MACRO

ASSEMBLY PROGRAM

INTERNAL OPERATIONS MANUAL

Prepared by: Robert A. Saunders
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td><strong>SECTION 1</strong></td>
<td></td>
</tr>
<tr>
<td>Input Tape Handler</td>
<td>5</td>
</tr>
<tr>
<td>Initialization and Title Sequence</td>
<td>6</td>
</tr>
<tr>
<td>Reset Sequence</td>
<td>7</td>
</tr>
<tr>
<td>Symbol Generator</td>
<td>7</td>
</tr>
<tr>
<td>Symbol Processor</td>
<td>8</td>
</tr>
<tr>
<td><strong>SECTION 2</strong></td>
<td>11</td>
</tr>
<tr>
<td>Storage Words</td>
<td>11</td>
</tr>
<tr>
<td>Location Assignments</td>
<td>11</td>
</tr>
<tr>
<td>Variables and Symbol Definition</td>
<td>12</td>
</tr>
<tr>
<td>Pseudo-instructions</td>
<td>14</td>
</tr>
<tr>
<td>Constants</td>
<td>17</td>
</tr>
<tr>
<td><strong>SECTION 3</strong></td>
<td>19</td>
</tr>
<tr>
<td>Macro Instructions</td>
<td>19</td>
</tr>
<tr>
<td>Macro Instruction Tables</td>
<td>20</td>
</tr>
<tr>
<td>Macro Instruction Definitions</td>
<td>22</td>
</tr>
<tr>
<td>Macro Instruction Usage</td>
<td>24</td>
</tr>
<tr>
<td>Macros Within Macros</td>
<td>25</td>
</tr>
<tr>
<td><strong>SECTION 4</strong></td>
<td>29</td>
</tr>
<tr>
<td>Error Alarms</td>
<td>29</td>
</tr>
<tr>
<td>Start Over Sequence</td>
<td>29</td>
</tr>
<tr>
<td>Symbol Package</td>
<td>29</td>
</tr>
<tr>
<td>Conclusion</td>
<td>33</td>
</tr>
<tr>
<td><strong>APPENDIX 1</strong></td>
<td></td>
</tr>
<tr>
<td>Macro Program Listing</td>
<td>35</td>
</tr>
<tr>
<td><strong>APPENDIX 2</strong></td>
<td></td>
</tr>
<tr>
<td>Macro Instruction Example</td>
<td>105</td>
</tr>
</tbody>
</table>
MACRO FIO-DEC is based on MACRO III, an assembly program for the TX-O computer at the Massachusetts Institute of Technology. The TX-O was built at Lincoln Laboratory and is now on loan to the Electrical Engineering Department at MIT. Since the PDP-1 is very similar in its logical design to the TX-O, it was thought worthwhile to prepare a version of the MACRO assembly program for use on the PDP-1. The program was written in MACRO language, and originally was assembled on the TX-O. An elementary version of DDT (see DECUS distribution MIT-2) was also prepared and was used in debugging MACRO. The present version incorporates a number of improvements over the original, and has been in use in its present form for several months at MIT.

The program is a two-pass assembler, with a macro-instruction facility which generates words from encoded stored model statements. With one minor exception, it is a linear scan character processor, examining each character once in order on each pass. In order to reduce wear and tear on input-output equipment, both input and output are buffered. The tape reading routine has an optional parity check, but except for this, and stripping the parity bits, the tape handling routines are essentially transparent to the rest of the program. We shall begin our discussion with an investigation of these routines.
SECTION 1

INPUT TAPE HANDLER

Each time the main program requires a character, `rch` is called. Characters are stored three to a word, and `fwd` is a counter which indicates which of the three characters is to be read out next. When a word is exhausted, the next is picked up at `rc8`, and saved in `fwb`. Normally, control drops through the tests immediately following, `fwd` is reset to 3, and the next character is stripped off at `rc1`. The character is saved in `t`, `rcp`, and the AC. The subroutine then returns to the main program.

When the last word is fetched, special treatment is necessary, for as will be seen later, it may not have three characters in it. The precise number is to be found in `nfc`, from which `fwd` is set when the program reaches `rc3`.

The next time through `rc8`, it will be found that no more words remain in the buffer, and control passes to `rfb`. The buffer indices are reset, and the program commences reading. Tape will be read until a stop code is encountered, a carriage return is encountered during filling the last 24 words of buffer, or a parity error is found. Deletes are filtered out, but all other characters are stored. Sense switch 6 is examined to see if parity is to be checked, and if it is off, parity is checked. The character is planted in a rotate instruction, which rotates according to the number of ones in the instruction. Thus, executing this on a word of alternate ones and zeroes generates a parity. If an error is found, a diagnostic is printed, and the character as read is displayed in the IO. The type symbol subroutine (`tys`) is used for typing. Continue causes the character to be accepted by going to `rfa`. Start ignores the character by returning to the read instruction (`rf2`). Note that the action on Start, if not otherwise conditioned by the test word, is determined by `sov`. This will be dealt with in detail later.

The characters are assembled into words directly into storage. The previous contents of the buffer words are lost by being shifted off the end of the word at `rf3`. Next we check for whether the remaining stop conditions are met. Stop codes go to `rf6`, where the last word has its characters correctly aligned for the readout routine. The end checks are set up, and control returned to `rc8`. If the buffer is within 24 (octal) words of being full, `rf4` is set to exit to `rf6` on the next carriage return. Since, in the usual MACRO-language
typescript, the next character after a carriage return is almost always an ignored tab, no great harm will be done if the reader cannot stop before the next character.

INITIALIZATION AND TITLE SEQUENCE

From ps2 to pte is initialization for starting or continuing a pass. Complete discussion of the initialization will mostly be confined to a general description, with specifics being related at the initialized routines.

The initial entry to the program is at ps5. The program stops at ps1-1, and on Continue goes through ps1, which sets for Pass 1; np1, which sets up to begin a pass; and through np2, which sets up to begin processing a single tape. At np2 is a sequence which detects whether there is a tape in the reader and the reader is turned on. An rpa is given without a wait, and if no character has appeared in the IO within about 80 milliseconds, the reader is assumed to be not ready and the program stops. When the reader is ready, the tape reading routine is initialized such that the buffer will appear completely empty, and tape will be read as soon as rch is called.

At pte, flag 5 is off iff (if and only if) a title is to be punched. If it is off, some blank tape is fed before anything else is done. Next the characters comprising the title are read. Leading stop codes are ignored; and also leading spaces, to prevent blank tape from being considered as spaces in the event that parity is not being checked. Leading carriage returns are also ignored. The first non-ignored character sets flag 6, so that spaces will no longer be ignored; and if the character is a middle dot, flag 5 is set to discontinue punching the title. The character is typed with completion requested but no in-out wait, and if the character is to be punched, this is done while the typewriter is typing. It has been found empirically that six lines can be punched during typing one character with negligible likelihood of the typewriter completion appearing before punching is done.

The carriage return following the title is detected at pt5, and when it has been found, pass 1 or pass 2 is typed out, followed by punching the input routine, if this is necessary. The input routine on the MACRO tape, as read into storage, is used as data. Some more tape is fed, and control passes to rst.
RESET SEQUENCE

The terminating character switches determine MACRO's treatment of the terminating characters tab, comma, equals, slash, and left parenthesis. The macro-instruction definition indicator \texttt{mii} determines the setting of these switches. If \texttt{mii} is on (-0), these switches are set to appropriate parts of the macro-instruction definition routine.

Indicators for each word are reset at \texttt{rsk} and \texttt{rsw}. At \texttt{rsk}, the left and right parenthesis switches are reset, and the dummy-symbol pushdown counter \texttt{prs} is set to 0. At \texttt{rsw}, the accumulated word value \texttt{wrd} is zeroed; the polysyllabic word indicator \texttt{syl} is turned off by clearing flag 5; the temporary storage \texttt{nsm}, \texttt{asa}, and \texttt{amn} is cleared (these are used by the slash routine for determining the symbolic location after a location assignment); the defined indicator \texttt{def} is turned on; and the dummy symbol indicator, flag 6, which is used by the macro definition routines, is turned off. At \texttt{sp}, the indicators for each syllable are cleared: the sign of the next syllable is set positive, the symbol letter indicator is cleared, and so are the overbar indicator, the syllable value \texttt{num}, the symbol storage \texttt{sym}, and the character counter \texttt{chc}. Control then falls into the main character processing loop, which begins at \texttt{r}.

SYMBOL GENERATOR

There are three kinds of symbols which are developed in the main character loop: integers, pseudo-instructions, and "symbols," which term we shall reserve for sequences of one, two, or three letters or numerals containing at least one letter. Letters and numerals are dispatched on at \texttt{r} and go to \texttt{l} and \texttt{n} respectively. Numerals are combined into \texttt{num} at \texttt{n}. The current radix control at \texttt{n} multiplies the preceding digits by eight or ten for octal or decimal. So that 777777 (octal) yields minus rather than plus zero, a check at \texttt{n} does a special treatment of zero. Letters turn on the letter indicator \texttt{let} and also letters-in-upper \texttt{liu} if in upper case. Letter and number flow combines at \texttt{n} where the character count \texttt{chc} is stepped and the first three characters are combined into a symbol \texttt{sym} at \texttt{l}. If a fourth character is encountered, \texttt{let} is checked; if a letter has occurred, it is a pseudo-instruction, and otherwise it is merely a number of four or more digits. Pseudo-instructions cause the P-I name to be saved in \texttt{api} for error printing purposes, and reset various indicators preparatory to picking up possible arguments. Additional characters are read until a break character (space, plus, minus, tab, or carriage return) is encountered, which ends the pseudo-instruction name, and the second three characters are saved in \texttt{syn}. At the break character, control is transferred to search
for the pseudo-instruction name at spm.

SYMBOL PROCESSOR

Symbols are combined by addition or subtraction as indicated by plus or minus signs, which go to \( p \) and \( m \) on dispatching. All routines which are called at the end of a symbol go to evl, which evaluates any symbol and performs the indicated arithmetic.

The symbol system is based on the idea that a symbol will be defined relatively infrequently, but will be used quite often. It is reasonable to spend a relatively long time defining a symbol if this will make it possible to evaluate it quickly. The symbol table is therefore kept sorted at all times, and a binary or logarithmic search is used to evaluate symbols. For those not familiar with the idea, the remainder of this paragraph is devoted to a discussion of the principle. Consider a dictionary, in which it is desired to locate a word, say pen. First look in the center of the book, and determine whether the word found there is before pen, after pen, or pen itself. If the word is before pen, which is likely to be the case, look next in the center of the back half of the book. Suppose the word found to be tree. Now pen is known to be before tree, so we next look in the center of the preceding quarter. The process is repeated, dividing the word list by two each time until the word is found. It is apparent that if there are two to the \( n \)th words, a maximum of \( n \) lookups are required, and the average number will be \( n-1 \).

To secure an alphabetic ordering of the symbol table, it is necessary to modify the codes of the letters so that the concise code is converted to alphabetic order. The easiest way to do this is by "inverting the zone bits," i.e., complementing the highest bit of each character if the next highest is a 1. This is done at the permute zone bits subroutine per, which also complements the sign bit. The transformation is reciprocal, i.e., permuting a permuted symbol un-permutes it. This fact is used by the error print routine.

