces are skipped, etc.

Otherwise, all characters up to but not including the first command, CR or ; go in the macro arg, except that trailing spaces and tabs before a ; will be ignored.

If the scan of the argument stopped on a CR or ;, there are no more arguments. Any more arguments wanted will be made null. The CR or ; will remain to be re-read after the expansion of the macro. If the scan stopped on a comma, the next argument's scan will start with the character after the comma.

1.7.5 .MEXIT

A macro is exited upon falling through the .ENDM directive, or by encountering the directive

```
.MEXIT
```

anywhere inside the macro. This directive causes assembly to immediately pop out of the innermost macro call, .REPT, .IRP, or .IRPC.

1.7.6 .TTYMA

A variant of the macro call permits the arguments to be collected from the controlling terminal at assembly time:

```
.TTYMA A,B,C
```

reads a line from the TTY, then defines A,B, and C from that line using the macro argument scanning rules. Within the scope of the .TTYMAC (up to the matching .ENDM) A,B, and C will be replaced by the strings obtained by scanning what was read from the TTY.

1.8 VALUE-RETURNING PSEUDO-OPS

Value-returning pseudo-ops may be used anywhere an ordinary symbol may be used. Such a pseudo will usually skip over any comma following its last argument. If there is anything on the line after the pseudo's last arg, the comma should be used, e.g.

```
FOO==.LENGTH¢/ABCDEF/,+1

  sets FOO to 7
```

1.8.1 .ADRMD

.ADRMD ADDR — A value-returning pseudo-op whose value is the addressing mode of ADDR. The addressing mode of ADDR is the 16 bit quantity which is inclusive or'ed with the basic instruction and accumulator to form the first word of a general class instruction. If you want to follow this construction with an arithmetic operator, put a comma in betwen, otherwise the operator may be included in ADDR.

1.8.2 .LENGTH

.LENGTH String — String is read as a macro argument, and the number of characters in String is returned as a value which can be used in arithmetic expressions. (A comma should be used after String and before any following arithmetic operators.)

1.8.3 .FIRST

.FIRST Statement — The statement is translated into binary. The first word of the statement is returned as the value of .FIRST. No storage is allocated for the statement translated. As many characters as are necessary are used to translate the statement, thus care must be used in including .FIRST as a term in an expression.

1.8.4 .ADDRE

.ADDRE Statement — The statement is translated into binary. The second word of the statement is returned as the value of .ADDRE. No storage is allocated for the statement translated. As many characters as are necessary are used to translate the statement, thus care must be used in including .ADDRE as a term in an expression.

1.9 SPECIAL SYMBOLS

May be used as normal symbols, or set with "=".

.	The location counter (including offset). . always equals %.+%OFFSET. Setting .=A is equivalent to %.=A-%OFFSET
%.	The unoffset location counter (where code will actually be loaded).
%FNAM2	The result of taking the current source file's version number, and turning it into a decimal number, ignoring non-digits.
%NARG	The number of args given to the innermost macro invocation.
%OFFSE	The value of the offset.
%XCREF	Stops CREFfing if not 0.
%XLIST	The value of the XLIST count.
.RPCNT	Normally 0 except in repeats. Then, is 0 the 1st time through, 1 the next, etc.
.IRPCN	Like .RPCNT but for IRPs instead of repeats.
%RADIX	The current radix for numbers without specifiers.

1.9.1 PRE-DEFINED SYMBOLS

The following symbols are pre-defined.

Symbol	Octal Value	Hex. Value	Description
Processor Registers			
%0	0	0	Register 0 (Program Counter)
%1	1	1	Register 1
%2	2	2	Register 2
%3	3	3	Register 3
%4	4	4	Register 4
%5	5	5	Register 5
%6	6	6	Register 6
%7	7	7	Register 7
Status Register, RST, and SST Names			
%A	4000	800	Processor ACTIVE
%C	10	8	CARRY bit
%E	1	1	EQUALS bit
%F1	20	10	Flag 1
%F2	40	20	Flag 2
%F3	100	40	Flag 3
%G	2	2	GREATER bit
%L1	10000	1000	Level 1 interrupt
%L2	20000	2000	Level 2 interrupt
%L3	40000	4000	Level 3 interrupt
%L4	100000	8000	Level 4 interrupt
%LP	200	80	Loop Complete
%N	2000	400	NEGATIVE bit
%O	400	100	ODD bit
%V	4	4	OVERFLOW bit
%Z	1000	200	ZERO bit
ENB, ENW, INH, and INW Names			
.L1	1	1	Level 1 interrupt
.L2	2	2	Level 2 interrupt
.L3	4	4	Level 3 interrupt
.L4	10	8	Level 4 interrupt
Low Core Interrupt Vector Addresses			
%ABRT0	50	28	Abort for processor 0
%ABRT1	70	38	Abort for processor 1
%ABRT2	110	48	Abort for processor 2
%ABRT3	130	58	Abort for processor 3
%ILOP0	40	20	Illegal Instruction for processor 0

%ILOP1	60	30	Illegal instruction for processor 1
%ILOP2	100	40	Illegal Instruction for processor 2
%ILOP3	120	60	Illegal Instruction for processor 3
%LOW	400	100	First location after low core
%LVL1	0	0	Level 1 interrupt
%LVL2	10	8	Level 2 interrupt
%LVL3	20	10	Level 3 interrupt
%LVL4	30	18	Level 4 interrupt

