HOSPITAL COMPUTER PROJECT

A JOINT RESEARCH EFFORT OF THE MASSACHUSETTS GENERAL HOSPITAL AND BOLT BERANEK AND NEWMAN INC

MEMORANDUM NUMBER SIX-E

PROGRAMMING SOFTWARE STATUS REPORT

Submitted by Bolt Beranek and Newman Inc Cambridge, Massachusetts 1 May, 1966

HOSPITAL COMPUTER PROJECT*

M E M O R A N D U M S I X E

Status Report on Programming Software

Submitted by

BOLT BERANEK AND NEWMAN INC

Medical Information Technology Department

Jordan J. Baruch, Principal Investigator, 1962-1966
Paul A. Castleman, Principal Investigator, 1966-
Richard H. Bolt, Supervisor, 1966-

Nancy Adams
John Barnaby
Sheldon Boilen
Richard A. Bolt
Michael Cappelletti
Jonathan Cole
Bernard Cosell
Edward Gilbert
Edith Grossman
Nancy Haggerty
Alice Hartley
Gray Hodges
Toby Jensen
Charles Jones
Larry Kershaw
Philip LaFollette
Nancy Lambert
Maureen Letu
Sanford Libman
Barbara Lieb
William Lucci
Richard Mahoney
Ernest McKinley
Charles Morgan
Andrew Munster
Nicholas Pappas
Robert Payson
Lawrence Polimeno
Sylvia Sarafian
William Slack
Sandra Sommers
Lee Stein
Peter Storkerson
Sally Teitelbaum
Frederick Webb
Walter Weiner
Steven Weiss

* This joint research project is supported by Grant FR 00263 and Contract PH 43-62-850 from the National Institutes of Health and the American Hospital Association. Efforts contributed by the Laboratory of Computer Science, Massachusetts General Hospital, are directed by Principal Investigator G. Octo Barnett, M.D. The report, Memorandum Six E, gives a technical description of programming software developed by BBN for the computer system being used on this project.

TABLE OF CONTENTS

TABLE OF CONTENTS cont.

TABLE OF CONTENTS cont.

HOSPITAL COMPUTER PROJECT
STATUS REPORT MEMORANDUM SIX E
PROGRAMMING SOFTWARE

INTRODUCTION

This is the fifth volume of Memorandum Six, a series of reports describing the Hospital Computer System developed by Bolt Beranek and Newman in collaboration with Massachusetts General Hospital, under sponsorship of the National Institutes of Health and the American Hospital Association. Other volumes in the series describe the system hardware, the time-sharing Executive and Common Routines, the hospital user programs, and the information storage and retrieval programs.

This volume describes the on-line Programming System that is used to compose, edit, assemble, and debug software for the PDP-1b-45, the central processor designed for use in the collaborative project.

The current Programming System, with its full range of programming aids, is the realization of goals first approached at Bolt Beranek and Newman by the Simbug System. Simbug, one of the first operating time-shared systems in the general-purpose category, enabled several users to perform debugging operations simultaneously. The on-line Programming System described here accommodates a full staff of programmers and offers each of them, in effect, sole access to a general-purpose computer.

The system consists of four programs: Midas, a macro-assembler; Editor, a symbolic text editor; the DDT debugging program; and Handle, a program that provides for the manipulation of files generated by the three other programs in the system.

Chapter I contains a detailed description of the Midas assembly language. Midas offers the programmer a wide range of expression, permitting him to deal with individual computer words or to manipulate large blocks of computation.

Chapter II describes Editor, the on-line text editor used for preparing symbolic programs to be assembled by Midas.

Chapter III provides a description of DDT, the system's debugging and control program. DDT serves two purposes within the Programming System. First, it is used to check out and correct new programs that, for this purpose, run under its control. Second, DDT controls the running of Midas, Editor, and Handle, providing inter-program communication within the Programming System.

The final chapter, Chapter IV, describes Handle. The discussion of Handle includes, in addition to its actual functions and commands, a description of the file organization for the entire Programming System.

I. THE MIDAS ASSEMBLY SYSTEM

The Midas Assembly System described in this volume was adapted by Bolt Beranek and Newman from the Midas Assembler originally written for the PDP-1 at Massachusetts Institute of Technology. Some features of the Massachusetts Institute of Technology's system have been eliminated. Other features have been added to take advantage of the file structure of the Bolt Beranek and Newman system and to provide on-line assembly via Teletype terminal.

Like the Massachusetts Institutes of Technology's version of Midas, the Bolt Beranek and Newman system incorporates the basic features of the Macro Assembly Program. Macro was also developed at the Massachusetts Institute of Technology, first for the TX-Ø computer, and then for the PDP-1. The major departure that Midas makes from Macro lies in the handling of macro-instructions. A macro-definition in a Macro source program is partially assembled when first encountered; that is, everything except dummy symbols is translated into machine language and stored in the macro table. In Midas the actual source language text of the definition is stored in the macro table, and a complete assembly is performed each time the macro is used. This method significantly extends the system's macro-instruction capabilities, permitting recursive and conditional definitions. The advantage gained by the Midas method for storing macro-definitions will become clear when macro-instructions are discussed in detail later in this section.