Returning to evl, we see the symbol permuted, followed by a check of the macro-instruction indicator mii. If it is on, control is transferred to wsp to check for dummy symbols. If it is off, \( r \) is checked; if it is on, a symbol table search is necessary, otherwise the number (integer) is combined into wrd. It is also combined into amn, which accumulates the numeric part, if any, of a word for determining the new symbolic location in the event of a location assignment.
Location assignments are also dealt with at el, where the symbol, if any, to be used in a symbolic location is determined. There is a three state indicator nsm, which is initially +0, and is set to +1 after the first symbol of a word, and to -1 after any other symbol. It is also set to -1 in the event of a symbol preceded by a minus sign, for such a symbol cannot be the symbolic part of a symbolic location. Further discussion of this point will be postponed until a complete investigation of location assignments.

The logarithmic search begins at e2. There is a shift counter t1 which constructs the repeated increments to the address in the symbol table. The table is stored from register 7750 down, with the symbols in even-numbered registers and values in the next higher odd-numbered registers. Register 7750 is called low and contains lac the lowest address in the symbol table. The first location examined is that contained in low, and hence the lowest entry in the table. Succeeding addresses are computed as necessary, but the contents thereof are not examined until it is determined that the address does in fact lie in the symbol table. The decision as to whether to go up or down is seen to involve the overflow indicator (initially cleared at e2+ 2). This is a consequence of the fact that the symbols can assume all possible arithmetic values. Here the reason for complementing the sign bit becomes apparent. The table is arranged in numerical order, with the most negative number, originally the smallest positive number, at the bottom. It will be seen that if an overflow occurred, the sign of the result will be exactly the opposite of what it should be to move the search in the correct direction. Thus we do a skip on no overflow, and overflow causes a complement. Next we do a three way branch to move the search up, down, or exit on finding the symbol in the table. The remaining portion of the routine at eqat is related to variables and will be discussed later.

It will be seen that the maximum size of the symbol table must be a power of 2, since the shift counter is halved at each iteration and the search must always move an integral number of registers. The maximum corresponding to the initial value of the shift counter will never be realized in practice, for the symbol table would first collide with the top of the macro-instruction or constant table. The top of the latter tables is kept in register hih, and a collision results in an alarm of storage capacity exceeded.

Also in evl is a subroutine ed whose purpose is to frustrate the PDP circuitry that filters out minus zeroes on addition. Additions to wrd are done through this subroutine. This assures that when an expression such as (777776±1) appears in a source program, minus zero and not plus zero will be the result.
SECTION 2

STORAGE WORDS

The storage word termination routine places words in the punch buffer, counts the location counter and determines when punching should take place. Control is passed to the punch routine on Pass 2 whenever the location gets to a multiple of 100. This results in convenient sized binary blocks. There is a subroutine sch which checks syl and chc to see whether anything occurred since the last tab, carriage return or other terminator; if something has, the next instruction is skipped; otherwise the terminator is redundant and is ignored, since the next instruction returns control to r.

This routine is used as a subroutine by the macro-instruction processor and constant routine.

LOCATION ASSIGNMENTS

The location assignment character \(<\rangle\) enters at b. If preceded by a word terminator, it denotes the beginning of a comment, and control passes to itc to ignore characters until the next tab or carriage return. Otherwise, evl is called and the new location is set up. First the symbolic location is constructed according to the following rule: A symbolic location exists if the location can be expressed as symbol + number, where the number may be 0. In the event that the assignment is expressed as the sum of symbols, the old symbolic location, if any, is retained. If the assignment is purely numeric, asi is turned off (-0) and asm and amr are cleared, since asa and amn will contain zero. Otherwise, the alarm symbol indicator is left on (+0), and asm contains the symbolic part of the location, and amr the numeric part.

If, on Pass 1, a location assignment contains an undefined symbol, the location is considered indefinite, which fact is denoted by a negative number in loc. If the location is definite, loc is set from wrd at bnp. The location is taken modulo machine size, while the sign bit is preserved to retain whether or not the location is definite.

On Pass 2, an undefined symbol in a location assignment causes an alarm, but the location does not become indefinite, for the undefined symbol is simply ignored. If the assignment is defined, or on recovery from an alarm stop, wrd is taken modulo machine size and compared
with loc. If the two are identical, it is not necessary to start a new block, and the routine exits to bnp. If they are different, control passes to pun, with the new location saved in wrd while pun uses the old one to punch out the block.

At pun, the location is compared with the block origin to determine whether there are any words in the punch buffer. If there are not, it exits at once to bnp to set up the next block. It also exits if the punch indicator pun is off. If punching is to be done, the first and last address are punched, followed by the contents of the punch buffer, followed by a checksum which is the sum of all other words in the block. Register \( t \) is a counter which counts through the buffer, and the checksum is kept in ckl. Punching of each word is done by a subroutine pnb which displays the origin of each block in the AC as punching is done, enabling the operator to observe the progress of the assembly. Five lines of blank tape are punched at the beginning of each block.

After the block is completed, the new block origin is taken from wrd, where it was saved, and put into org. The punch buffer index ts is reset, and the routine normally exits to rnu.

VARIABLES AND SYMBOL DEFINITION

There are three basic ways to define symbols in MACRO: by parameter assignment, by address tag, and by variable definition. The appearance of a comma directs control to the address tag routine. If the location is indefinite, the routine exits at once; otherwise, evl is called. If the word preceding the comma is defined, its value is compared with the location counter; if they differ, an error is flagged at mdt. The symbol field on the error printout contains the tag if the tag consisted of one symbol; otherwise sym is cleared before the error is called. After return, or if the definition was correct, the new symbolic location is determined. In the event that the tag was polysyllabic, the old symbolic location is retained.

Should the word preceding the comma be undefined, the routine exits at once if the tag was polysyllabic; otherwise the symbol is defined at vsm, and the new symbolic location is determined as before.

Parameter assignments go to the parameter assignment routine at the occurrence of the equal sign. The expression to the left of the equal sign must consist of a single symbol which may
not bear an overbar. If these requirements are met, the symbol is saved in \texttt{scn} (which is also used by the macro-instruction processor), and the terminating character switches (\texttt{bt} for bar (slash), \texttt{at} for equal sign, \texttt{ct} for comma, \texttt{tt} for tab and carriage return) are set so that any terminator other than \texttt{bt} or \texttt{ct} causes an alarm. The routine then exits to \texttt{rnw} to await the expression for the value.

When the terminator occurs, the routine exits in the event nothing has appeared; and otherwise calls \texttt{evl}. If it is well defined, control passes to \texttt{q2} which saves the value, and then sets up indicators so that \texttt{evl} may be used to determine whether the symbol on the left of the equal sign was defined. If it was, the new value replaces the old one. If it was not, it is defined by \texttt{vsm} and the routine goes to reset. If the expression on the right was undefined, the attempted definition is ignored on Pass 1, and causes an error comment on Pass 2.

Variables are handled at \texttt{evl} by a variety of routines. The logic is that we must first have a symbol. If the symbol is defined, nothing further is done unless it has an overbar. If it is defined as \texttt{-0}, on Pass 1 we act as if it were really undefined and exit, and on Pass 2 we redefine it to the correct value which is the sum of the variables origin (as determined by the location of the pseudo-instruction variables on Pass 1) and the variables counter, which counts the different variables as they are defined. If it is defined as other than \texttt{-0}, on Pass 1 we give an error alarm (for this implies it was defined in a conflicting manner elsewhere), and on Pass 2 we ignore it, assuming that a previous occurrence has caused it to be defined correctly. Thus, on Pass 1, we go defining all variables as \texttt{-0}, and on Pass 2 we redefine them to their correct values as they occur. The scheme avoids requiring a separate list of variables, as they are stored in the main symbol table at all times, but has the disadvantage that the first appearance must have an overbar, or the variable will be incorrectly evaluated as \texttt{-0}.

The actual defining of symbols is handled by the \texttt{vsm} routine. Since the symbol table is maintained sorted at all times, \texttt{vsm} must locate the correct place for the new symbol and move all lower symbols down two registers to make room for it. The routine starts at the bottom of the symbol table and works its way up, using the overflow indicator in the same way that it is used in the logarithmic search. At the outset a check is made to see whether all of storage has been used; if it has, an error comment is made.
PSEUDO-INSTRUCTIONS

The pseudo-instruction system uses a form of list structure in the principal table, which begins at mai. There are two relevant registers, mai and psi, which contain indices to the table. From mai+1 to npi-1 are the system pseudo-instructions arranged in a three-entry table. The first two entries are the name of the pseudo-instruction and the last is the location to which control is to be transferred in the event one is found. Index psi is a pointer to the last pseudo-instruction name in the table. If there are macro-instructions defined, it points to the last macro name. At npi the macro storage begins. Each macro block begins with three registers, of which again the first two contain the name, but the third entry is now a pointer back to the beginning of the previous macro or pseudo name. These pointers contain law in the instruction part, and the negative sign is used to distinguish these pointers from pseudo-instruction locations. These considerations dictate the form of the search for the pseudo or macro name.

First we load the I-O with mdi, which is an indicator which is on (negative) if this name is that of a macro-instruction to be defined. Then we look at the last name defined, via the pointer psi. If the first three characters match, the second three are checked. If these match also, we either go to the mdm alarm if we are trying to define a macro of this name, or go to the appropriate routine. If the sign of the pointer is negative, we have a macro name, compute the beginning of the macro information storage and go to mac. If it is positive, the pointer addresses the location containing the location to which control is to be transferred.

If the first three match but the second three do not, it is recorded in flag 2 that at least one approximation to the correct name has been found, and the location is retained in sp5. The search is continued until either the correct name is found or the table is exhausted. If no name is found, and the name being searched is the name of a macro being defined, control passes to dmi, define macro instruction; if an approximation has been found, we go to the appropriate routine as before. If all the preceding fail, the name is undefined and causes an alarm atipi.

The various pseudo-instructions are fairly straightforward in their execution. Character and Flexo treat their arguments in an obvious manner. Text checks rqc, which is negative in the range of a repeat, and if it is off, sets up switches and picks up the terminating character,
which is saved in t2. Register t1 counts the characters in each word. Until the terminating character is matched, complete words are sent to the storage word routine, or to the storage word part of the macro processor if in a macro definition. When the terminator is matched, the last word is filled out with zeros (spaces) as necessary, and after it is disposed of, the routine exits through the storage word routine to rnw.

The pseudo-instruction Repeat sets all terminating switches to illegal format except comma, tab, and carriage return and then exits to pick up the count. The termination of the count goes to rql, which checks definiteness and for a positive or zero count. If all is well, the pointers for the readout of the flexo list are saved in private temporary storage, and carriage returns are arranged to trap. The routine exits to reset. Each succeeding carriage return is counted until the count runs out; until it does, the flexo pointers are restored to their old values and the character reader re-reads the characters. When the count runs out, the carriage return switch is restored and the routine exits. The reason Text is not allowed in a Repeat is to ensure that all characters required by the Repeat are in storage. Otherwise, rfb might have stopped reading tape on a carriage return in the Text (and therefore, inside the Repeat), and the trick of restoring the pointers would not work.

Start causes a complaint if it occurs in a repeat or macro definition and otherwise sets the terminating switches to pick up the starting address. The address termination returns to s, where on Pass 1 the program is stopped ready to begin Pass 2, and on Pass 2, if everything is definite, the address is saved and the punch buffer dumped. The origin for a continuation tape is set up from loc, and the program stops. Continue punches a start block if pch is on, preceded and followed by some blank tape. The program again stops, and Continue begins Pass 1 anew retaining all symbol definitions. The contents of soy control action on Start.