Relative Addresses of Components of Interrupt Vectors

%CURPC	4	4	Storage for interrupted PC
%DEVNO	0	0	Interrupting Device Number
%PSTAT	2	2	Storage for interrupted STATUS
%SERVC	6	6	Address of service routine
%BDADR	0	0	QUIT Bad Address
%BDINS	0	0	Illegal Instruction Bad Address

Addresses of Register Block for each Processor

%CPU0	177400	FF00	Registers for processor 0
%CPU1	177440	FF20	Registers for processor 1
%CPU2	177500	FF40	Registers for processor 2
%CPU3	177540	FF60	Registers for processor 3

Relative Addresses of Registers in Register Block

%REG0	0	0	Register 0
%REG1	2	2	Register 1
%REG2	4	4	Register 2
%REG3	6	6	Register 3
%REG4	10	8	Register 4
%REG5	12	A	Register 5
%REG6	14	C	Register 6
%REG7	16	E	Register 7
%STAT	20	10	Status Register
%INST	22	12	Instruction Register
%LADR	24	14	Address of Last instruction
%CTRL	36	1E	Control Register

High Core Addresses

%AREG1	177600	FF80	Address Register for console 1
%AREG2	177604	FF84	Address Register for console 2
%AREG3	177610	FF88	Address Register for console 3
%AREG4	177614	FF8C	Address Register for console 4
%DREG1	177602	FF82	Data Register for console 1
%DREG2	177606	FF86	Data Register for console 2
%DREG3	177612	FF8A	Data Register for console 3
%DREG4	177616	FF8E	Data Register for console 4

PLURIBUS Registers

%MAP0	176000	FC00	Map Register for Page 0
%MAP1	176002	FC02	Map Register for Page 1
%MAP2	176004	FC04	Map Register for Page 2
%MAP3	176006	FC06	Map Register for Page 3

1.10 RELOCATION

Relocatable programs are assembled as if they were to be loaded at location 0, but in fact they can be loaded anywhere. This is useful for such things as separately assembled subroutine libraries which may be used by many different programs and hence may be loaded into many different places. Because the assembler doesn't know where the program will eventually be loaded, it must mark all symbols which represent addresses as "relocatable". Their actual value is the value the assembler assigned (assuming the program would load at 0) plus the address where the program actually loads. For example, if Y is address 100, and the program loads at 1000, the actual value of Y is 1100. In general the actual value of a symbol A representing address a, is A = a + r, where r is the address where the program will be loaded, called the relocation constant.

When the assembler evaluates expressions containing relocatable values, it keeps track of a relocation count, or the number of times r must be added to the value to yield the final value, as well as the value. For example in the expression A + B + C, the relocation count is 3. (A+B+C = (a+r)+(b+r)+(c+r) = a+b+c+3*r). Expressions not inside parentheses must have a relocation count of 0 or 1. Expressions with a relocation count of zero are said to be absolute.

The following table shows how relocation counts are computed.

OPERANDS	OPERATOR	+	-	*	/
A	A	0	0	0	0
A	R	r2	-r2	a*r2	illegal
R	A	r1	r1	a*r1	r1/a
R	R	r1+r2	r1-r2	illegal	illegal

where A stands for an absolute quantity, R stands for any quantity with a non-zero relocation count, rn is the relocation count of the n'th operand, and the table entries represent the resultant relocation count.

NOTE: The operators ±, &, \ , ?, and _ cannot be used on relocatable values.

EXAMPLES:

(lowercase represents absolute, uppercase represents relocatable)

A+x-y	is	relocatable
2*A-B	is	relocatable
A+B	is	illegal (relocation count is 2)
A-B	is	absolute
A+B+C-<D+E>	is	relocatable

PART 2

OPERATING PROCEDURES

The Pluribus Assembler can be used to assemble Pluribus source programs on TENEX. It provides the user with the power of a time-sharing system for assembly of programs. These may then be punched on paper tape and run on a Pluribus multiprocessor.

The source code may come from any TENEX file.

2.1 RUNNING THE ASSEMBLER

2.1.1 Initial Dialogue

Once the assembler has been started it will respond by typing "*" and wait for the user to provide operating instructions via the appropriate command string. Blanks are not allowed in the command string.

2.1.1.1 COMMAND STRING

The general form of the command string is:

Binary Output, Listing Output, Error Output = Source Input, Source Input, ... Source Input

1. The source files are effectively concatenated and assembled. If the binary spec is empty, no binary file is generated. If the listing spec is empty, a listing file is generated with default names. If no listing is desired, the comma should be omitted as well. The error file is treated similarly.