The following pages assume that the reader has a working knowledge of basic PDP-1 mnemonic codes and their octal representations. The reader who does not is referred to Volume Six A of this Memorandum, Chapter II, Section B, page 16.

A. INTRODUCTION TO ASSEMBLY

A problem that can be expressed quite briefly in words or mathematical notation will often, when put in the form of a computer program written in machine language, require hundreds of computer instructions. Since the value and location of each of these must be fully specified, the clerical effort involved in writing a long program may exceed that expended in analyzing the problem in terms of individual computer operations. Further inconveniences that arise from handling large amounts of information in the form of numerical code are the introduction of clerical errors into a correctly formulated program and the need for detailed documentation so that others may understand the coding.

Such difficulties led to the development of translating programs that make a computer available as an aid in the preparation and documentation of the programs that will be run on it. These translators permit the programmer to express a computation in terms of a convenient mnemonic language (the source language), which the translating program is able to translate into machine code (the object language).

Two general categories of translating programs, compilers and assemblers, have been developed to accept symbolic input and produce binary output. Compilers are translators that are "problem-oriented"; that is, the various compiling programs are designed to interpret a symbolic language similar to the language in which the problem would originally be stated. Thus, a compiler designed for mathematical applications would accept mathematical expressions, and in addition to translating the numbers and symbols into appropriate quantities, would translate the entire expression into an appropriate sequence of computer instructions. A PDP-1 compiler, for example, might translate the expression Z=X+Y into the series —LAC X, ADD Y, DAC Z. In the same way, compilers designed for business applications are designed to accept a language in which it is convenient to state business problems.

Because they accommodate various users in terms with which they are familiar, compilers permit a person with a limited knowledge of computers to write his own programs. Assemblers, on the other hand, are "computer-oriented," consisting principally of instructions which correspond to internal computer instructions. The simplest assemblers also include a minimal number of rudimentary control operations that direct the translation process.

The Midas Assembly Program, while offering all the basic 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 (Midas 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. The name is then, in effect, an abbreviation of the sequence that will be substituted for it at assembly time, with desired parameters inserted in their proper contexts. A programmer might, for example, take a sequence such as LAC X, ADD Y, DAC Z, assign it the name ADDXY, specify that X, Y, and Z are dummy symbols for which program symbols will be substituted as desired and then use merely the macro name ADDXY followed by appropriate symbols throughout the rest of the source program. The assembler will generate the predefined coding just as the compiler generates coding appropriate to the expression it is interpreting. Other pseudo-instructions available in Midas are ones that 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 Section I-A.

Section I-B describes the functions and formats of all system pseudo-instructions.

Section I-C explains the use of macro-instructions for performing an assembly.

Section I-D provides instructions for performing an assembly.

Error conditions that are detected during an assembly and associated error messages are listed in Section I-E.

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

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 and places these words in their proper locations 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.

a. 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 net specified as illegal.

The character set is generally divided into the following categories:

<u>alphanumeric characters</u>:	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.
<u>combining operators</u>:	Single characters representing fixed arithmetical or logical operations to be performed by Midas.
<u>punctuation characters</u>:	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 complete listing. Most generally used delimiters are space, tab, and carriage return.

b. Legal Strings

(1) Basic Strings

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

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

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).

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 of the identifying six must be a letter.

Longer symbols, useful for mnemonic or documentary purposes, may be used, since the Midas processor 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.

(2) Complex Strings

(i) 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 character <.>, used to represent the current value of the location counter modulo ($2^{12.}$), functions as a syllable.

Expression

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

(ii) 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"	Folded 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 $2^{18.}-1$
-	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 additive ones. If a string of consecutive additive operators occurs, Midas will perform only the last. A string of consecutive product operators, however, cannot be processed.

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

a)	Expressions:	LIO 1ØØ	1ØØ LIO	22Ø1ØØ
	Syllables:	<LIO>,<1ØØ>	<LIO>,<1ØØ>	<22Ø1ØØ>
b)*	Expressions:	LIO ADD	JSP	62ØØØØ
	Syllables:	<LIO>,<ADD>	<JSP>	<62ØØØØ>
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).

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.

a. The Current Location Counter

The Midas Assembler assign assembled words to sequential locations, starting from any given location. A register in the assembly program, referred to as the Current-Location Counter, is indexed whenever a location is assigned, indicating the location which will be assigned to the next word assembled. It is

*Assume that LIO=22ØØØØ, ADD=4ØØØØØ, and JSP=62ØØØØ.

initially set at 11∅ 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 any sequence. Using other conventions, the programmer may direct the assembler to increment the current location a specified number of times although no words are assembled.

A source language string may be specified as a direct representation of the value of the location counter at any time during the assembly and used in place of that value throughout the source program. In addition, a single character is provided that may be used to represent its value in any word while it is being assembled. Any value assumed by the current location counter is an actual location of a storage word. Representations of its value, however, are equivalent to the sum of the location plus a number referred to as the <u>offset count</u>. Unless reset by the programmer, the offset count will be ∅, and representations will coincide with the locations at which they are derived. The offset count may be set by the programmer, however, and an internally consistent set of values derived relative to the locations being assigned. The utility of offsetting the current location is discussed later in connection with the pseudo-instruction OFFSET. It is mentioned here to emphasize the fact that a number is assigned to a symbolic address tag in order to utilize a list position rather than to represent it.