The variables pseudo-instruction is considered illegal if in a macro definition or in a region of indefinite location. Because of limited storage, variables may be used only once. If repeated usage were allowed, two entries would be required for each use; as it is, the two numbers are kept in va1 and va2 which are the beginning of, and the first free register after, the variables storage. Although a count of variables is kept on Pass 2, it is necessary to record the first free register, because in the event that the operator should desire to repeat Pass 2, the variables count would be zero as all variables would be correctly defined on the
first Pass 2. On Pass 2, a check is made to see that the pseudo-instruction location agrees with that found on Pass 1, and if it does not, there is an alarm. If all is well, a location assignment is simulated to leave room for the variables, and the program continues.

The pseudo-instruction dimension causes symbols to be defined as variables, with the variables counter being advanced according to the size of the array. Terminating switches are set up so that commas are ignored, left parens save the symbol in tcn (and check flag 5 to make sure only one symbol appeared), and right parens do all the work. The array size is evaluated and checked for definiteness. The saved symbol is then looked up. On Pass 1 control goes to di3 which, if the symbol is undefined, defines it as -0. On Pass 2, the correct definition is constructed. On both passes, the variables counter is suitably advanced and the routine exits. The terminators are restored when a carriage return or tab is encountered.

The pseudo-instruction constants is quite similar to variables in its operation. The values of the constants are stored in order in the macro-instruction table above the last macro definition, starting at a register whose address is kept in con. On Pass 1, the location is advanced according to the total usage of parenthesis operators, whether or not any identical constants occur, and the location of the beginning of the constants storage is saved in the first entry of the constants origin table. On Pass 2, the stored constants are dumped into the punch buffer via the storage word routine. There is no ambiguity as to how far to advance the location counter, as the number of parentheses, which is kept in nca, must be the same on both passes. The number of different constant values is determined by nco, which will generally be less than nca. Storing the constants on top of the macro definitions has both advantages and disadvantages. The primary advantage is economy of space in the assembler, for all of the available table space must be used before the tables collide, and any saving in one table is automatically available to the others. The major disadvantage is that an unnecessarily large block of space may be reserved for constants in the assembled program. To avoid this, it would be necessary to save the values of constants on both Pass 1 and Pass 2, leaving one register in the reserved storage area for each constant which is undefined at its appearance on Pass 1, plus whatever is required for the defined ones. Since in general there will be constants used before all the macros are defined, putting the constants on top of the macro table is not feasible in this scheme. The constants are placed in the constants table by the constant table search routine which will be discussed later.
Although it is not done here, it is quite possible to check for agreement of location of the pseudo-instruction constants on Pass 1 and Pass 2. If they disagree, it is clear that the result on the assembled program would be disagreeable, as all preceding constant syllables would have been incorrectly assembled. It should be pointed out that the second entry in the \textit{cor} table is set up on Pass 2 and is used only by the symbol package for printing out the constants areas.

\textbf{CONSTANTS}

Constants syllables are enclosed in parentheses. Left parentheses normally go to \texttt{lp}, and right parens go to \texttt{rt} from which they go to \texttt{rp} unless there is no matching left paren, in which case control goes to \texttt{ilf}. There is a four entry table (\texttt{cv1-cv4}) in which are stored the macro-instruction dummy symbol pushdown counter (described later), \texttt{wrd}, the sign preceding the left paren, and whether \texttt{wrd} is defined. There is a subroutine \texttt{pi} which handles the indices on the \texttt{cv} tables which is called here to move the pointers up one level. If the table overflows, control goes to \texttt{tmc} for an alarm. The first left paren saves all the terminating character switches in private temporary storage and sets them to go to the constant evaluating routine or \texttt{ilf}. In either case, control then goes to \texttt{rsw} to reset all storage associated with words and syllables. The value of the constant is then accumulated.

Right parens now go to \texttt{rp}, which evaluates the constant, and if not in a macro definition, calls \texttt{co} which files the constant in the constant list and returns the location in which it will be stored. The appropriate sign is applied, and the value is added to the previous value of \texttt{wrd}. Again \texttt{pi} is called, this time to move the pointers down one level. The indicators for syllables are then reset, and if the routine was entered from a right paren, the routine exits to process the next character in sequence. The word terminators comma, tab and \texttt{cr} also enter at \texttt{rp}, but when finished they go around again until the level is reduced to zero. The check for carriage return at \texttt{rp3} is a patch that was put in to fix a bug in the repeat logic.

When the level is reduced to zero, the terminating character switches are restored to their original values and the routine exits to the appropriate switch.

The \texttt{co} routine is straightforward. The constants appearance counter \texttt{nca} is stepped, and on Pass 1 the routine exits at once returning \texttt{-0}. On Pass 2 \texttt{def} is checked, and if any undefined
symbols appeared, an alarm is flagged. The search for a matching constant begins at the bottom of the constant table, to which con points. If a matching value is found, at co6 the position in the table is found, added to the current constant origin, and returned as the value of the syllable. If the search is exhausted unsuccessfully, the pointer to the top of the table nco is increased by one and, if there is any storage left, the new constant is added to the list. The value of the syllable is then constructed as before.

There is a fairly large amount of initialization for the constants routines at np1. The top of the macro instruction list is used to determine con, and nco points to it until there are constants in the table. The constants appearance counter nca is cleared, and the constant origin indices are set to zero. The pseudo-instruction constants also clears nca and nco and advances the constant origin indices.
MACRO INSTRUCTIONS

The macro instruction facility in MACRO is both the strongest and weakest part of the program. It is the strongest in the sense that it is that part of the program which contributes most toward ease of programming, especially in setting up tables of specialized format. It is the weakest in that it is quite inflexible and does not incorporate any of the more significant improvements in assembler technology that have occurred since the logic was first written in 1957.

There are two frequently used ways of organizing macro instruction storage: either the input characters comprising the definition are stored away, with dummy symbols usually marked in some special way, or the input characters are partially assembled, and the assembled words are stored with provision for inserting the dummy symbol values when the macro is used. The first scheme requires a relatively large amount of storage for macro definitions and has considerable complication in the treatment of dummy symbols if macro calls are permitted within macro definitions. However, the rest of the assembler can be used as a subroutine when the macro is called, and considerable flexibility is available in the use of dummy symbols, since an entire character string can be inserted as, say, part of a macro to print a message on the on-line typewriter. The second scheme realizes some economies in macro instruction storage, particularly if macro calls within macro definitions are relatively infrequent, and has a slightly less involved treatment of dummy symbols. The principal disadvantage is that dummy symbols can not supply other than numerical values to the compiled instructions without a large amount of involved coding. It is the second scheme which is used here.

Before delving into the mechanics of macro operation, we should consider some implications of macro calls within macros. Firstly, a macro definition within a macro definition is not allowed. Macro calls within macro definitions are allowed, and dummy symbols from the definition are allowed to be used in the macro call. A macro call cannot have any effect on the macro being defined except possibly to insert additional storage words into the definition. Thus it is not possible to have a macro call a macro which does nothing but, say, double an argument of the first macro. Calling a macro within a macro definition causes the data for the called macro to be re-copied into the data for the macro being defined,
with no change except such as may be required for the proper translation of dummy symbols. With this background, we can examine the macro processor in detail.

MACRO INSTRUCTION TABLES

The best place to start is with an examination of the macro-instruction table structure. The principal table is **main**. After the pseudo-instruction data, the first word is a code word consisting of code bits which are read from left to right. The other entities in the table are identified by these bits. The code combinations are as follows:

- 0 denotes a storage word.
- 10 denotes a dummy symbol specification.
- 110 denotes a constant.
- 1110 denotes a dummy symbol parameter assignment.
- 1111 marks the end of the macro definition.

Subsidiary combinations are used after these identifiers as necessary.

The order of entities is as follows: First will appear any relevant dummy symbol specifications. Next will appear one of the other entities, with which all of the dummy symbol specifications are associated. Parameter assignments and storage words are the lowest order, and they may include constants. If a storage word or parameter assignment contains constants, and both the word or assignment and the constants contain dummy symbols, the dummy symbols within each constant appear first, followed by the constant designator, followed by dummy symbols for the word or assignment, followed by the word or assignment data.

Each dummy symbol specification code bit pair is immediately followed by seven more bits which specify the dummy symbol sign and the dummy symbol number. The six bits for the number are written in reverse order. All these bits are written into the table by **sco** and **scz**, store code bit one and store code bit zero. The writing of the dummy symbol specification uses an additional routine **wro** which calls **sco** and **scz**. There is a corresponding routine **rro** which reads dummy symbol specifications.

Storage words store one additional bit which is zero or one depending on whether the word is zero or non-zero, respectively. If the word is non-zero, it is stored in the macro instruction table.
Constants and parameter assignments are very similar in that both have associated a value and a dummy symbol number. The value is treated as it is in storage words. The dummy symbol number is treated as in dummy symbol specifications, except that the sign bit is used to tell whether this is a new dummy symbol (denoted by a 0) or a redefinition of an old one (denoted by a 1). Constants behave like parameter assignments in that their effect is to define a new dummy symbol whose value will ultimately be the location of the stored constant.

The net result in the main table is an assortment of codewords and value words. The type of any particular word is determined by the preceding codeword in an elementary manner: the first word is a codeword, in which one writes bits until it is full; then one starts on a new codeword. Any value words which occur in the meantime are stored in order after the codeword, and the new codeword is put in the next available space. As there are routines for writing code bits, so is there a routine for testing them: tcb, which is used when a macro is called. Its operation will be considered later.

Also used by the macro processor is a set of erasable tables. First there is dsm, the dummy symbol table, which has the flexo codes of defined dummy symbols. Each dummy symbol has a number which is its position in this table. Dummy symbols are numbered sequentially in order of definition starting with R, which is always defined and is dummy symbol number 1.

Next there is dss, the dummy symbol specification table, which is used when defining a new macro-instruction in terms of an old one. The nth entry in dss gives the dummy symbol in the macro being defined corresponding to dummy symbol n in the one previously defined. The first entry is always 1, since dummy symbol R always transforms into itself. An entry of -0 means that there is no dummy symbol in the new definition corresponding to one in the old definition because the value of the old dummy symbol has been determined by some means; for example, if first A had been defined, and second had been defined as first 1, there is no dummy symbol in second corresponding to A in first, because A now has a definite value, i.e., 1.

Next in the list is dsv, the dummy symbol value table. It contains the values of all the dummy symbols when a macro instruction is used.
Finally there is pdl, the dummy symbol pushdown list. The pdl table is used to ensure that the order of dummy symbols fed into the mai table corresponds to that described above. Pointers to this list occur in cv]. As constant levels build up because of left parentheses, pointers in cvl mark the beginning of each level. When left parentheses reduce the level, all the dummy symbol specifications down to the next level are stored and a constant assignment defines a single dummy symbol on the lower level whose value is the location of the constant. The dummy symbol specifications in pdl are stored by prs, prepare specifications; and all specifications at any one level are stored in mai by ss, store specifications.

Since we have doubtless by now left the reader in a sea of confusion, without further ado we will enter into a description of how all this is done in the hope that some clarity may yet be achieved. The reader is advised to construct some macro definitions and examine the resulting mai table in an actual assembly for further examples of how all of this works. An example is given here in Appendix 2.