   An abbreviated form of command string which produces only a binary file is:

   Source Input, Source Input, ... Source Input

   Each input/output designator consists of the group:
   DEV:FILNAM.EXT

   a. DEV is
      DSK for disk
      PTP for high speed punch
      PTR for high speed reader
      DTAn for DECtape n
      TTY for the user terminal
      TTYn for terminal number n

If not specified the device is assumed to be the disk for the binary, listing, and first source files. For successive source files, it is assumed to be the same device as the first source file.

b. FILNAM is the filename of the appropriate file. The name of the first source file is the default filename supplied (when not specified) for the binary, listing, and error files.

c. EXT is the filename extension for that file. If it is not specified, the assumptions are:

```
BINARY OUTPUT:   .BIN
LISTING OUTPUT:  .LST  (.CRF if /C switch is on)
ERROR OUTPUT:    .ERR
SOURCE INPUT:    .PLR
```

2. Special Options.

If special features of the assembler are to be used, then the indicated characters preceded by a slash must be typed in the command string.

a. /B- suppress binary.

b. /C- produce a CREF (cross reference) listing. Must be used in the listing field and it is recommended that no device, file, or extension be specified. The assumed name will be CREF.TMP if none is specified. If only a filename is specified, the extension .CRF will be assumed. Forces /L.

c. /D- give back TTY and run detached. Logout when finished. Forces /E.

d. /E- force error file to be written.

e. /H- hexadecimal error and listing files are generated.

f. /I- double size of symbol table for each /I seen.

g. /L- output listing even if no listing spec. (Useful with abbreviated format).

h. /M- suppress the listing of lines generated by MACROS.

i. /N- suppress error messages to the Teletype. Forces /E.

j. /O- addresses in listing include offset.

k. /T- cause terminal - formatted listing to other listing device, or wide format listing to TTY.

l. /V- the following number is to be used as number of lines per page in the listing.

Command Language Examples

a. *DSK:BETA.BIN, LPT:/N=DSK:BETA.PLR
Assemble file BETA.PLR from the disk. Binary to disk file BETA.BIN, listing on the line printer. Suppress error messages to the console.

b. *BETA,LPT:/N=BETA
Same as example a. above.

c. *,PTP:=DTA3:ALPHA
Assemble file ALPHA (or, ALPHA.PLR) from DECtape No. 3. Output a papertape containing the listing file.

2.1.1.2 Command files

If "@FILSPEC" is placed in a command string, the contents of the file FILSPEC are effectively inserted in the command string, surrounded by spaces.

The character following "@FILSPEC" is lost, and should be a comma unless "@FILSPEC" is at the end of the command string, when the same carriage return that ends the command string will end "@FILSPEC".

Carriage returns and linefeeds in the file are treated as spaces. Control-L or Control-C terminates the file. Command files may refer to other files to a depth limited by the size of the pushdown stack.

Command files default independently to device DSK:. They have no effect on the defaulting of other file names.

The character Control-Q cancels an entire command string, just as a sufficient number of Control-A's would.

2.1.2 Closing Dialogue

When the Assembler has completed a run it will print the number of errors detected and the amount of processor time used. If the previous command line was terminated with a carriage return another * is typed. If another program is to be assembled, the command string may be typed. If the previous command string was terminated

by an ESC or ALT MODE the Assembler automatically returns to the monitor on completion of the assembly.

2.1.3 Cross Reference Listing

If a /C option had been typed for a cross reference listing, the assembler generates a modified listing file. This file contains the listing and indicator marks for uses of all symbols. The file can be converted into a listing with cross references by the following sequence:

1. Type ^C and wait for "@".
2. Type CREF to load CREF program.
3. CREF responds with *.
4. Type the RETURN key if listing is to go on line printer or TTY:= and RETURN key if it is to go to the terminal.
5. When CREF types *, type ^C to return to the Monitor. Refer to the decsystem10 assembly language handbook for more details.

2.2 OUTPUT

The initial dialogue determines which output is generated by the assembler. This usually includes binary output and listing output.

2.2.1 Listing Format

The assembler can generate either of two listing formats. One is designed for a terminal and the other is for a line printer. The assembler assumes the line printer format unless the listing device is specified as a terminal in the command string.

Line Printer Format - Each page is headed by a title line as follows:

PLURIBUS Vnnn Date Time Page n

The version number, Vnnn, is the assembler version. The date is given as XX-YYY-ZZ where XX is the day, YYY is the month and ZZ is the year. The time is of the form XX:YY where XX is the hour and YY is the minute. The page number is a simple number (e.g., PAGE 5) or a simple number trailed by a dash and another number (e.g., PAGE 5-1). The first form is used when the new page is started because of a form feed in the source. The second form indicates that a new page was started because 55 lines were printed without a form feed. The format of the listing consists of, left to right:

	The error flags are printed here.
a. Location Counter	The location counter for the instruction or data is printed as 6 octal digits.
b. Object Fields	Up to 3 fields containing the object code. The first field is the instruction word. Additional words are printed to the right.
c. Source Image	The input source image is printed to the right of the third object field.

Certain statements are printed with slight variations of the above. For example, on a direct assignment only the value of the expression is printed in the first object field with the other fields blank. Numbers followed by a ' are relocatable, those followed by a * are external references, and those followed by a ! are both.

Terminal Format - Each page is headed exactly like the line printer format. The listing is like the line printer format except that the object field is only one field wide. The extra object fields are printed by themselves on successive lines below the object field of the first line.

2.2.1.1 Binary Output

The binary output produced is a TENEX file consisting of 8 bit bytes of data. Each byte is the image of one frame of paper tape. The format of the binary is standard SUE loader format. This consists of a sequence of blocks of data. Each block has one of two forms. The data blocks consist of 4 components. The first component is a byte giving the number of data bytes in this block. This count must be less than or equal to 254(decimal). The next item is an address consisting of 2 bytes, higher order part followed by low order part. This is the address of the first byte where the block of data is to be stored. Following the address are the data bytes in order of increasing address. Following the data bytes is a two byte checksum. The checksum is calculated by taking the sum of all previous bytes in the block.