While the fact that program addresses and computer instruction codes may be symbolically represented is sufficient knowledge for writing source programs, the fact that these represent numbers to the assembler, and locations and operations only in context of the source program, will be useful to understanding the full range of symbolic manipulations that may be performed. For example, if a computation to be performed requires the number 1ØØ, we might <LAW> the number rather than provide a register containing it, and if the number 1ØØ is represented by the symbol <A>, we can <LAW A> whenever we need 1ØØ in the accumulator whether <A> derived its value as an address tag or as a parameter. As further illustration, one step in the computation might be to add the contents of register 1ØØØ to the value in the accumulator, requiring a word in the executable sequence containing the number 4Ø1ØØØ. Given the symbol <ADD> representing the numerical operation code <4ØØØØØ>, the symbol <B>, which has been set to equal <1ØØØ> as a symbol address tag, and the operator space, which signifies addition, the source language expressions, <ADD B> or <B ADD>, would give the proper value. Storage of the expression in an executable sequence is accomplished by setting the expression in the proper context. If we need to perform a computation using the number <4Ø1ØØØ>, we might think in terms of creating a data word to contain it when we could just as well use the instruction word and would particularly wish to when computer space is limited. The source language takes advantage of the fact that a computer word may be an instruction, data, or both according to its context in the program, in that strings, while associated with a number in one prescribed ccntext, may then be generally applied.

While the advantages of contextual as well as explicit interpretation may be clear to programmers who are familiar with the techniques of machine language coding and who will be looking for these properties in the source language, others might well tend to think in terms of symbols for addresses and symbols for words as quite different entities.

b. 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 PDP-1 instruction codes are part of the initial contents of the symbol table. (A list of these is included in Appendix B.) These and programmer constructed symbols, which stand for numerical values, are "substitutive" 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 entered in the macro table, and a macro name is defined as a reference to the macro table location of the instruction to which it is assigned.

Midas assembles a source program in two passes; that is, two complete scannings of the source program are necessary to produce an object program. Midas was constructed this way so that symbols may be associated with values at any point in the source program without restricting their use prior to definition, thus permitting the programmer freedom from rigid format constraints.

c. 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 8 for accepting octal integers. Subsequent settings of 8 or 1∅ may be effected by source program specification.

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 II in connection with the pseudo-instructions, EQUALS and OPSYN.

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

a. 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 obtained by adding the value of the location counter when scanned to the offset count. The resulting value is entered into the symbol table modulo ($2^{12.}$) if terminated by a comma, and modulo ($2^{16.}$) if by a colon. 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+3>, terminated by a comma or colon will be admitted for use as an address tag only if it coincides with the value of the current location counter plus the offset count. Since relative addressing using address arithmetic is an intrinsic system capability, an untagged word may always, in effect, be referenced by such an expression. If a previously defined symbol occurs as an address tag, the symbol will not be redefined and will be admissible only if its value agrees with that of the current location counter plus the offset count.

b. 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 the identifying string itself, that is, within the first six characters, in at least one of its appearances. The symbol is stored in the symbol table without the qualifying character and may be referred to with or without it. Actual values are entered for all variables still undefined at the appearance of the pseudo-instruction VARIABLES. The following are legal variable names: <ABC#>,<#ABC>,<AB#C>. For regularity of documentation the second form listed, however, is usually used.

c. Assigned Parameters

A programmer may, with an identity declaration, assign a value to a symbol that is to be used as a parameter.

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.

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.

a. 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 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.

b. 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 the accumulator could be written as <SUB(1ØØ.)>. Midas will 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

```
(1)  LAC(LAW(FLEXO ABC))
```

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

For example,

```
(2)  LAC(LAW(FLEXO ABC↵
```

is equivalent to (1) above.

Omission of the right parenthesis in

```
(3)  LAC(LAW(ABC)-1↵
```

would, however, change the meaning.

c. 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. 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 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 I-D permits the programmer to arrange for Midas to type a message if the location becomes indefinite on Pass 1.

5. 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.

Figure 1. Two sample Midas programs for solving the equation: Y = 7A+B/2, for A = 4Ø, B = 6Ø. The symbolic programs in columns (1) and (2) will produce the machine language program in column (3).

```
      (1)                       (2)                          (3)
SAMPLE PROGRAM 1          SAMPLE PROGRAM 2

1ØØ/    LAC A             1ØØ/        LAC (4Ø          1ØØ    2ØØ111
        MUL SEVEN                     MUL (7           1Ø1    54Ø112
        RIR 1S                        RIR 1S           1Ø2    672ØØ1
        DIO SEVENA                    DIO SEVENA       1Ø3    32Ø114
        LAC B                         LAC (6Ø          1Ø4    2ØØ113
        SAR 1S                        SAR 1S           1Ø5    675ØØ1
        ADD SEVENA                    ADD #SEVENA      1Ø6    4ØØ114
        DAC Y                         DAC #Y           1Ø7    24Ø115
        HLT                           HLT              11Ø    76Ø4ØØ
A,      4Ø                CONSTANTS                    111    ØØØØ4Ø
SEVEN,   7                VARIABLES                    112    ØØØØØ7
B,      6Ø                START 1ØØ                    113    ØØØØ6Ø
SEVENA, Ø                                              114    ØØØØØØ
Y,      Ø                                              115    ØØØØØØ

START 1ØØ
```