MACRO INSTRUCTION DEFINITIONS

The appearance of the pseudo-instruction define marks the beginning of a macro definition. Control passes to dfn, where the first test is for whether a macro definition is already in progress. If it is not, terminating switches are set so that equals and comma are illegal, slash for anything other than a comment is illegal, and tab or carriage returns other than redundant ones are illegal. The location counter is saved in tlo and zeroed. The symbolic location is killed, and the macro define indicator mdi is turned on. The macro instruction pointer is boosted to leave room for the pseudo-instruction information, and the routine exits to rmw to await the name of the macro being defined. When this has been read and checked for multiple definition (see Search for Pseudo-instruction), control passes to dmi. Here the name and other pseudo-instruction data is set up, but psi is not stepped as yet as recursive definitions are not allowed. The macro define indicator is turned off, and the macro instruction indicator is turned on. The dummy symbol counter is set to zero, the specification pushdown counter is set to zero, and the terminators are set to pick up dummy symbols. Dummy symbols terminated by tab and carriage return go to pdl and pds, respectively. Checks are made to see that legitimate dummy symbols are used, and if all is well, the dummy symbol is filed in the dummy symbol table at dd. The last dummy
symbol, followed by a carriage return, sets the define exit to go to reset terminating character switches. It is possible to check for duplicately defined dummy symbols, but it is not done in this version of the program.

Reset terminating character switches sets the switches to go to the appropriate macro definition routines. Dummy symbols appearing in expressions are detected at wsp, which is logically part of evl. Search for dummy symbol sds is called after the sign is set up, and the next instruction is skipped if the symbol is defined. Subroutine pr enters the specification for the dummy symbol in the dummy symbol pushdown list.

Storage word terminators (tab and cr) go to sw. If there are undefined symbols in the word, there is an alarm, otherwise, the alarm location and location counter are stepped and control goes to ss, which stores the dummy symbols from the pushdown list, and then to smb to store the word after the code bits are written. Final exit is to mw. Register tea is a temporary for subroutine exit addresses (hence the name).

The equal sign in a dummy symbol parameter assignment goes to da. If the symbol to the left of the equal sign is in good order it is saved in tcn and the terminators are set to pick up the expression for the value. The terminator traps to dal where the usual checks are made. The saved symbol is then looked up in the dummy symbol table. If it is defined, a negative sign is attached to flag this as a redefinition; otherwise dd is called to define a new dummy symbol. Note that sds returns the dummy symbol in the 10 where it is used by dd. Next mp is called, which writes the appropriate entries in the mai table. Final exit is to rst to reset the terminators.

Constants in a macro definition go to Ip and rp as before, but are treated differently at rp. Instead of calling co, control passes to rp8, which first calls mc to write a constant entry in the mai table, and then defines a new dummy symbol (whose flexo name is zero) whose number is used to complete the entry in the mai table. A specification for the newly created dummy symbol is written on the specification pushdown list, from which it will be filed in the mai table preceding the entry for the entity in which the constant has been used. After this, we go back to rp5 to move the pointers and restore the terminators if necessary.

The macro definition is ended by the pseudo-instruction terminate. This is illegal if not
in a macro definition. The location counter is restored, the symbolic location cleared, and
the macro-instruction indicator turned off. The pseudo-instruction index is set to include
the new definition, and four ones written into the codeword. The last codeword
is rotated around into the correct position and stored in the mai table. The routine then
exits to rst to set the terminating characters to normal assembly position.

To conclude this part of the macro definition procedure, let us turn to the code bit routines.
The two entries sco and scz both save the return address, and save the bit to be stored in
tc which cannot be in use at the same time. The bit counter scn is stepped, and until it
overflows, control goes to sc4 where the new bit is added to the current codeword which
is stored in scw. When a codeword overflows, it is stored in the mai table at sc3, and then
sm, store word in mai is called. It does not store anything useful, however; it merely is
used to locate the point in the mai table at which the NEXT codeword will be stored. The
reason for this is of course that the codeword must precede any value words which may be
associated with it. The liosc3 makes the code bit routine transparent to the IO, which
fact is used by wro.

MACRO INSTRUCTION USAGE

We will defer until later any discussion of macro calls within a macro definition. Assume
a macro has been called, and mii is off. The pseudo-instruction search routine goes to
mac, where the address of the first word of macro data, as determined by spm, is saved
in aw, which is the general pointer for reading out of the mai table. The terminating
switches are set to pick up the arguments (if any), and the dsv table is cleared. Control
now passes to r2 to pick up the arguments.

Commas terminating arguments go to ae1, from whence evl is called, and if the argument
is defined, its value is stored in the dsv table at ae4. The routine exits at ae6 until the
last argument is terminated, when control passes to am.

Assemble macro-instruction into program (am) reads and dispatches on the principal code-
bits. The codebit tester returns to one after the call if the codebit is a one, and goes to
the address in the AC if the codebit is a zero. Storage words go to awm. There are two
nested subroutines here: rw, read word, which gets the next word out of the mai table;
and ar, which checks the zero-nonzero codebit and calls rw if necessary. Note that rw leaves the number in the AC, the IO, and in t. It is added into wrd by the ed add routine, and if not in a macro definition, the complete word is filed in the punch buffer by the storage word routine.

Dummy symbol specifications go to as, where the dummy symbol number is read. The sign bit is saved in tc and used to set up the sign operation at as6. When not in a macro definition, the dummy symbol value is read next and added into wrd by ed. The routine then exits to am1 to read the next principal code bits.

Constants go to ac, where the value word is read and, if mii (which ar returns in the IO) is off, co is called and the location of the stored constant put in wrd. The new dummy symbol which represents this constant is then stored in the dsv table. The routine then exits to ami, which clears wrd. The expression in which the constant syllable was used will have a dummy symbol specification for the associated dummy symbol, and it is by this means that the correct value of the constant syllable will appear in the expression. This obtains complete generality with respect to usage of dummy symbols within and without constant syllables of arbitrary depth.

MACROS WITHIN MACROS

We are now prepared to deal with the question of macro calls within macro definitions. The macro being defined will in general have associated dummy symbols. The index to these symbols is saved in dsl as soon as control gets to mac. In addition to clearing the dsv table, we now clear the dss table in order to make the routines work in the event of unsupplied arguments, which are taken as zero. Now the arguments are picked up. These may contain dummy symbols, which by the time the terminator occurs, will have been entered on the pushdown list and will have set the dummy symbol indicator. If this has occurred, a new dummy symbol will be defined which represents the argument dummy symbol or symbols, and a parameter assignment will be written into the mai table to signify this fact by the routine at ae7. Furthermore, the number of this dummy symbol as it will be used in the macro being defined is entered in the dss table in the position corresponding to the dummy symbol used in the previously defined macro. If an argument contains no dummy
symbols, the \texttt{dss} entry is made \texttt{-0} to signify that no new dummy symbol need be included when reading specifications for old ones. The old dummy symbol may be said to be \textit{inactive}. Constant syllables appearing in arguments are treated as elsewhere: a new dummy symbol is defined whose value will be that of the constant. This is taken care of by the \texttt{lp} and \texttt{rp} routines as we have seen before. Note that this is done whether or not the constant syllable contains dummy symbols. After the arguments are completed, control goes to \texttt{am} as usual.

At \texttt{am}, we insure that the specification pointer is reset and start reading codebits. Storage words go to \texttt{mw} instead of \texttt{tb3} after reading out of \texttt{mai}, and thus get stored back into \texttt{mai} for the new definition. Arguments, after reading the sign and dummy symbol number, go through \texttt{as8} instead of skipping to \texttt{as5} and examine the \texttt{dss} entry. If it is zero, there is no new dummy symbol to worry about and the dummy symbol value is picked up as usual. If it is not zero, there is a dummy symbol, which has the proper sign applied and then is entered on the pushdown list. If the dummy symbol number is \texttt{1}, then the value is added into \texttt{wrd}, as this is the only way that the location counter as used in the macro being defined can get into the macro being read. If it is anything else, the dummy symbol value must \texttt{not} be added in at this point, for it will be included when the macro being defined is ultimately used. To see this, recall that 1) if the argument included dummy symbols, a dummy symbol assignment was written which included the value, and 2) if the argument did not include dummy symbols, the \texttt{dss} entry is zero and the value will be added here.

Constants go to \texttt{ac}, where, after reading the value, we call \texttt{mc} to rewrite the value for the new definition and then go to \texttt{ac1}. Here we read the associated dummy symbol number which we will then look up in \texttt{dss}. If the sign is positive, this is a new dummy symbol and \texttt{dd} is called; the new dummy symbol number is then entered in the \texttt{dss} table. If the sign is negative this is a dummy symbol redefinition and the old \texttt{dss} entry is examined to determine whether this dummy symbol was active before. If it was, nothing more need be done, as the old \texttt{dss} entry is correct; if it was not, a new dummy symbol must be defined. In any case we leave \texttt{cc} with an active dummy symbol. The new dummy symbol number is then written in the \texttt{mai} table to complete the constant entry, and we return to \texttt{ami}. It would appear that the dummy symbol value should be entered in the
dsv table, but in fact this is not necessary, as the dummy symbol will be referred to only once in whatever the constant is used in, and this reference will not refer to the dsv table since the corresponding dss entry is not 0 or 1. (See discussion of as above for elaboration of this point.)

Dummy symbol assignments read the dummy symbol value from the mai table, then enter it in the dsv table. If the dummy symbol defined includes no dummy symbols in its value, we go to aal where we clear the associated dss entry to signify this. If it does, we call cc as was done with constants to activate a suitable dummy symbol. A parameter assignment for this dummy symbol is then written into the mai table, and the routine exits to ami.

Encountering the code for the end of the macro definition restores the dummy symbol counter dsk to its old value, effectively undefining all dummy symbols associated with the called macro. Control then passes to rst to reset and continue with the definition.
SECTION 4

ERROR ALARMS

We have seen that a fairly large amount of error checking is done during the assembly process, and we should consider briefly the diagnostic routine. Most errors transfer control to an appropriate calling routine which determines the point to which to return, the particular routine to which to go, and the name of the error. The error routine proper has two entries, one for errors which print in the fifth field of the error listing and one for those which do not. The return point is put into sov and the name of the error picked up and printed out. Next the absolute location is printed if definite, or ind is printed if it is not. Next the alarm symbol indicator is tested, and if there is a symbolic location it is printed. Next the last pseudo-instruction used is printed. If there is a fifth field, it is printed at als. Completion of an alarm printout is followed by a carriage return. Next the test word is checked to see whether immediate continuation is desired, and if it is not the machine is stopped. Continuation returns to the appropriate routine. There is some extra coding to make sure that the columns line up correctly if the symbolic location or api fields are vacant.

START OVER SEQUENCE

The first routine in the program is the sequence that determines action on depressing the start key. We have seen that sov contains the address to which control is transferred on Start unless test word switch 0 is on. If it is on, the switches are placed in the IO and the first five registers of temporary storage are set in order to 1 or -0 depending on whether the associated switch is 1 or 0. If the continue pass bit was on, control goes to np2, otherwise control goes to ps1 or ps4 for Pass 1 or Pass 2, respectively.

SYMBOL PACKAGE

The symbol package is a six link chain. The routines sit in the temporary tables and use appropriate parts of the main program as necessary. The first link is symbol punch. If sense switch 1 is off or gets turned off, the routine exits to the input routine to read in the next link. If it is on, we first feed some tape and then listen for characters from the on-line typewriter. These are punched by the title puncher in the main program which returns control.
to ls. A tab termination goes to ls2 which listens for $s$ or $m$ for symbols or macros. If symbols are to be punched, sps-1 will have jmp sps which will punch the symbol table and then go to the macro puncher if flag 5 is off signifying macros are wanted too. If just macros are wanted, we go at once to the macro routine.

Both the symbol and macro punchers use the end subroutine which copies the appropriate storage into the punch buffer and transfers control to pun+6 when the buffer is full or the end of the macro or symbol table is reached. When punching a block is done, control returns to PCB+1. Flag 4 gets set on the last block, and finding it on causes the subroutine to exit through psx.