The second form of block is the JUMP block. This is normally the last block on the tape and indicates to the loader both to stop loading the tape and to transfer to the location specified by the JUMP block. This block has three components. The first component is a byte containing the decimal number 255 which indicates a JUMP block. The second component is a two byte address indicating the address to transfer to. If this address is 0 the loader halts rather than starting the program. The third component is a two byte checksum, calculated by taking the sum of the previous bytes in the block.

The standard Lockheed loader and the BBN loader differ in one respect. The Pluribus needs certain registers to be loaded as words rather than bytes. The BBN loader therefore loads all data as words rather than bytes, requires an even number of bytes in each data block, and requires an even starting address for each block. This can be accomplished by requiring that each change in the location counter that immediately precedes a data-generating directive or instruction be a change to an even address, and that each sequence of consecutive directives generate an even number of bytes.

2.2.1.2 Loading Programs

The binary file generated can be copied to the paper tape punch with the system copy command. This paper tape can then be loaded in the paper tape reader of a Pluribus multiprocessor. Once the paper tape is inserted and the reader turned on, the user may press ATTN and LOAD to start the ROM loader. The loader will read the tape and halt or transfer to the start of the program depending on the .END directive at the end of the program.

2.2.2 Error Message Format

An error message has the following components:

REL-LOCTR ABS-LOCTR PAGE LINE DESCRIPTIVE-MESSAGE,

The REL-LOCTR has the form LABEL+DISP.

It gives the value of the unoffset location counter, relative to the most recently defined label. It does not appear if no labels have yet been defined (tabs are printed instead). The ABS-LOCTR is the unoffset location counter in octal. The PAGE and LINE numbers are in the source file. In addition, on the first error in a source file other than the first file, the message

FILE SUCH-AND-SUCH

will be printed.

If an error involving a direct assignment statement of the form

. = Expression

is encountered, the assembler unconditionally outputs the source line onto the Teletype printer (normally during pass 1) and into the listing file. An error of this type normally renders the object program useless.

2.3 CHARACTER SET

TAB (011), SPACE (040), and the printing characters (041-137) are treated as text information by the assembler. Lowercase alphabetics (141-172) are converted to uppercase (101-132) by the assembler, except inside .ASCII or .ASCIZ, or after a ' or ".

CR (015) signals the end of a line.

FF (014) causes a listing page to be ejected.

NULL (000), LF (012), VT (013), EOF (032), and rubout (177) are ignored.

All other characters are illegal and will cause an error.

APPENDIX A

SPECIAL CHARACTERS

Character	Function
form feed	source line terminator
line feed	source line terminator
carriage return	source statement terminator
:	Label terminator
=	Direct assignment indicator
%	Register term indicator
tab	Field and Item terminator
space	Field and Item terminator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminal register indicator
, (comma)	Operand field separator
;	Comment field indicator
+	Arithmetic addition operator
- (minus)	Arithmetic subtraction operator
&	Logical AND operator
!	Logical OR operator
?	Logical exclusive OR operator
*	Signed multiplication operator
/	Signed division operator
\	Arithmetic remainder operator

_ (underscore)	Arithmetic left shift operator
"	Double ASCII character indicator
' (quote)	Single ASCII character indicator
.	Assembly location counter
^	Indicates Macro brackets or Explicit number radix

APPENDIX B

GENERAL CLASS ADDRESS MODE SYNTAX

R is a register expression. E is an expression. ER is either a register expression or an expression in the range 0 to 7. N is the value of the ER. C is an expression in the range 0 to 15(decimal).

Format	Address Mode Name	Address Mode Number	Meaning
E	Absolute	030010	E is the address of the operand.
E(ER)	Indexed	03001N	E plus the contents of the register specified, ER, is the address of the operand.
E(-ER)	Autodecrement	01001N	The contents of register ER are decremented before contents of ER are added to E to give address of the operand.
E(ER)+	Autoincrement	02001N	The contents of the register specified by ER are incremented after the contents of ER are added to E to give the address of the operand.
R	Register	00400R	Register R contains the operand.
(ER)	Indexed	03000N	The contents of ER give the address of the operand.
(-ER)	Autodecrement	01000N	The contents of register ER are decremented before being used as the address of the operand.
(ER)+	Autoincrement	02000N	The contents of the register specified by ER are incremented after being used as the address of the operand.
@E	Indirect	030210	E specifies the address of the memory location giving address of the operand.

@E(ER)	Indirect Indexed	03021N	E plus the contents of the register specified, ER, is the address of the operand.
@E(-ER)	Indirect Autodecrement	01021N	The contents of register ER are decremented before the contents of ER are added to E to give the address of a location containing the address of the operand.
@E(ER)+	Indirect Autoincrement	02021N	The contents of the register specified by ER are incremented after the contents of ER are added to E to give the address of a memory location containing the address of the operand.
@(ER)	Indexed Indirect	03020N	The contents of ER give the address of a location containing the address of the operand.
@(-ER)	Indirect Autodecrement	01020N	The contents of register ER are decremented before ER is used as the address of a location containing the address of the operand.
@(ER)+	Indirect Autoincrement	02020N	The contents of the register specified by ER are incremented after the contents of ER is used to specify the location containing the address of the operand.
=C	Short Constant	00420C	The operand is C.
=E	Long Constant	004010	If E is an expression outside range 0 to 15, the value of the operand is E.
=E(ER)	Long Constant	00401N	The value of the operand is E plus the contents of ER.
#E	Long Constant	00401N	The value of the operand is E.