PRINTOUT FROM ASSEMBLY:

```
      (1)                                       (2)

DEFINED SYMBOLS ALPHABETIC              CONSTANTS AREA,  INCLUSIVE

                                              FROM          TO

  A       111                                 111           113
  B       113
  Y       115                           DEFINED SYMBOLS ALPHABETIC
 SEVEN    112                                  Y            115
SEVENA    114                                SEVENA         114
```

The strings <CONSTANTS>, <VARIABLES>, and <START> in the sample programs are pseudo-instructions, which will be discussed in the following section.

B. 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.

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.

2. CHARACTER, FLEXO, and TEXT

These three pseudo-instructions were originally devised for storing 6-bit Friden code characters. The translation of 7-bit Teletype code to Internal Code may result in 6 or 12. bits. It has been left to the programmer to introduce 12.-bit code only where two 6-bit characters are permitted. A table of characters and their Internal Code is included as Appendix C.

The pseudo-instruction CHARACTER is used to place a character code in the left, middle, or right 6-bit portion of an 18.-bit word. The symbol, CHARACTER, is followed by L, M, or R according to desired bit position, immediately followed by the character to be coded. Characters with 12.-bit codes cannot be stored this way.

The format is	Stored as
CHARACTER RA	ØØØØ41
CHARACTER MB	ØØ42ØØ
CHARACTER LC	43ØØØØ

The character strings shown above are pseudo-instruction syllables and may be used in the same manner as symbols or integers. For example,

```
LAC (CHARACTER LA + DTB) is equivalent to
LAC (41ØØØØ + DTB.
```

The pseudo-instruction FLEXO is used to specify an entire computer word filled with character code, either three 6-bit characters or one 12.-bit and one 6-bit.

The format is		Stored as
FLEXO ABC		<u>414243</u>
	or	
FLEXO A←		<u>417776</u>

The pseudo-instruction TEXT is used to assemble a string of characters by groups of two or three, as appropriate, into successive words in the object program.

A TEXT statement consists of the symbol <TEXT> 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 <TEXT /MESSAGE/>, Midas will store character code for the word, MESSAGE. A 12.-bit code character is not a good choice for 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.

If the character <#> is used in the argument of CHARACTER, FLEXO, or TEXT, it is handled in an exceptional way; it is stored as end-of-message (Internal Code 74) rather than as its own Internal Code configuration, Ø3. Consequently, there is no provision for entering <#> as actual text. Most hospital user programs use the code, 74, as a text terminator; it may appear in no other context.

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.

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 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 ten times in the same program.

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

```
        CONSTANT AREA RESERVED, INCLUSIVE
          FROM          TO
           325          337
```

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

```
        CONSTANTS AREA, INCLUSIVE
          FROM          TO
           325          332
```

indicates those registers actually containing constants.

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 still 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 and the result obtained entered as its absolute address. Each defined variable represents an address at which Midas assigns a storage word whose 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. Thus it is wise for a programmer to use Midas command "E" (described in Section I-D) to arrange a printout if the location becomes indefinite on Pass 1 so that he can correct the condition before the processing of VARIABLES.

In the current version of Midas the pseudo-instruction VARIABLES may be used no more than ten 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 subsequent symbol definitions or macro-expansions have altered the sequence of assembled words.

5. DIMENSION

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

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

where 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.

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 effects this 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 CHAR, CHARACTER
```

and then defines a macro-instruction, CHARACTER, in terms of CHAR, which now calls the pseudo-instruction, on Pass 2 CHAR will again be made equivalent to CHARACTER, which has been redefined. CHAR 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, CHAR will be associated with CHARACTER as desired on Pass 1 only and retain its identity with the original pseudo-instruction on Pass 2.

7. NULL

The NULL pseudo-instruction performs no action, but it is 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.

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 garbage collector will soon be available for this system. Available macro storage space is $61,44\emptyset_{10}$ computer words (approximately 18∅,∅∅∅ characters).

The psuedo-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 with which it is associated. For example, the coding:

```
OFFSET 5
ABC,    LAW 1ØØ
        JMP ABC
```

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

```
1ØØ,  LAW 1ØØ
      JMP 1Ø5
```

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.

9. <u>REPEAT</u>

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 <u>count</u> of the REPEAT, specifying the number if iterations desired, and the TEXT is the source program section to be iterated, called the <u>range</u> 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 are used within the range to denote storage words. Tabs may also appear within a macro-definition or as an argument. Brackets may be used to enclose portions of the range or the entire range so that carriage returns may 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, LAC A →| ADD B →| DAC C
```

will generate for assembly the coding

```
LAC A
ADD B
DAC C
LAC A
ADD B
DAC C
```

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

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 I-D), Midas appends to the binary output a word containing <JMP EXPRESSION>, where EXPRESSION is the argument of the last START processed.