The macro punch will punch macros only if some have been defined. If some have, end is called. At the end of the job some blank tape is fed, followed by punching a start block. Some more tape is fed, and the routine goes back to the input routine.

The next link contains a text printing subroutine, the initial symbol table, and the constants area printer which will run if either switch 2 or switch 3 is on. A pointer to the cor table is checked to see whether any constants areas were designated, and if none were, the routine exits to the input routine. Otherwise, pss is checked, and constants origins are dumped on Pass 1, and the entire cor table on Pass 2. Flag 5 is used as a pass indicator. When finished, control returns to the input routine.

The alphabetic symbol print is the next link, which runs if sense switch 2 is on. It uses the symbol table and text printer which remain in storage from the preceding link. Since the symbol table is ordered alphabetically, the logic is simple enough. Each symbol is looked for in the initial symbol table, and if it is not there, it is printed out. When done, the heading for numeric symbol print is written if switch 3 is on, and then control goes back to the input routine.

The numeric symbol print is the most complex part of the symbol package. A floor register ($t1$) and a ceiling register ($t$) are kept, with the floor initially containing zero. Successive passes are made through the symbol table comparing the value words with the floor and ceiling. If a symbol is less than the floor, it is discarded, and if it is equal, it is printed out if not in the initial symbol table. If it is larger than the floor, it is compared with the ceiling and if it is greater, it is discarded. If it is less, the ceiling is set from the symbol value. Thus at the end of each pass, the floor represents the value of the symbols just printed, and the ceiling
represents the value of the symbol or symbols next in line to be printed. Therefore, the ceiling is moved into the floor and the ceiling is set to \(-0\) (777777), and the process is repeated until \(-0\) is found in the floor, which insures that all symbols have been printed.

Now let us follow the coding. Pointers to the initial symbol table \(sy_3\) and \(sy_4\) are set up, the ceiling \((t)\) is zeroed, and a carriage return typed. We then drop into the main loop. The ceiling is moved to the floor, \(-0\) put into the ceiling, and the symbol table pointers initialized. Now we start comparing values with the floor. Note that overflow will be a problem, for either number can vary over the whole range of values from 0 to 777777. Thus a simple subtraction will not yield a meaningful difference. Furthermore, it turns out not to be convenient to use the overflow indicator, which is better suited for use when the range of values is from 400000 (smallest) to 377777 (largest). Therefore we proceed in the following way. The numbers are \texttt{xor}'ed and the sign of the result examined. If it is positive, the numbers are of the same sign and a meaningful subtraction can be performed, and this is done at \texttt{sq1}. If it is negative, the number with the negative sign is the larger. In either event, going to \texttt{syi} discards the number, while going to \texttt{sq2} starts doing precisely the same sort of comparison with the ceiling. Identity between the floor and value goes to \texttt{syc} where the check against the initial symbol table is made.

At \texttt{syc} the symbol location is put into \texttt{syz} for printing purposes. Now the value is compared with the value of the present symbol on the initial symbol list. If they are equal, the symbols are compared at \texttt{syf}, and if these are equal also, this is an initial symbol and control passes to \texttt{syi}. If the initial symbol value is less than or equal to the symbol table value, the initial symbol table pointers are moved upward until this is no longer true. Note that the initial symbol table is arranged in numerical order. Thus it is not necessary to compare the symbol table symbol with all the initial symbols, but only with the next one which it is expected that will be found.

At \texttt{syi} the main symbol table pointers are moved up. When the top of the symbol table is reached, the floor is checked for \(-0\), and when this is found, the routine exits to the input routine after waiting for the last carriage return.

The next link in the chain is \texttt{restore}, called by sense switch 4. This routine resets the macro-instruction indices, then uses \texttt{vsm} and the initial symbol table to reconstruct the initial symbol table from scratch. When this is done, we go once again to the input routine to read the last link.
The final routine determines where to return control in the main program after the symbol package is done. If restore was run, control goes to ps5. Otherwise, pss and flag 6 are checked to return control to the appropriate place in the start routine, ready to begin or continue the assembly.
This completes our discussion of the MACRO assembly program. The version described here does not use sequence break and will run on any PDP-1. Enterprising programmers may wish to make changes to the routine to incorporate sequence break or make other improvements. It is hoped that this memo will facilitate this. We strongly suggest that no fundamental changes be incorporated, particularly those affecting the source language, for source language compatibility, and to a lesser extent, operating compatibility, are desirable goals. However, this should not be interpreted as ruling out any changes. We recognize that the program is not in any sense ideal or perfect. Nonetheless, it will give satisfactory service for its intended purpose.
APPENDIX 1

MACRO PROGRAM LISTING
MACRO FIO-DEC • part 1, 2-13-62

ncn=10  nfw=200  nds=30  ncd=20  ncl= 0

4240/
  pbf, /punch buffer
  pbf+101/ flx, /flexo input buffer
  flx+nfw/ dsm, /dummy symbols
  dsm+nds/ dss, /argument translation indicators
  dss+nds/ dsv, /m-1 argument values
  dsv+nds/ pdl, /dummy symbol specifications
  pdl+ncd/ cv1, /constants dummy symbol levels
  cv1+ncl/ cv2, /constants value levels
  cv2+ncl/ cv3, /constant signs
  cv3+ncl/ cv4, /constants definite on this level
  cv4+ncl/ cor, /list of constant origins
  cor+ncl+1/ cr2, /second constants origin
  cr2+ncl+1/ ck1, /checksum
  ck1+1/ org, /block origin
  org+1/ psi, /pseudo instruction index
  psi+1/ hai, /macro instruction storage
  7750/ low, /symbol table end

define
  error ROU,RET,NAM
  law RET
  jda ROU
  NAM
  terminate

 0/
/start over entry

    lat
    sma
    sov,  jmp xy

    so1,  swap
    init so3,pss

    so4,  ril 1s
           clc
           spi
           law 1
    so3,  dac xy
           index so3,(dac pss+5,so4
           lac npa
           sma
           jmp npa

    so5,  lac pss
           spa
           jmp ps1
           jmp ps4
/reset terminating character switches

rst, law rsk
dap rsx
lio mii
init bs, rnw.
init ct, c
init dtb+57, lp
spi
jmp rsm
dio mdi
init bt, b
init qt, q
law tab
jmp rsi

rsm,
init bt, df2
init qt, da
law sw
rsm

dap tt
rsx, jmp xy

/rst to convert next word

rsk,

rnw, init lp1, cv1
init prs, pdl
init rt, ilf

rsw, dzm wrd
clf 5 /syl
dzm nsm
dzm amn
dzm asa
clf 6 /dsi
law 1
dac def
law r

rss, lio (opr

sp,
dio sgn

dap spx
dzm let
clf 4 /liu
dzm ovb
dzm num
dzm sym
dzm chc

spx, jmp xy
/read and dispatch on one character

r,
    jsp rch
    add (dtb
dat .+2
    clc
    jmp xy

/re-dispatch on last character read

r2,
    lac rcp
    jmp r+1

/dispatch table

<table>
<thead>
<tr>
<th>dtb,</th>
<th>jmp p</th>
<th>jmp n</th>
<th>/space, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp n</td>
<td>jmp n</td>
<td>/2, 3</td>
<td></td>
</tr>
<tr>
<td>jmp n</td>
<td>jmp n</td>
<td>/4, 5</td>
<td></td>
</tr>
<tr>
<td>jmp n</td>
<td>jmp n</td>
<td>/6, 7</td>
<td></td>
</tr>
<tr>
<td>jmp n</td>
<td>jmp n</td>
<td>/8, 9</td>
<td></td>
</tr>
<tr>
<td>jmp il</td>
<td>jmp r</td>
<td>/, stop code</td>
<td></td>
</tr>
<tr>
<td>jmp il</td>
<td>jmp il</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp il</td>
<td>jmp il</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp il</td>
<td>jmp il</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp n</td>
<td>bt, jmp</td>
<td>/space, +</td>
<td></td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/s, t</td>
<td></td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/u, v</td>
<td></td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/w, x</td>
<td></td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/y, z</td>
<td></td>
</tr>
<tr>
<td>jmp il</td>
<td>jmp cqt</td>
<td>/i, comma</td>
<td></td>
</tr>
<tr>
<td>jmp r</td>
<td>jmp r</td>
<td>/color</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tt,</th>
<th>jmp il</th>
<th>/tab</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp il</td>
<td>jmp l</td>
<td>/middle dot, j</td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/k, l</td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/m, n</td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/n, p</td>
</tr>
<tr>
<td>jmp l</td>
<td>jmp l</td>
<td>/q, r</td>
</tr>
<tr>
<td>jmp il</td>
<td>jmp il</td>
<td></td>
</tr>
<tr>
<td>jmp pm</td>
<td>jmp rt</td>
<td>/,, ,</td>
</tr>
<tr>
<td>jmp ovr</td>
<td>jmp lp</td>
<td>/,(</td>
</tr>
</tbody>
</table>

| jmp il        | jmp l | /a |
| jmp l         | jmp l | /b, c |
| jmp l         | jmp l | /d, e |
| jmp l         | jmp l | /f, g |
| jmp l         | jmp l | /h, i |
| jmp rcd       | jmp rl | /l. c., period |
| jmp rcu       | jmp il | /u. c., backspace |
| jmp il        | dtc, jmp tt | /car ret |

rcu,
    stf 3
    jmp r

rcd,
    clf 3
    jmp r
/case dependent characters

cqt, szf 3
qt, jmp q
c, jmp c
pm, szf 3
jmp p
jmp m

/process alphabetic or numeric character

l, dac let
 szf 3 /cas
 stf 4 /liu
 jmp ln

l2, lac sym
 ral 6s
 ior t
 dac sym
 jmp r

n, law 17
 and t
 dac t1

n2, lac num
 ral 3s
 xct .+1

n1, xx /opr=octal, add num=decimal
 dac num
 add t1
 sza
 jmp n3
 lac t1
 xor num

n3, dac num

ln, idx chc
 sub (3
 spq
 jmp 12
 lac let
 sma
 jmp r
 dzm num
 dzm let
 dzm chc
 stf 5 /syl
 lac sym
 dac api
/read three more characters for p-i or m-i

lac t
dac syn
setup t1,3
jap rch

ln4,

sza i
jmp spm /space
sad (54
jmp spm /minus
sad (36
jmp spm /tab
sad (77
jmp spm /cr
sad (35
jmp rch+1 /color change

ln3,

isp t1
jmp .+2
jmp rch+1
lac syn
ral 6s
ior t
dac syn
jmp rch+1

/over bar indicator

ovr, law 1
dac ovb
jmp r
/search for pseudo or macro instruction

spm, clf 2
  lac psi
  lio mdi

sp2, dap sp1
  lac sym

sp1, sad
  jmp sp3
  idx sp1

sp7, idx sp1
  lac i sp1
  spa
  jmp sp2
  law i 5
  add sp1
  sas (sad mai-2
  jmp sp2
  sp1
  jmp dmi
  szf 2
  jmp sp4
  jmp ipi

sp3, stf 2
  idx sp1
  dap sp5
  lac syn

sp5, sas
  jmp sp7
  sp1
  jmp mdm

sp4, idx sp5
  dap sp8
  lac i sp5
  sas

sp8, jmp 1
  idx sp5
  jmp mac
/address tag routine (comma)

c,            lac loc
          spa
          jmp rnw
          jsp evl
          spi
          jmp c1
          lac loc
          sad wrd
          jmp c2
          szf 5
          dzm sym
          jsp mdt