#E(ER)	Long Constant	00401N	The value of the operand is E plus the contents of ER.

APPENDIX C

INSTRUCTIONS

The instructions which follow are grouped according to the operands they take and the bit patterns of their instruction formats. There are 8 groups of instructions:

1. Control class instructions with 8 bit field
2. Control class instructions with 4 bit field
3. Control class instructions with absolute or relative addressing
4. Rotate and shift instructions
5. Branch instructions
6. General class instructions
7. Subroutine call instruction
8. Jump instruction

Op-code	Octal Value	Hex. Value	Group	Description
ADD	041000	4200	6	Add to Register
ADDB	045000	4A00	6	Add Byte to Register
ADDBM	005000	0A00	6	Add Byte to Memory
ADDM	001000	0200	6	Add to Memory
AND	041400	4300	6	And to Register
ANDB	045400	4B00	6	And Byte to Register
ANDBM	005400	0B00	6	And Byte to Memory
ANDM	001400	0300	6	And to Memory
BC	112000	9400	5	Branch if Carry
BE	110400	9100	5	Branch if Equals
BEV	104400	8900	5	Branch if Even
BF1	112400	9500	5	Branch if Flag 1 set
BF2	113000	9600	5	Branch if Flag 2 set
BF3	113400	9700	5	Branch if Flag 3 set
BG	111000	9200	5	Branch if Greater
BL	116000	9C00	5	Branch if Less than
BLE	101000	8200	5	Branch if Less than or Equals
BLP	114000	9800	5	Branch if Loop complete
BM	115400	9B00	5	Branch if Minus
BNC	102000	8400	5	Branch if Not Carry
BNE	110400	8100	5	Branch if Not Equals
BNF1	102400	8500	5	Branch if Flag 1 off
BNF2	103000	8600	5	Branch if Flag 2 off
BNF3	103400	8700	5	Branch if Flag 3 off
BNG	101000	8200	5	Branch if Not Greater
BNL	106000	8C00	5	Branch if Not Less than
BNLP	104000	8800	5	Branch if Loop Not complete
BNM	105400	8B00	5	Branch if Not Minus
BNO	104400	8900	5	Branch if Not Odd
BNOV	101400	8300	5	Branch if Not Overflow

BNZ	105000	8A00	5	Branch if Not Zero
BO	114400	9900	5	Branch if Odd
BOV	111400	9300	5	Branch if Overflow
BR	110000	9000	5	BRanch unconditionally
BZ	115000	9A00	5	Branch if Zero
CMP	043000	4600	6	Compare memory with register
CMPB	047000	4E00	6	Compare Byte with register
CMPBM	00700	0E00	6	Compare Register with Memory Byte
CMPM	00300	0600	6	Compare Register with Memory
ENB	004000	0800	2	Enable interrupts
ENW	004100	0840	2	Enable interrupts and Wait
EOR	042400	4500	6	Exclusive OR to Register
EORB	046400	4D00	6	Exclusive OR Byte to Register
EORBM	006400	0D00	6	Exclusive OR to Byte of Memory
EORM	002400	0500	6	Exclusive OR to Memory
HLT	000000	0000	1	Halt
INH	004200	0880	2	Inhibit interrupts
INW	004300	08c0	2	Inhibit interrupts and Wait
IOR	042000	4400	6	Inclusive OR to Register
IORB	046000	4C00	6	Inclusive OR Byte to Register
IORBM	006000	0C00	6	Inclusive OR to Byte of Memory
IORM	02000	0400	6	Inclusive OR to Memory
JMP	040000	4000	8	Jump
JSB	040000	4000	7	Jump to Subroutine
KEY	004020	0810	2	Set Key Register
LDA	040000	4000	6	Load Register
LDAB	044000	4800	6	Load Byte to Register
LDABM	004000	0800	6	Load Byte to Memory
LDAM	000000	0000	6	Load to Memory
MOV	040000	4000	6	Move to Register
MOVB	044000	4800	6	Move Byte to Register
MOVBM	004000	0800	6	Move Byte to Memory
MOVM	000000	0000	6	Move to Memory
MTR	003400	0700	3	Memory to Registers
MTS	002400	0500	3	Memory to Status
NOP	100000	8000	5	No OPeration
RET	002000	0400	3	Return from interrupt
RLA	120400	A100	4	Rotate Left Arithmetic
RLL	121400	A300	4	Rotate Left Logical
RRA	122400	A500	4	Rotate Right Arithmetic
RRL	123400	A700	4	Rotate Right Logical
RST	001000	0200	1	Reset Status
RTM	001400	0300	3	Register to Memory
SLA	120000	A000	4	Shift Left Arithmetic
SLL	121000	A200	4	Shift Left Logical
SRA	122000	A400	4	Shift Right Arithmetic
SRL	123000	A600	4	Shift Right Logical
SST	001200	0280	1	Set Status
STM	000400	0100	3	Status to Memory
SUB	040400	4100	6	Subtract
SUBB	044400	4900	6	Subtract Byte