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.

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 I-D.

13. ØIF and 1IF

A programmer may find it useful, particularly when handling comples 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 a defined symbol
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    SYMBOL
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:

```
LAC (ØIF VP X/+3
```

is equivalent to

```
LAC (3) or LAC (4)
```

while

```
LAC (ØIF VP X+3
```

is equivalent to

```
LAC (Ø) or LAC (1)
```

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

For example:

```
        (1)  END: Ø

             REPEAT ØIF VP 7777-END, PRINTX /OVERFLOWED CORE/
```

The address assigned to END (colon indicating address modulo $2^{16.}$) is subtracted from 7777. If 7777 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.

```
        (2)  REPEAT 1IF P, EXPUNGE TYO,TYI,ONE
```

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

14. PRINT and PRINTX

The pseudo-instructions PRINT and PRINTX 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 is

```
                    PRINT or PRINTX TEXT
```

where the argument may be text of the form used with the pseudo-instruction TEXT 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 causes 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 ∅IF or 1IF, with the qualifier P. For example, in response to

```
REPEAT ∅IF P, PRINT /TEXT/,
```

Midas will print only on Pass 1.

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.

Within the range of a REPEAT, STOP will halt the expansion of all subsequent text unless the REPEAT occurs within the body of a macro-instruction or an IRP. In that case STOP, whether in a REPEAT range or not, will suppress subsequent coding until the occurrence of the next TERMINATE or ENDIRP.

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.

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

C. 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.

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 dummy 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.

a. <u>Basic Format</u>

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

```
(1) DEFINE
        ABSOLUTE   -   (MACRO NAME)
        SPA  }
             }     -   (BODY OF THE MACRO)
        CMA  }
    TERMINATE
```

The macro name, <ABSOLUTE>, subsequently serves as a macro call in the source program. Midas will assemble the body of the macro (<SPA> and <CMA>) into the object program at every appearance of the macro call.

```
(2) DEFINE
        SUM A,B,C
        LAC A
        ADD B
        DAC #C
    TERMINATE
```

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

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

```
    LAC XORG
    ADD XINC
    DAC #XMAX
```

b. Dummy Arguments

A programmer may use as many distinct symbols as desired 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 for which the programmer must supply symbols 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

*The following delimiters are also acceptable: +, -, space, ∪, ∩, ~, tab, =, (,).

symbol rather than generate one. A generated symbol may be used to define a variable in the text; Midas will generate a symbol of the form <#...A∅1>.

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

```
1)  DEFINITION                 CALL:  CLEAR TAB,1∅∅

    DEFINE CLEAR A,N/B         EXPANSION
           LAW A                      LAW TAB
           DAP B                      DAP...A∅1
    B,     DZM                 ...A∅1, DZM
           IDX B                      IDX ...A∅1
           SAS (DZM A+N               SAS (DZM TAB+1∅∅
           JMP B                      JMP ...A∅1
    TERMINATE

2)  DEFINITION                 CALL:  SAVEAC

    DEFINE SAVEAC /A           EXPANSION
           DAC #A                     DAC #...A∅1
           JSP SUBR                   JSP SUBR
           LAC A                      LAC ...A∅1
    TERMINATE
```

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

```
DAC #TEMP
JSP SUBR
LAC 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 I-C-3, Storage of Macro-Instructions.

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

In the pseudo-instruction <CHARACTER RA> the string <RA> satisfies the requirements for a legal symbol. Midas, however, understands its special meaning within the context of the pseudo-instruction. During the dummy symbol scan, however, Midas would interpret RA as a single symbol unless an apostrophe is used to indicate that <A> alone is a dummy symbol, as in

```
DEFINE MACRO A
       LAC (CHARACTER R'A
```

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.

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 arugments 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. 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 stored in 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.

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 macro name as the argument of a TERMINATE instruction. Then if the DEFINE associated with that TERMINATE refers to another macro name, the error print <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 M-I the use of apostrophe and the insertion of a supplied argument into a nested definition.

M-I

```
DEFINE FLOAT INSTR
  OLD'INSTR-INSTR
DEFINE INSTR X
  LAW X
  JDA F'INSTR
TERMINATE INSTR
TERMINATE FLOAT
```

For example, if FLOAT MUL appears, the expansion will be

```
OLDMUL-MUL
DEFINE MUL X
  LAW X
  JDA FMUL
TERMINATE MUL
```

This macro-instruction may be used to change PDP-1 instructions to subroutine calls. Their original meanings could be restored by

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

M-2

```
DEFINE MACRO X,Y
  LAC X
DEFINE MAC2 Y
  ADD Y
TERMIN
TERMIN
```

The call <MACRO ONE,TWO> will generate

```
LAC ONE
DEFINE MAC2 TWO
  ADD 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.

M-3 It is usually safer to use rather meaningless symbols as dummy arguments to avoid duplication of real arguments.

For example,

```
DEFINE MACRO X
  LAC X
DEFINE MAC2 COUNT
  ADD (X+3
  DAC COUNT
TERMIN
TERMIN
```

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

```
LAC COUNT
DEFINE MAC2 COUNT
  ADD (COUNT+3
  DAC COUNT
TERMIN
```

Example M-4 below illustrates a macro-instruction that redefines itself when first called.