/c2,           szf 5
              jmp rnw

c3,           dzm asi
          dzm aml
          move sym, asm
          jmp rnw

c1,           szf 5
          jmp rnw
          lac loc
          dac t3
          jsp vsm
          jmp c3

/def in io on return
/syl
/parameter assignment (equal sign)

q,
  lac let
  szf 5
  jsp ipa
  sza i
  jsp ipa
  lac ovb
  sza
  jsp ipa
  lio sym
  dio scn
  init bt, ilf
  dap qt
  dap ct
  init tt, qq
  jmp rrw

qq,
  jsp sch
  jmp rst
  jsp evl
  spi i
  jmp q2
  spq
  jmp rst
  jsp usq

q2,
  lio scn
  dio sym
  move wrd, scn
  clc
  dac let
  law 1
  dac def
  jsp evl
  lac def
  spq
  jmp q1
  lac scn
  dac i ea
  jmp rst

q1,
  move scn, t3
  jsp vsm
  jmp rst

sch,
  dap sck
  szf 5
  jmp .+3
  lac chc
  szm
  idx sck

sck,
  jmp xy
/evaluate syllable and accumulate word value

evl,       dap ex
         lac sym
         jda per
         dac sym
         lac mii
         spa
         jmp wsp

ev2,      lac let
         spa
         jmp el
         add num

sga,      xct sgn
         add amm
         dac amn
en,       lac num
sgn,      xx
         jda ed

ev1,      lac pss
         lio def
ex,       jmp .

ndf,      clc
         dac def
         dac t3
         jda ed
         lio sym
         dio lus
         lac ovb
         sub pss
         sas one
         jmp evx
         jsp vsm
         idx vct
         jmp evx

e1,       lac sgn
         sad (opr
         jmp e11

e12,      law i 1
         dac nsm
         jmp e2

e11,      dac nsm
         szm
         jmp e12       /if +1
         sza
         jmp e2       /if -1
         law i
         dac nsm
         move sym, asa
/evaluate symbol (logarithmic search)

e2,     law 4000
dac t1
clo
lac low
jmp e1+1

edn,   lac (sub
dip e1
lac t1
rar 1s
dac t1
sad (1
jmp ndf
lac ea

e1,     t1
dac ea
sub low
spa
jmp eup
lac ea
sub (lac low-1
sma+sza-skp
jmp edn

ea,     lac .
sub sym
szo
cma
sma+sza-skp
jmp edn

.eqt,   sza
jmp eup
idx ea
lac i ea
dac num
lac ovb
sza i
jmp en
lac num
lio pss
cma
sza
jmp evk
spi
jmp ndv
lac vct
add vci
dac num
dac i ea
idx vct
jmp en
eup, lac (add
jmp edn+1

ndv, clc
dac def
move sym, lus
jmp en

evk, spi i
jmp en
move sym, lus
error alu, en, flex mdv

ed, 0
dap edx
lac ed
add wrd
sza
jmp ed1
lac ed
xor wrd

ed1, dac wrd
edx, jmp xy
/insert symbol in symbol table

vsm, 
    dap vsx
    law i 2
    add low
    dac low
    dap v1
    add one
    sad hih
    jsp sce
    clo

vs1, 
    lac v1
    dap v2
    add one
    dap v4
    add one
    dap v1
    add one
    dap v3
    sas (lio low+1
    jmp vs2

vs3, 
    lac sym
    dac i v2
    lac t3
    dac i v4

vsx, 
    jmp xy

vs2, 
    lac i v1
    sub sym
    szo
    cma
    spq-i
    jmp vs3

v1, 
    lio xy
    jmp vs3

v2, 
    dio xy
    jmp vs2

v3, 
    lio xy
    jmp vs3

v4, 
    dio xy
    jmp vs1

/low+2+i
/low+1+i
/low+i
/low+3+i
/pseudo-instruction repeat

rpt, lac rqc
spa
jsp irp
init bt,ilf
dap qt
init ct, rq1
dap tt
jmp rsk

rq1, jsp evl
spi
jsp usr
lac wrd
spq
jmp rq4
cma
dac rqc
init dtc,rq2
move fwd,rqx
move rc8,rqy
move fwb,rqz
jmp rst

rq2, count rqc,rq3
init dtc,tt
jmp tt

rq3, move rqx,fwd
move rqy,rc8
move rqz,fwb
jmp tt

rq4, sza
jmp irp
jsp rch
sas (77
jmp rch+1
jmp rst

irp, error alm, rq4+2, flex ilr

rqc, 0
rqx, 0
rqy, 0
rqz, 0
/pseudo-instruction character

ch,  
  jsp rch
  lio (rar 6s
  sad (51
  jmp ch1  /r
  lio (opr
  sad (44
  jmp ch1  /m
  lio ch2
  sas (43
  jsp ilf  /l

ch1,  
  dio ch3
  jsp rch

ch2,  
  ral 6s

ch3,  
  xx
  dac num
  jmp r

/pseudo-instruction flexo

fx,  
  dzm num
  setup t1,3
  jsp rch
  lac num
  ral 6s
  ior t
  dac num
  count t1,rch+1
  jmp r
/pseudo-instruction text

txt, lac rqc
spa
jsp ilf
load txv,law txq
init txx, rch+1
jsp rch
dac t2

txq, dzm wrd
setup t1,3

txw, jsp rch
sad t2
jmp txk

tax, lac wrd
ral 6s
ior t
dac wrd
isp t1

txx, jmp xy

txv, xx
dap bs
lio mii
spi
jmp mw
jmp tb3

txk, load txv,law rnw
init txx, txa
init bs, rnw
lac t1
sad (-3
jmp rnw
dzm t
jmp txa
/syllable separation characters (plus, minus, space)

p, 
  jsp sch
  jmp r

m, 
  jsp evl
  stf 5
  lac t
  lio (opr
  sza i
  jmp md
  szf i 3
  lio (cma

ml, 
  law r
  jmp sp

(relative address syllable (.)

rl, 
  lac chc
  lio sgn
  sma
  lio (opr
  dio rl3
  lac loc

rl3, 
  xx
  add wrd
  dac wrd
  stf 5
  lac mii
  sma
  jmp r
  rir 9s
  law 10
  rcr 3s
  jda pr
  jmp r


/storage word termination characters tab and carr ret)
tab,
  jsp sch
  jmp rrw
  jsp evl
  spi+sma-skp
  jsp ust
tb3,
  idx aml
tb4,
  idx loc
tb2,
  lac wrd
ts,
  dac .
  idx ts
  lac loc
  dac wrd
  and (77
  szm
  jmp bs
  lac pss
  spq
  jmp bnp
  jmp pun
/location assignment termination character
b1,
  lac def
  sma
  jmp bnp
  lac (400000
  jmp b3
b,
  jsp sch
  jmp itc
  jsp evl
  lac nsm
  sad (-1
  jmp bai
  dzm asi
  lio (-0
  sza i
  dio asi
  move asa, asm
  move amn, aml
ba1,
  lac pss
  spq
  jmp b1
  lac def
  spq
  jmp usb
b5,
  law 7777
  and wrd
  dac wrd
  sad wrd
  jmp bs
start
Macro FIO-DEC part 2

/punch binary block

pun,    lac org
    sad loc
    jmp bnp
    lac pch
    spq
    jmp bnp
    cli
    repeat 5, ppa
    lac org
    add (dio
    dac clk
    jda pnb
    lac loc
    add (dio
    jda pnb
    load t,dac pbf

pub,    lac i t
    jda pnb
    lac i t
    add clk
    dac clk
    idx t
    sas ts
    jmp pub
    lac clk
    add loc
    add (dio
    jda pnb

/form origin for next block

bnp,    lac wrd
        and (407777
        dac org

b3,      dac loc
        init ts, pbf

bs,      jmp .

loc,     0
/pseudo-instruction start

sta,  lac mi1
io rqc
spa
jsp  ils
init bt,11f
dap qt
dap ct
init tt,s
jmp r2

s,  lac pss
spa
jmp 1st
jsp evl
spl
jmp uss

s2,  move wrd,tcn
init bs,s4
move loc,wrd
jmp pun

s4,  init sov,np2
hlt+cla+cli+clf+6-opr-opr-opr
lac pch
spa
jmp s6
law i 40
jda fee
lac tcn
add (jmp
jda pnb
law i 240
jda fee

s6,  init sov,np2
lio (-0
hlt+clc+stf+6-opr-opr
jmp ps1

1st,  init sov,np2
hlt+cla+cli+stf+6-opr-opr-opr

//  pss   flg 6  tag
//   -0    0  s5
//    1    0  s4
//   -0    1  1st
//    1    1  s6
/initialize for new pass

ps2,  
    law 1
    dac pss
    dac pch
    dac tit
    move ini,inp

ps4,  
    move psb,psi
    lac mai
    move psa, mai
    jmp np1

ps5,  
ps3,  
    move mai,psa
    move psi,psb

s5,  
    init sov,ps2
    clc
    dac pss
    hlt+cli+clf+6-opr-opr

ps1,  
    clc
    dac pss
    dac pch
    law 1
    dac ini
    move psi,psb
    lac mai
    dac psa

np1,  
    dac hih
    add (sad-lac+1
    dac con
    dac nco
    dzm nca
    dzm asi
    law 4
    dac org
    dac loc
    law 1
    dac mii
    dzm vai
    dzm vct
    load n1, opr
    init cn6,cor
    init cn7,cr2
np2, load t, -4000
rpa-i
spi i
jmp .+5
isp t
jmp .-3
hlt+clc+cli-opr-opr
jmp np2
dzm api
dzm fwd
init ts,pbf
init rc8,flx+nfw+2
dzm rqc
init dte,tt
clc+clf 7+cli-opr-opr
add pss
add pch
add tit
sas (3
stf 5
/print and punch title

pte,   law i 40.
szf i 5
jda fee
jmp pt1+i

pt1,   lot 1
jsp rch
sad (13
jmp rch+i
sza
jmp pt0
szf i 6
jmp rch+i

pt0,   sad (77
jmp pt5
stf 6
sad (40
stf 5
ral 1s
add (ftp
dap pt2
dap pt3
idx pt3

pt1,   liot
iot 4003
szf 5
jmp pt1

pt2,   lac.
repeat 3, jda pt6

pt3,   lac.
repeat 3, jda pt6
jmp pt1

pt6,   0
dap pt7
lac pt6
cli
rcl 6s
ppa

pt7,   jmp .

pt5,   szf i 6
jmp pt1+i
dzm tit
/print pass 1 and 2

pps, jsp spc
lac (723554
jda tys .
jsp spc
lac (flex pas
jda tys
\yo
jsp spc
law 1
add pss
jda tys
law 3477
jda tys

/black carret

/punch input routine

law i 1
add pss
add pch
spq
jmp rst

pf2, law i 40
jda fee
lac inp
spq
jmp rst

p12, load pt6,dio 7751

p13, lac pt6
jda pnb
lac i pt6
jda pnb
index pt6,(dio 7776,p13
lac (.jmp 7751
jda pnb
dzm inp
jmp pf2

spc, dap .+3
cli
ty0
jmp .
/pseudo instruction terminate

ter, lac mii
spq-i
jsp ilf
lac tlo
dac loc
clc
dac asi
law 1
dac mii
lac dm3
dap psi
jsp sco
jsp sco
jsp sco
jsp sco
lio scw
jmp .+2
ril is
isp scn
jmp -.2
dio i sc3
jmp rst
/pseudo instruction define

dfn,
  lac m1
  spq
  jsp ilf
  law ilf
  dap qt
  dap ct
  law df1
  dap tt
  law df2
  dap bt
  lio loc
  dio tlo
  dzm loc
clc
dac asi
dac mdi
  idx mai
  dap dm3
  idx mai
  dap dm1
  idx mai
  dap dm2
  sub low
  sma
  jmp sce
  jmp rnmw

df1,
  jsp sch
  jmp r
  jsp ilf

df2,
  jsp sch
  jmp itc
  jsp ilf
/define macro instruction

dmi, lio sym
dm3, dio.
   lio syn
dm1, dio.
   clc+clf 4-opr /liu
   clf 5 /syl
   dac mii
dzm sym
dzm scw
   law 1
dac mdi
   lac psi
dm2, dac.
   idx mai
dap sc3
   law i 23
dac scn
   init prs, pd1
   init dsk, dsm+1
   init ddx, rsk
   init ct, pd1
   init tt, pds
   jmp r2

/pick up dummy symbol

pds,  law rst /tab
dap ddx
   lac chc
   spq
   jmp rst

pd1,  lac sym /comma
   jda per
   dac sym
   szf 5 /syl
   jmp pd2-1
   lac let
   sza 1
   jmp pd2-1
   szf i 4 /liu
   jsp ids

pd2,  lio sym
   jmp dd+1
/search for dummy symbol

sds, 0
dap sdx
dap sdy
idx sdy
init sd1,dsm
sd2, lac sds
sd1, sad xy
jmp sd4
index sd1,dsk,sd2
lio sds
sdx, jmp xy
sd4, lac sd1
sub (sad dsm-1
sdy, jmp xy

/define new dummy symbol

dd, dap ddx
dio i dsk
idx dsk
sad (sad dsm+nds-1
jsp tmp
sub (sad dsm
ddx, jmp .