SUBBM	004400	0900	6	Subtract Byte from Memory
SUBM	000400	0100	6	Subtract from Memory
TST	043400	4700	6	Test
TSTB	047400	4F00	6	Test Byte
TSTBM	007400	0F00	6	Test Byte with Memory
TSTM	003400	0700	6	Test with Memory

APPENDIX D

ASSEMBLER DIRECTIVES

Menmonic	Operand	Stands for	Operation
.ADDRE	Statement	Address word	Returns second word statement
.ADRMD	ADDR	Address Mode	Returns 16 bit address field for ADDR
.ASCII	/xxx...x/	ASCII	Generates 8-bit ASCII characters for text enclosed by delimiters.
.ASCIZ	/TEXT/		Same as .ASCII but a zero byte is appended to the text string.
.BLKB	N	Block of bytes	Reserve (assemble) a block of N uninitialized bytes
.BLKW	N	Block of words	Starting on an even address (advance one byte if necessary) reserve a block of N uninitialized words.
.BYTE	E, E,...	BYTE	Generates bytes of data
.END	E	END	Indicates the physical end of the program and optionally specifies the transfer address (E).
.ENDC	none	END of Conditional	Terminates the range of a conditional directive.
.ENDM		End Macro	Terminates a .MACR or .TTYMA
.ENDR		End Repeat	Terminates a .REPT
.ENTRY	NAME	Entry	Entry to routine Makes a routine entry point accessible to separately assembled programs.

.EOT	none	End of Tape	Indicates the physical end of the source input medium.
.ERROR	MSG	Error	Causes an error whose message is just the source line ".ERROR MSG". As with other errors, the PC, page, and line are printed.
.EVEN	none	EVEN	Insures that the assembly location counter is even by adding 1 if it is odd.
.EXTRN	NAME	External Name	Declares a request for a routine in a separately assembled program.
.FIRST	Statement	First Word	Returns first word of statement
.IF	CONDITION	Conditional Assembly	Assemble lines up to the following .ENDC only if condition is true.

The Conditions are:

1) Arithmetic Conditionals

TEST EXP
Where test can be LE,LT,GE,GT,NZ,Z
and any expression can be used. e.g., .IF Z x-y

2) String Comparison:

DIF ARG1.ARG2 Where arg1 and arg2 are macro-type args.
IDN ARG1,ARG2

3) String Comparison with Null String.

B ARG	True if arg (a macro-type arg) is null.
NB ARG	True otherwise.

4) Definition Testing

DF EXP	True if all syms in exp are defined.
NDF EXP	The opposite.

5) Pass Number Testing

P1	True on pass 1
P2	True on pass 2

.IIF CONDITION STMT
Conditional assembly
Assemble STMT only if CONDITION is true. See .IF for description of conditions.

.INSRT FILSPC Insert a file
Pushes the current source file and/or REPEAT, MACRO, etc. and starts reading from the specified file. The file or MACRO containing the .INSRT will be resumed starting with the next line.

.IRP DUMMY,"(A1,A2,A3")
Repeated Substitution
Repeats all the following text up to the matching .ENDM substituting A1 for DUMMY the first time, A2 for DUMMY the second time, etc. The stuff following the comma is a macro-arg and might also be delimited by /.../, for example.

.IRPC DUMMY STRING
Repeated Char Substitution
Repeats the text up to a matching .ENDM once for each character in STRING, each time substituting that character for DUMMY. DUMMY must be a symbol; STRING is read in as a macro-argument.

.LENGTH STRING Number chars
Returns the number of characters in STRING, which is read in as a macro-argument. When used in an arithmetic expression, a comma should appear after STRING and before any following arithmetic operator.

.LIF CONDITION Conditional assembly
STMT-ON-NEXT-LINE
Assembles the next line only if CONDITION is true. If CONDITION fails, list next line but do nothing with it. See .IF for description of conditions.

.LIST	Enable listing		Decrement XLIST count if not already 0. Listing takes place only if XLIST count equals 0.
.MACRO M	A,B,C	Macro Def	Define M as a MACRO with arguments named A,B, and C. The MACRO definition is on the following lines, up to the matching .ENDM. (Intervening matching. MACR's and .ENDM's go into the MACRO being defined.)
.MEXIT		Leave macro	Pops out of the innermost .REPT, macro-call, .IRP, or .IRPC.
.MSG	LINE	List message	Like .ERROR LINE but doesn't increment the error count.
.ODD		Odd Address	Moves to next odd address. Same as .=.+1-<.&1>
.OFFSET	EXP	Offset the PC	When "." is used in relative addressing or label definitions the offset is added to the real location counter. Offset is zero at the beginning of each pass.
.PAGE		New Page	Starts a new page in listing.
.PRINT	/TEXT/	Type out	Prints the text on TTY.
.RADIX	N	Set Radix	Change radix to N
.RAD50	/XXX/	RADix 50	Generates the RADIX 50 representation of the ASCII characters in delimiters.
.REPT	EXP	Repeat	Assembles text up to the matching .ENDR EXP times.
.STITL	LINE	Subtitle	LINE is listed on the 2nd

			line of each header. If .STITL is on the first line of a page, it takes effect on that page, otherwise on the next page.
.TITLE	name	TITLE	Generates a name for the object module.
.TTYMA	A,B,C	Teletype macro	Reads a line from the TTY, defines A,B, and C from that line using the macro-arg scanning rules. Within the scope of the .TTYMAC (up to a matching .ENDM) A,B,C will be replaced by scanning what was read from TTY.
.WORD	E, E,...	WORD	Generates words of data
	E, E,...	(the void operator)	Generates words of data
.XCREF	A,B,C	Don't CREF	Prevents CREF output from being generated for symbols A,B,C.
.XLIST		Don't list	Increments XLIST count. Listing is done only if XLIST count is zero.