M-4

```
DEFINE INCREM
  DZM X
DEFINE INCREM
  LAW 1Ø
  ADD X
  DAC X
TERMIN
TERMIN
```

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

```
DZM X
DEFINE INCREM
  LAW 1Ø
  ADD X
  DAC X
TERMIN
```

Subsequent calls will generate

```
LAW 1Ø
  ADD X
  DAC X
```

5. The Pseudo-Instructions IRP and IRPC

The pseudo-instruction IRP (indefinite repeat) generates sequential iterations of text a number of times determined by the 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 that, 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 represent a syllable 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.

IPRC 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 member of the list.

The following examples illustrate the use of IRP and IRPC. Notice that the second dummy symbol may be omitted if not referenced in the body, although its position must be retained by a comma.

```
I-1.  IRP [NAME,,RPA,PPA,TYO,TYI],[VALUE,,1,5,3,4]
          NAME=IOT VALUE
      ENDIRP
```

This IRP will effect symbol table entries for the listed instruction symbols and their corresponding IOT commands.

```
I-2.  DEFINE TYPE DIGIT
        IRPC [NUM,,Ø123...9]

        REPEAT 1IF VZ DIGIT-NUM,PRINTX /NUM/
        ENDIRP
      TERMINATE
```

Example I-2 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 subargument list until its equal is found and printed.

Example I-3 illustrates how to use the second dummy symbol, which represents a list.

```
I-3  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.

Example I-4 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
```

```
I-4.  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.

D. OPERATION OF THE MIDAS ASSEMBLY SYSTEM

1. Preparation of a Source-Language Program

The programmer prepares his source-language program on-line via Teletype terminal, using the symbolic editing program, Editor, to type in and edit text and to store the program in English-file format on a programmer's quarter-tracks.* The file is accessible to Midas by name and version number.

2. Performing an Assembly

a. Initial Procedure

English files to be processed must be on the drum for access by Midas. Editor and Handle are used to place files on the drum; Editor will enter files from the on-line Teletype or from paper tape; Handle, from magnetic tape. Midas runs under DDT control and is called in the following way. First, the user types C"F", which puts the Call program under DDT control. Call requests the file name (Midas) then asks whether or not to start the program. If "Y" is the response to this question, Call brings Midas into core and starts it running. If "N" is the response, Call simply brings Midas into core, and to start it, the programmer must type 1Ø1"G"; to continue it, 1ØØ"G".

*The programmer's quarter-tracks are described in Section IV-A

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 8. The macro table is empty. The symbol table contains all pseudo-instructions and a minimal list of PDP-1 instructions (these are listed in Appendix B). Additional instructions are available in an English file entitled, "System Midas Initial Symbols." Also available is a p-tape that contains a complete set of map macros and system IOT's. It is usually more convenient, however, for a programmer to construct his own p-tape of symbols that he is likely to use.

b. <u>The Control Language</u>

Commands to Midas are specified by control characters, such as 1 for do Pass 1 and C for continue the present pass. Some of the commands require arguments, such as the name of the file to be processed. Arguments, when required, are typed after the control character. Midas performs a requested function after line feed terminates the command string. Before the line feed is entered, an input string may be deleted by typing any 12.-bit character other than a carriage return. Spaces and carriage returns are ignored in a control string.

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

Control Characters	Function	Required Arguments
1	Initiate Pass 1	Name of English file followed by a comma and the version number.
2	Initiate Pass 2	Same as for 1.
C	Continue present pass on additional file.	Name and version number of additional file.
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. 16--print all characters processed 15--print an error comment on Pass 1 if the location goes indefinite 14--define undefined symbols as ∅ on Pass 2	An octal number
J	Add a jump block to assembled binary program. Selects address following last START encountered	None
H	Halt Midas	None

Control Character	Function	Required Arguments
B	Set up an indexed file of binary output in programmer's storage area.	Name and version number as required for file access.
S	Set up indexed file of symbol table and macro table.	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 two files would be accomplished by the following sequence of commands.