/macro instruction constant

mc, dap tea
dzm num
stf 6 /ds1
jsp ss
jsp sco
jsp sco
mca, law smb
jmp scz

/macro instruction storage word

sw, jsp sch
jmp rmw
jsp evl
sma+spi-skp
jsp usm
sw2, law rmw
mw, dap tea
idx aml
idx loc
law mca
jmp ss
/dummy symbol assignment

da,  szf 1 4
jsp ilf
szf 5
jsp ipa
lac sym
jda per
dac tcn
init bt,ilf
dap qt
dap ct
init tt,da1
jmp rnw

da1,  jsp sch
jmp rnw
jsp evl
sma+spi-skp
jsp usd
da3,  lac tcn
jda sds
jmp dab
add (400000

da,  jda mp
jmp rst

mp,  0
dap mpx
jsp ss
jsp sco
jsp sco
jsp sco
jsp sco
jsp scz
init tea,mp1
jmp smb

mp1,  lac mp
jda wro

mpx,  jmp xy

dab,  law dab
jmp dd

/iff undef
/macro instruction usage

mac,      dap aw
       move dsk,dsl
       init bt,ilf
       dap qt
dzm tcn
       init tt,aev
       init ct,ae1
       init ae6,rsk
       init ae4,dsv
       clear dsv,dsv+nds-1
       lac loc
dac dsv
       lac mii
       sma
       jmp r2
       clear dss+1,dss+nds-1
ma1,     jmp r2

/evaluate macro instruction arguments

aev,     init ae6,am
ae1,     jsp evl
          sma+spi-skp
          jsp usp
ae3,     idx ae4
          add (dss-dsv
          dap ae5
          sad (dio dss+nds-1
          jsp tmp
          lio wrd
ae4,     dio xy       /dsv
          szf i 6       /dsi
          jmp ae5-1
          lac mii
          spq
          jmp ae7
          clc
ae5,     dac xy
ae6,     jmp xy
ae7,     cli
          jsp dd
          dac i ae5
          jda mp
          jmp ae6
assemble M-I into program

am,
  lac pss
dac def
init prs,pdl
ami,
  clf 6
  dzm wrd
aml,
  law awm
  jda tc
  law as
  jda tc
  law ac
  jda tc
  law aa
  jda tc
am5,
  lac dsl
dap dsk
  jmp rst

assemble M-I storage word into progr. or mai

awm,
  law aw3
ar,
  dap ary
  law ar5
  jda tc
  law ar1
rw,
  dap rwx
aw,
  lio xy
  idx aw
dio t
  lac t
rwx,
  jmp xy
ar1,
  jda ed
ar5,
  lio mii
ary,
  jmp xy
aw3,
  law ami
  spi
  jmp mw
dap bs
  jmp tb3
assemble argument (dummy symbol) into M-I word

as,
  jsp rro
  add (dsv-1
  dap as5
  add (dss-dsv
  dap as8
  and (777000
  dac tc
  lio (cma
  sma
  lio (opr
  dio as6
  lio mii
  spl i
  jmp as5

as8, lac xy /dss
  szm
  jmp as7

as5, lac xy /dsv

as6, xx /sgn
  jda ed
  jmp ami

as7, xor tc
  jda pr
  lac i as8
  sas one
  jmp ami
  jmp as5
/assemble constant

ac,  jsp ar
   law ac1
   spi
   jmp mc
   jsp co
   dac wrd
   law ami

sv,  dap svx
   jsp rro
   add (dsv-1
   dap sv1
   lio wrd

sv1, dio xy
   sub (dsv-1

svx, jmp xy

ac1, jsp rro
   jda cc
   jda wro
   jmp ami

cc,  0
   dap ccx
   lac cc
   add (dss-1
   dap cc2
   spa
   jmp cc1

cc5, cli
   jsp dd

cc2, dac xy

ccx, jmp xy

cc1, lac i cc2
   /dss
   spq
   jmp cc5
   add (400000
   jmp ccx
/assemble assignment

aa, jsp ar
jsp sv
lio mii
spi i
jmp ami
szf i 6 /ds1
jmp aa1
jda cc
jda mp
jmp ami

aa1, add (dss-1
dap aa2
clc

aa2, dac xy /dss
jmp ami

/write dummy symbol specification

wsp, szf i 4 /liu
jmp ev2
lac (-200000
xct sgn
sub (-200000
dac t1
lac sym
jda sds
jsp uds
add t1
jda pr
jmp evx

/prepare dummy symbol specifications

pr, 0
lio pr
prs, dio.
dap prx
idx prs
sad (dio pdl+ncd
jsp tmp
str 6 /ds1
prx, jmp xy
/store dummy symbol specification

ss,      dap ssx
         lac prs
         dap sst
         lac i lp1.
         dap prs
         sub one
         dap ss1
         jmp ss2

ss3,     jsp sco
         jsp scz

ss1,     lac xy      /pdl
         jda wro
ss2,     index ss1,sst,ss3

ssx,     jmp xy

sst,     lac xy

/store word in mai

smb,      lac wrd
         sza
         jmp sm7
         lac tea
         jmp scz

sm7,      jsp sco
         lio wrd
         lac tea

sm,       dap smx
         idx mai
         dio i mai
        lio pss
         spi i
         jmp sm2
         dac hih
         sad low
         jsp sce

sm2,      cla

smx,      jmp .
/encode dummy symbol specification

wro, 0
dap wrx
lio wro
law i 7;
dac t3

wr0, law wr2
spi
jmp sco
jmp scz

wr2, rir 1s
isp t3
jmp wr0

wrx, jmp .

/decode dummy symbol specification

rr0, dap rrx
dzm t2
setup t3,7

rr0, law rr1
jda tc
law 100

rr1, add t2
rar 1s
dac t2
isp t3
jmp rr0
lac t2
lio t2

rrx, jmp xy
/store code bit

sco,    dap scx
        lac (400000
        jmp sc1

scz,    dap scx
        cla

sc1,    dap tc
        isp scn
        jmp sc4
        lac scw

sc3,    dap tc
        lac tc
        ral 1s
        dac scw
        jsp sm
        lac mai
        dap sc3
        lio sc3
        setup scn,22
        jmp scx-1

sc4,    lac tc
        ior scw
        ral 1s
        dac scw
        cla

scx,    jmp xy

/test code bit

tc, 0
dap tcx
isp tcn
jmp tc3
jsp rw
setup tcn,22
jmp tc5

tc3, lio tc
ral 1s

tc5, dio tc
cla
spi

tcx, jmp xy
jmp 1 tc

start
Macro FIO-DEC part 3

/set to pick up constant

1p,
   jsp evl
   law 1
   jda pi
   sad (dio cv4+ncl
   jsp tmc
   lio prs
1p1,
   dio xy
   lio wrd
1p2,
   dio xy
   lio sgn
1p3,
   dio xy
   lio def
1p4,
   dio xy
   sas (dio cv4+1
   jmp rsw
   move tt,ttt
   move ct,tct
   move qt,tqt
   move bt,tbt
   init tt,rp
   dap rt
   dap ct
   init qt,ilf
   dap bt
   jmp rsw

   ttt,  0
   tct,  0
   tqt,  0
   tbt,  0
/save constant and reduce level

rt,       jmp xy

rp,       jsp evl
lac mi1.
spq
jmp rp8
jsp co

rp5,      xct i lp3
add i lp2
dac wrd
law 1
dac def
law i 1
jda p1
sas (dio cv4
jmp rp3
move ttt,tt
move tct,ct
move tqt,qt
move tbt,bt
init rt,ilf
stf 5  /syl

rp3,      jsp rss
lac t
sad (55  /right paren
jmp r
sas (77
jmp r2
jmp tt

rp8,      jsp mc
jsp dd
jda wro
lac (-200000
xct i lp3
sub (-200000
add wro
jda pr
cla
jmp rp5

p1,       0
dap pix
lac p1
add lp1
dap lp1
add (cv2-cv1
dap lp2
add (cv3-cv2
dap lp3
add (cv4-cv3
dap lp4
pix,      jmp xy
/constant table search

co,  
dap cox  
idx nca  
lac psq  
spq  
jmp co8  
lac def  
spq  
jmp usc  
lac con  
dap co3  
jmp co4+1

c02,  
lac wrd  
c03,  
sad xy  
jmp co6  
c04,  
index co3, nco, co2  
add one  
dac nco  
add (lac-sad+1  
dac hih  
sad low  
jsp sce  
lio wrd  
dio i co3

c06,  
lac co3  
sub con  
add i cn6  
and (7777  
c08,  
dac num  
cox,  
jmp xy
/pseudo-instruction constants

cns, lac mii
  spq
  jsp ilf
  lac loc'
cn6, dac xy /cor table (first)
  dac tlo
  lac nca
  add aml /aml is "alarm location"
  dac aml
  lac pss
  spq
  jmp cn5
  init bs,cn4
  lac con
  dap cn3
  jmp cn8
cn3, lac xy /const. list
  dac wrd
  jmp tb4
cn4, idx cn3
  add (sad-lac
cn8, sas nco
  jmp cn3
  lac loc
cn7, dac cr2 /sto cor table (second)
  lac tlo
  add nca
  dac wrd
  init bs,cn1
  jmp ba1
cn1, init bs, rnw
  move con,nco
dzm nca
  idx cn6
  index cn7,(dac cr2+ncn, rnw
tmc, error alm, alh, flex tmc
/pseudo-instruction "dimension"

dim,
  init rt, di2
  init dtb+57, di1
  init ct, rsw
  init bt, ilf
dap qt
  init tt, rst
  jmp rsw

di1,
  move sym, tcn
  szf 5
  jsp ilf
  jmp rsw
di2,
  jsp evl
  spi
  jsp usp
  move tcn, sym
  move wrd, tcn
  clc
dac let
  jsp evl
  spa
  jmp di3
  spi
  jmp mdd
  lac vct
  add vc1
dac i ea
di4,
  lac vct
  add tcn
  dac vct
  jmp rsw
di3,
  spi i
  jmp mdd
dac t3
  jsp vsm
  jmp di4
mdd,
  move sym, lus
  error alu, rsw, flex mdd
/pseudo-instruction variables