APPENDIX E

INITIAL SYMBOL TABLE

The following symbols are pre-defined for programming I/O devices etc.

SYMBOL	OCTAL VALUE	HEX. VALUE
%A	004000	0800
%ABRT0	000050	0028
%ABRT1	000070	0038
%ABRT2	000110	0048
%ABRT3	000130	0058
ADD	041000	4200
ADDB	045000	4A00
ADDBM	005000	0A00
ADDM	001000	0200
AND	041400	4300
ANDB	045400	4B00
ANDBM	005400	0B00
ANDM	001400	0300
%AREG1	177600	FF80
%AREG2	177604	FF84
%AREG3	177610	FF88
%AREG4	177614	FF8C
.ASCII	PSEUDO-OP	
.ASCIZ	PSEUDO-OP	
BC	112000	9400
%BDADR	000000	0000
%BDINS	000000	0000
BE	110400	9100
BEV	104400	8900
BF1	112400	9500
BF2	113000	9600
BF3	113400	9700
BG	111000	9200
BL	116000	9C00
BLE	101000	8200
.BLKB	PSEUDO-OP	
.BLKW	PSEUDO-OP	
BLP	114000	9800
BM	115400	9B00
BNC	102000	8400
BNE	100400	8100
BNF1	102400	8500
BNF2	103000	8600
BNF3	103400	8700
BNG	101000	8200
BNL	106000	8C00

BNLP	104000	8800
BNM	105400	8B00
BNO	104400	8900
BNOV	101400	8300
BNZ	105000	8A00
BO	114400	9900
BOV	111400	9300
BR	110000	9000
.BYTE	PSEUDO-OP	
BZ	115000	9A00
%C	000010	0008
CMP	043000	4600
CMPB	047000	4E00
CMPBM	007000	0E00
CMPM	003000	0600
%CPU0	177400	FF00
%CPU1	177440	FF20
%CPU2	177500	FF40
%CPU3	177540	FF60
%CTRL	000036	001E
%CURPC	000004	0004
%DEVNO	000000	0000
%DREG1	177602	FF82
%DREG2	177606	FF86
%DREG3	177612	FF8A
%DREG4	177616	FF8E
%E	000001	0001
ENB	004000	0800
.END	PSEUDO-OP	
.ENTRY	PSEUDO-OP	
ENW	004100	0840
EOR	042400	4500
EORB	046400	4D00
EORBM	006400	0D00
EORM	002400	0500
.EOT	PSEUDO-OP	
.ERROR	PSEUDO-OP	
.EVEN	PSEUDO-OP	
.EXTRN	PSEUDO-OP	
%FNAM2	PSEUDO-OP	
%F1	000020	0010
%F2	000040	0020
%F3	000100	0040
%G	000002	0002
HLT	000000	0000
.IF	PSEUDO-OP	
.IFF	PSEUDO-OP	
%ILOP0	000040	0020
%ILOP1	000060	0030
%ILOP2	000100	0040
%ILOP3	000120	0060

INH	004200	0880
.INSRT	PSEUDO-OP	
%INST	000022	0012
INW	004300	08C0
IOR	042000	4400
IORB	046000	4C00
IORBM	006000	0C00
IORM	002000	0400
.IRP	PSEUDO-OP	
.IRPC	PSEUDO-OP	
.IRPCN	PSEUDO-OP	
JMP	040000	4000
JSB	040000	4000
KEY	004020	0810
%LADR	000024	0014
LDA	040000	4000
LDAB	044000	4800
LDABM	004000	0800
LDAM	000000	0000
.LIF	PSEUDO-OP	
.LIST	PSEUDO-OP	
%LOW	000400	0100
%LVL1	000000	0000
%LVL2	000010	0008
%LVL3	000020	0010
%LVL4	000030	0018
.L1	000001	0001
%L1	010000	1000
.L2	000002	0002
%L2	020000	2000
.L3	000004	0004
%L3	040000	4000
.L4	000010	0008
%L4	100000	8000
%LP	000200	0080
%MAP0	176000	FC00
%MAP1	176002	FC02
%MAP2	176004	FC04
%MAP3	176006	FC06
MOV	040000	4000
MOVB	044000	4800
MOVBM	004000	0800
MOVM	000000	0000
.MSG	PSEUDO-OP	
MTR	003400	0700
MST	002400	0500
%NARG	PSEUDO-OP	
%N	002000	0400
NOP	100000	8000
%O	000400	0100
.ODD	PSEUDO-OP	