```
1  PROGRAMX,1
C  PROGRAMX,2
2  PROGRAMX,1
C  PROGRAMX,2
J
```

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. These commands are discussed below with regard to the order in which they can be useful.

S--Set up File of Symbol Table and Macro Table

The "System Midas Initial Symbols" file and available p-tapes are processed on Pass 1 as individual files in a multiple file assembly. Before continuing Pass 1 on his own program, the programmer may use <S>. Having set up these files in symbol-table format, the programmer need not perform Pass 2 on them. The file produced by an <S> command may be retrieved by command <T>.

<S> is generally also used at the end of an assembly to produce a file containing all program symbols. This file is read into DDT to permit symbolic debugging.

I--Initialize Symbol Table and Macro Table

When assembling a very long program whose symbols may exceed the symbol-table limit or when appending a file whose symbols may be defined in an earlier segment, a user may wish to file the current symbols and use <I> to initialize the symbol table before continuing. On Pass 2 the first table may be read in and the table reinitialized as necessary.

The faculty for initializing the symbol and macro tables also permits the programmer to assemble a new program without restarting Midas from DDT.

B--File Binary Output

<B> is normally used once during an assembly, at the end of Pass 2. However, if the programmer wants his binary output in two or more sections, he may use <B> to save a partial binary file and thus initialize the binary output buffer. Note that while filing and initializing are separate functions with regard to the symbol table, they are one with regard to binary output.

H--Halt

When an assembly is completed and the binary output filed, the user types <H> to halt Midas. Hitting the Break key returns control to DDT at any time.

E. 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 two bits of the first and second word indicate the type of block as follows:

Bits	Type
ØØ	Absolute
Ø1	Relocatable
1Ø	Library

The present manual will describe only Absolute Blocks.

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.

F. 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)
USW	1ØØØ	ALPHA+1	REPEAT	GAMMA

Column (1) contains a descriptive error code; (2), the octal address at which the error occurred; (3), the symbolic address stated in terms of the last address tag seen. Column (4) contains the last pseudo-instruction symbol or macro name Midas encountered. Column (5), used only in errors involving symbol definition, contains the offending symbol.

The error codes and the conditions with which they are associated are listed on the following pages, indicating action on or impossibility of continuation of the assembly.

Error Designation	Condition Causing Error	Action on Continuation
Undefined symbols		
USα	Undefined symbol (α indicates where found):	All undefined symbols are evaluated as zero.
C	in a constant	
D	in size of dimension array	
F	in OFFSET count	
I	in argument of ∅IF or 1IF	
L	in a location assignment	
M	in storage word generated by a macro call	
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 pseudo-instruction	
Relocation Errors		
IRα	Illegal relocation; α identifies where the error was found with designations as listed for undefined symbol	Relocation

Error Designation	Condition Causing Error	Action on Continuation
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
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

Error Designation	Condition Causing Error	Action on Continuation
Other Errors cont.		
LGI	Location gone indefinite	If the appropriate bit is set (by Midas control <E>), LGI is printed on Pass 1.
IRX	Illegal operation with relocatable symbols (e.g., logical operations)	
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 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)	

Error Designation	Condition Causing Error
TMV	Too many variables (The pseudo-instruction VARIABLES has been used more than 1∅ times in one program)
SCE	Storage capacity exceeded (symbol table and macro table full, too many constant words used)
IAE	An error which cannot be diagnosed--often due to improper operation of Midas

Appendix A. Midas Character Set

I. Alphabetic

Letters (A-Z)
Digits (Ø-9)

II. Punctuation

Character	Function(s)
,	a) indicates address tag mod (2^{12}) b) separates elements of a List c) terminates count of a REPEAT
:	indicates address tag mod (2^{16})
=	equates symbol to the left with expression to the right
/	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
$	indicates global symbol*
tab and carriage return	a) word terminators b) varying meanings according to context.

*Relocatable programming feature.

III. 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 $2^{18.}-1$
-	Addition of the one's complement

IV. Illegal

A. General

break

rubout

\

altmode

←

B. (<u>except within a macro-instruction or an IRP</u>)

!
"
"B"
"C"
wru
"F"
"G"
"H"
;
<
>
?
vert. tab
"K"
"N"
"O"
"P"
"S"
"V"
"W"
"Y"
"Z"

V. Ignored (except within a macro-instruction or an IRP)

'
EOT
formfeed
carriage return
↑

Appendix B. Symbols in Permanent Midas Vocabulary

1. PDP-1 Instruction Symbols

The following list contains all instruction included on the "MIDAS INITIAL SYMS FOR T.S." tape. Those included in the initial vocabulary are starred.

MIDAS INITIAL SYMS FOR T.S.

```
  1S=1
  2S=3
  3S=7
  4S=17
  5S=37
  6S=77
  7S=177
  8S=377
  9S=777
  AAI=740003
 *ADD=400000
 *AND=020000
  CAL=160000
  CLA=760200
  CLC=761200
  CLF=760000
  CLI=764000
  CLL=740010
  CLO=651600
  CMA=761000
  CMI=770000
  CML=740004
 *DAC=240000
 *DAP=260000
 *DCH=140000
 *DIO=320000
 *DIP=300000
 *DIV=560000
 *DZM=340000
  HLT=760400
 *  I=010000
  IAI=740002
  IDA=740400
  IDC=741000
 *IDX=440000
  IFI=742000
  IIF=744000
```