%OFFSE	PSEUDO-OP	
.OFFSET	PSEUDO-OP	
.PAGE	PSEUDO-OP	
.PRINT	PSEUDO-OP	
.RAD50	PSEUDO-OP	
%REG0	000000	0000
%REG1	000002	0002
%REG2	000004	0004
%REG3	000006	0006
%REG4	000010	0008
%REG5	000012	000A
%REG6	000014	000C
%REG7	000016	000E
.REPT	PSEUDO-OP	
RET	002000	0400
RLA	120400	A100
RLL	121400	A300
.RPCNT	PSEUDO-OP	
RRA	122400	A500
RRL	123400	A700
RST	001000	0200
RTM	001400	0300
%SERVC	000006	0006
SLA	120000	A000
SLL	121000	A200
SRA	122000	A400
SRL	123000	A600
SST	001200	0280
%STAT	000020	0010
.STITL	PSEUDO-OP	
STA	000000	0000
STAB	004000	0800
STM	000400	0100
SUB	040400	4100
SUBB	044400	4900
SUBBM	004400	0900
SUBM	000400	0100
.TITLE	PSEUDO-OP	
TST	043400	4700
TSTB	047400	4F00
TSTBM	007400	0F00
TSTM	003400	0700
%V	000004	0004
.WORD	PSEUDO-OP	
%XCREF	PSEUDO-OP	
.XCREF	PSEUDO-OP	
.XLIST	PSEUDO-OP	
%XLIST	PSEUDO-OP	
%0	000000	0000
%1	000001	0001
%2	000002	0002

%3	000003	0003
%4	000004	0004
%5	000005	0005
%6	000006	0006
%7	000007	0007

APPENDIX F

SUE Opcode Equivalents

The Lockheed LAP-2 assembler, the PDP-1d PMIDAS assembler and this assembler all use different names for the processor operations. The LAP-2 assembler also differentiates the direction of data movement from the syntax of the statement. The Pluribus and PMIDAS assemblers do not. Tke following table gives the equivalent names in the three assemblers. A asterisk (*) following a name indicates that the direction of data movement is to memory.

LAP-2 Opcode	PLURIBUS Opcode	PMIDAS Opcode
	General Register Instructions	
ADDB	ADDB, ADDBM*	ADDB, ADDBM*
ADDW	ADD , ADDM*	ADD, ADDM*
ANDB	ANDB, ANDBM*	ANDB, ANDBM*
ANDW	AND , ANDM*	AND, ANDM*
CMPB	CMPB, CMPBM*	CMPB, CMPBM*
CMPW	CMP , CMPM*	CMP, CMPM*
EORB	EORB, EORBM*	EORB, EORBM*
EORW	EOR , EORM*	EOR, EORM*
IORB	IORB, IORBM*	IORB, IORBM*
IORW	IOR , IORM*	IOR, IORM*
MOVB	LDAB, STAB*	LDAB, STAB*
MOVW	LDA , STA*	LDA, STA*
SUBB	SUBB, SUBBM*	SUBB, SUBBM*
SUBW	SUB , SUBM*	SUB, SUBM*
TSTB	TSTB, TSTBM*	TSTB, TSTBM*
TSTW	TST , TSTM*	TST, TSTM*
	Jump Instructions	
JSBR	JSB	JSB
JUMP	JMP	JMP
	Branch Conditional Instructions	
NOPR	NOP	NOP
BRUN	BR	BT TR
BEQT	BE	BT EQ
BGTT	BG	BT GT
BLTT	BL	BT LT
BZET	BZ	BT ZE
BNGT	BM	BT NG
BLPT	BLP	BT LP
BODT	BO	BT OD
BOVT	BOV	BT OV
BCYT	BC	BT CY

BF1T	BF1	BT F1
BF2T	BF2	BT F2
BF3T	BF3	BT F3
BEQF	BNE	BF EQ
BGTF	BNG	BF GT
BLTF	BNL	BF LT
BZEF	BNZ	BF ZE
BNGF	BNM	BF NG
BLPF	BNLP	BF LP
BODF	BNO	BF OD
BOVF	BNOV	BF OV
BCYF	BNC	BF CY
BF1F	BNF1	BF F1
BF2F	BNF2	BF F2
BF3F	BNF3	BF F3

Shift Instructions

SLAO	SLA	LI AO, LX AO
SLLO	SLL	LI LO, LX LO
SLLC	RLL	LI LC, LX LC
SLLL	RLA	LI LL, LX LL
SRAO	SRA	RI AO, RX AO
SRLO	SRL	RI LO, RX LO
SRLC	RRL	RI LC, RX LC
SRLL	RRA	RI LL, RX LL

Control Instructions

RETN	RET	RET
STSM	STM	STM
REGM	RTM	RTM
MSTS	MTS	MTS
MREG	MTR	MTR
HALT	HLT	HLT
DSBL	INH	INH
DSBW	INW	INW
ENBL	ENB	ENB
ENBW	ENW	ENW
SETS	SST	SST
RSTS	RST	RST
SKEY	KEY	KEY, SKEY

APPENDIX G

Additions and Limitations

The relocatable code format for the Pluribus is not yet implemented. The following additions will be made to the language later: symbol generation feature; an .EXPUNGE command to eliminate symbols; a branching assembler directive which determines if an address is within range or outside range and generates either a branch or a branch over a jump accordingly; a remote code feature which allows code to be specified at one point but assembled at another.

PLURIBUS DOCUMENT 5: ADVANCED SOFTWARE

PART 4: SYSTEM RELIABILITY PACKAGE