```
*IOR=Ø4ØØØØ
*IOT=72ØØØØ
*ISP=46ØØØØ
*JDA=17ØØØØ
*JMP=6ØØØØØ
*JSP=62ØØØØ
 LAI=76ØØ4Ø
*LAC=2ØØØØØ
 LAP=76Ø3ØØ
*LAW=7ØØØØØ
*LCH=12ØØØØ
 LFI=742Ø5Ø
 LIA=76ØØ2Ø
 LIF=7441ØØ
*LIO=22ØØØØ
*MUL=54ØØØØ
 NOP=76ØØØØ
 OPR=76ØØØØ
*RAL=661ØØØ
*RAR=671ØØØ
*RCL=663ØØØ
*RCR=673ØØØ
*RIL=662ØØØ
*RIR=672ØØØ
*SAD=5ØØØØØ
*SAL=665ØØØ
*SAR=675ØØØ
*SAS=52ØØØØ
 SCF=74ØØ4Ø
 SCI=74Ø1ØØ
*SCL=667ØØØ
 SCM=74Ø2ØØ
*SCR=677ØØØ
 SFT=66ØØØØ
*SIL=666ØØØ
*SIR=676ØØØ
 SKP=64ØØØØ
*SMA=64Ø4ØØ
*SNI=644ØØØ
*SPA=64Ø2ØØ
*SPI=642ØØØ
 SPC=74ØØØØ
 SPQ=65Ø5ØØ
 STF=76ØØ1Ø
*SUB=42ØØØØ
 SWP=76ØØ6Ø
```

```
*SZA=640100
*SZF=640000
 SZL=740020
 SZM=640500
 SZO=641000
*TAD=360000
 XAI=740001
*XCT=100000
*XOR=060000
  XX=HLT
```

2. Pseudo-Instructions

Symbol	Function
CHARACTER	inserts Internal Code for one character
CONSTANTS	specifies storage areas for constant words
DECIMAL	classifies integers as decimal numbers
DEFINE	initiates macro-definition
DIMENSION	allocates storage area for arrays
ENDIRP	ends an indefinite repeat
EQUALS	establishes symbol equivalence
EXPUNGE	erases symbols from symbol table
IRP and IRPC	initiates indefinite repeat
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
PRINT	generates symbolic location printout and prints comment during assembly
PRINTX	prints comment during assembly
REPEAT	generates iterative source-language text

START	denotes end of source program and specifies starting address
STOP	ends expansion of IRP's, macro's, and REPEAT's
TERMINATE	ends macro-definition
TEXT	inserts Internal Code for character string
VARIABLES	reserves space for variables and arrays
WORD	appends word(s) to binary output block
ØIF	tests an expression; if true, value is zero; if false, one.
1IF	if true, value is one; if false, zero.

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

INTERNAL	ASCII	CHARACTER
ØØ	Ø4Ø	SPACE
Ø1	Ø41	!
Ø2	Ø42	"
Ø3	Ø43	#
Ø4	Ø44	$
Ø5	Ø45	%
Ø6	Ø46	&
Ø7	Ø47	'
1Ø	Ø5Ø	(
11	Ø51	)
12	Ø52	*
13	Ø53	+
14	Ø54	,
15	Ø55	-
16	Ø56	.
17	Ø57	/
2Ø	Ø6Ø	Ø
21	Ø61	1
22	Ø62	2
23	Ø63	3
24	Ø64	4
25	Ø65	5
26	Ø66	6
27	Ø67	7
3Ø	Ø7Ø	8
31	Ø71	9
32	Ø72	:
33	Ø73	;
34	Ø74	<
35	Ø75	=
36	Ø76	>
37	Ø77	?

INTERNAL	ASCII	CHARACTER
4∅	1∅∅	@
41	1∅1	A
42	1∅2	B
43	1∅3	C
44	1∅4	D
45	1∅5	E
46	1∅6	F
47	1∅7	G
5∅	11∅	H
51	111	I
52	112	J
53	113	K
54	114	L
55	115	M
56	116	N
57	117	O (OH)
6∅	12∅	P
61	121	Q
62	122	R
63	123	S
64	124	T
65	125	U
66	126	V
67	127	W
7∅	13∅	X
71	131	Y
72	132	Z
73	133	[
74	175,176,∅33	EOM
75	135	]
76	∅15-∅12	CARRIAGE RETURN-LINE FEED
77		WARNING

INTERNAL	ASCII	CHARACTER
77ØØ	ØØØ	NULL or BREAK or "@"
77Ø1	ØØ1	"A"
77Ø2	ØØ2	"B"
77Ø3	ØØ3	"C"
77Ø4	ØØ4	EOT
77Ø5	ØØ5	"E" or WRU
77Ø6	ØØ6	"F" or RU
77Ø7	ØØ7	"G" or BELL
771Ø	Ø1Ø	"H"
7711	Ø11	TAB
7712	Ø12	LINE FEED
7713	Ø13	"K" or VT
7714	Ø14	"L" or FORM FEED
7715	Ø15	CARRIAGE RETURN (OUTPUT ONLY)
7716	Ø16	"N"
7717	Ø17	"O"
772Ø	Ø2Ø	"P"
7721	Ø21	"Q"
7722	Ø22	"R" or TAPE
7723	Ø23	"S" or RDR OFF
7724	Ø24	"T"
7725	Ø25	"U"
7726	Ø26	"V"
7727	Ø27	"W"
773Ø	Ø3Ø	"X"
7731	Ø31	"Y"
7732	Ø32	"Z"
7733	Ø33	"["
7734	Ø34	SHIFT "L"
7735	Ø35	"]"
7736	Ø36	"↑"
7737	Ø37	"←"
774Ø-7743	UNUSED	
7744	134	BACKSLASH
7745	UNUSED	
7746	136	↑
7747	137	←
775Ø-7773	UNUSED	
7774	177	RUBOUT
7775-7777	UNUSED	