var,  lac mii
     spa
     jmp ilf
     lac loc
     spa
     jmp ilf
     lio vai
     spi
     jmp tmv
     load vai, -0
     lio pss
     spi
     jmp vaa
     sas vc1
     jmp vld

vac,  lac vc2
     dac wrd
     jmp b5

vaa,  dac vc1
     add vct
     dac vc2
     lac aml
     add vct
     dac aml
     jmp vac
/read characters from flexo buffer

rch,     dap rcz
         isp fwd
         jmp rc1

rc8,     lio xy
         dio fwd
         idx rc8
         sub rf3
         sza i
         jmp rc3
         sma
         jmp rfb
         /refill buffer

rc4,     dac fwd

rc1,     lio fwd
         cla
         rcl 6s
         dio fwd
         dac t
         dac rcp

rcz,     jmp xy

rc3,     lac nfc
         jmp rc4

rcp,     0
/refill flexo buffer

rfb,     init rc8, flx
dap rf3
law rf4+1

rf5,     dap rf4''
rf1,     setup nfc, 3
rf2,     rpa
dio t
rir 7s
spi
jmp rf2    /7th code=delete
sense 6
jmp rfa
lac t
sza i
jmp rf2

add (1000

dap .+2
law 5252
rar
spa
jmp 1lp

rfa,     cla
lio t
rcr 6s

rf3,     lio xy    /flx list
cr cl 6s
dio 1 rf3
rcr 6s
sad (130000    /stop code
jmp rf6
sad (770000    /car ret
jmp xy    /.+1 or rf6
count nfc, rf2
index rf3, (lio flx+nfw-24, rf1
law rf6
jmp rf5

rf6,     rcl 6s
isp nfc
ril 6s
isp nfc
ril 6s
dio 1 rf3
law 1 2
sub nfc
dac nfc
idx rf3
jmp rc8

1lp,     law 7143
jda tys
law 4777
jda tys
init sov, rf2
lio t
hlt+cl0-opr
jmp rfa
/pseudo-instructions octal, decimal, expunge and noinput

oct,   lac (opr
        jmp dec+1
dec,   lac (add num
     dac n1
de2,   clf 5 :       /syl
        jmp r2
no1,   clc
     dac ini
     jmp de2

xp,    lio pss
     law low
     spi
     dap low
     jmp de2

/ignore to tab or car ret

itt,   jsp rsl

itc,   clf 5
    dzm wrd
    jsp rss
    lac rcp
    jmp .+2

it1,   jsp rch
    sad (36
    jmp itx
    sas (77
    jmp it1

itx,   jmp r2
/feed subroutine

fee, 0
dap fex
cli
ppa
isp fee
jmp .-2
fex, jmp .

/punch routine

pnb, 0
lio pnb
dap pnx
lac loc
ppb
ril 6s
ppb
ril 6s
ppb
pnx, jmp .

/oct7znt subroutine

opt, 0
dap opx
lio (100000
lac opt
clf 1
op1, rcr 9s
rcr 6s
sza
jmp op2
law 20
op3, swap
szf 1
tyo
sad (10000
stf 1
c11
sas (10000
jmp op1
opx, jmp xy
op2, stf 1
jmp op3
/type subroutine

tys, 
  xx
  dap tyx
  law i 3
  dac opt

tyl, 
  lac tys
  and (770000
  sza 1
  jmp tyc
  rcl 6s
  tyo

 tyc, 
  lac tys
  ral 6s
  dac tys
  isp opt
  jmp tyl

tyx, 
  jmp .

/tab typer

 tb, 
  dap .+3
  law char r
  jda tys
  jmp .

/permute zone bits

 per, 
  0
  dap pex
  lac per
  cli
  rcr 6s
  sza
  jmp .-2
  dio per
  lac per
  and (202020
  ral is
  xor per
  xor (400000

 pex, 
  jmp .
/error print routines.

ust, error alu, tb3, flex usw
usb, error alu, b5, flex usl
usq, error alu, rst, flex usp
uss, error alu, s2, flex uss
usm, jda alu
    flex usm
usc, jda alu
    flex usc
usr, error alu, rst, flex usr
usp, jda alu
    flex usa
usd, jda alu
    flex usd
uds, dio lus
    error alu, evx, flex uds
il, error alm, r, flex ich
ilf, error alm, itt, flex ilf
ipi, error alm, itc, flex ipi
mdt, move sym, lus
    error alu, rnw, flex mdt
mdm, error alm, dm1, flex mdm
ipa, error alm, itt, flex ipa
ids, dgm sym
    jda alm
    flex ids
ils, error alm, alh, flex ils
sce, error alm, alh, flex sce
tmp, error alm, alh, flex tmp
vld, error alm, rnw, flex vld
tmv, error alm, rnw, flex tmv
/error print routine

alu, 0
    move alu, alm
    jmp alb

alm, 0
dzm lus

alb, dap +3
    lac alm
    dap sov
    lac xy
    jda tys
    jsp tb
    lac loc

spa
    jmp al1
    jda opt
    jmp al2

al1, lac (flex 1nd
    jda tys

al2, jsp tb
    lac asi
    spa
    jmp al6
    lac asm
    jda per
    jda tys
    lac aml
    sza i
    jmp al6
    lio aml
    lac (flex +
    spi

law char r-
    jda tys
    lac aml
    spa
    cma
    jda opt

al6, lac api
    sza i
    jmp al9
al7, jsp tb
lac api
jda tys
lac syn
jda tys
lac lus
sza i
jmp al8

als, jsp tb
lac lus
jda per
jda tys

al8, law 77
jda tys
lat
rar 1s
lio (-0
sma

alh, clo+hlt-opr
dio pch
jmp sov

al9, lac lus
sza i
jmp al8
jmp tb
jmp als
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>004277</td>
<td>400000</td>
<td>/1</td>
</tr>
<tr>
<td>625151</td>
<td>514600</td>
<td>/2</td>
</tr>
<tr>
<td>224145</td>
<td>453200</td>
<td>/3</td>
</tr>
<tr>
<td>141211</td>
<td>771000</td>
<td>/4</td>
</tr>
<tr>
<td>274545</td>
<td>453100</td>
<td>/5</td>
</tr>
<tr>
<td>364545</td>
<td>453000</td>
<td>/6</td>
</tr>
<tr>
<td>010171</td>
<td>050300</td>
<td>/7</td>
</tr>
<tr>
<td>324545</td>
<td>453200</td>
<td>/8</td>
</tr>
<tr>
<td>065151</td>
<td>513600</td>
<td>/9</td>
</tr>
<tr>
<td>000077</td>
<td>000000</td>
<td>//</td>
</tr>
<tr>
<td>224545</td>
<td>453000</td>
<td>/s</td>
</tr>
<tr>
<td>010177</td>
<td>010100</td>
<td>/t</td>
</tr>
<tr>
<td>374040</td>
<td>403700</td>
<td>/u</td>
</tr>
<tr>
<td>073060</td>
<td>300700</td>
<td>/v</td>
</tr>
<tr>
<td>376014</td>
<td>603700</td>
<td>/w</td>
</tr>
<tr>
<td>412214</td>
<td>224100</td>
<td>/x</td>
</tr>
<tr>
<td>010274</td>
<td>020100</td>
<td>/y</td>
</tr>
<tr>
<td>615141</td>
<td>454300</td>
<td>/z</td>
</tr>
<tr>
<td>141414</td>
<td>141400</td>
<td>//=</td>
</tr>
<tr>
<td>000041</td>
<td>000000</td>
<td>/ }</td>
</tr>
<tr>
<td>204040</td>
<td>403700</td>
<td>/j</td>
</tr>
<tr>
<td>771014</td>
<td>224100</td>
<td>/k</td>
</tr>
<tr>
<td>774040</td>
<td>404000</td>
<td>/l</td>
</tr>
<tr>
<td>770214</td>
<td>027700</td>
<td>/m</td>
</tr>
<tr>
<td>770214</td>
<td>207700</td>
<td>/n</td>
</tr>
<tr>
<td>364141</td>
<td>413600</td>
<td>/o</td>
</tr>
<tr>
<td>771111</td>
<td>110600</td>
<td>/p</td>
</tr>
<tr>
<td>364151</td>
<td>215600</td>
<td>/q</td>
</tr>
<tr>
<td>771111</td>
<td>314600</td>
<td>/r</td>
</tr>
<tr>
<td>000041</td>
<td>000000</td>
<td>/ }</td>
</tr>
<tr>
<td>101074</td>
<td>101000</td>
<td>+</td>
</tr>
<tr>
<td>001422</td>
<td>410000</td>
<td>/(</td>
</tr>
<tr>
<td>761111</td>
<td>117600</td>
<td>/a</td>
</tr>
<tr>
<td>774545</td>
<td>453200</td>
<td>/b</td>
</tr>
<tr>
<td>364141</td>
<td>412200</td>
<td>/c</td>
</tr>
<tr>
<td>774141</td>
<td>413600</td>
<td>/d</td>
</tr>
<tr>
<td>774545</td>
<td>414100</td>
<td>/e</td>
</tr>
<tr>
<td>770505</td>
<td>010100</td>
<td>/f</td>
</tr>
<tr>
<td>364151</td>
<td>513000</td>
<td>/g</td>
</tr>
<tr>
<td>771010</td>
<td>107700</td>
<td>/h</td>
</tr>
<tr>
<td>004177</td>
<td>410000</td>
<td>/i</td>
</tr>
<tr>
<td>010300</td>
<td>010300</td>
<td>/close quotes</td>
</tr>
<tr>
<td>000060</td>
<td>600000</td>
<td>/</td>
</tr>
<tr>
<td>030200</td>
<td>030200</td>
<td>/open quotes</td>
</tr>
</tbody>
</table>
/Indicators and variable storage

va1, 0 /variables pseudo-instruction indicator
vc1, 0 /beginning of variables
vc2, 0 /end of variables
vcn, 0 /variables counter
ovb, 0 /overbar indicator, 1= on, 0= off
pss, 0 /0 = begin pass, +1 = continue pass
nma, 0 /0 = do not punch, +1 = punch if pass 2
pch, 0 /0 = suppress input routine, +1 = punch input routine
tit, 0 /0 = suppress title, +1 = punch title
psa, 0 /end of pseudo-instruction list) at beginning
psb, 0 /end of macro-instruction list ) of pass 1
ini, 0 /aux. input routine indicator
hth, 0 /upper limit of macro instruction and constant list
nfc, 0 /test word for end of flexo word list
lus, 0 /last undefined symbol
fwd, 0 /flexo word from input tape
fwb, 0 /flexo word from list
wrd, 0 /partial sum of syllables of word
num, 0 /number = value of syllable.
sym, 0 /symbol = flexo word for symbol.
def, 0 /0 = indefinite word, +1 = definite
chs, 0 /character count of characters in syllable
let, 0 /0 = no letters in syllable, -0 = at least one letter
api, 0 /last pseudo-instruction for error stop
asi, 0 /relative location+0 = yes, -1 = no
asm, 0 /alarm symbol for relative location.
am1, 0 /location relative to above symbol (asm)
asm, 0 /for establishing above symbolic relative
asa, 0 /location from location
amm, 0 /assignment
cen, 0 /current address in constant list
nco, 0 /number of distinct constant values
nca, 0 /number of distinct syllables
tlo, 0 /temporary for current location
mil, 0 /macro instruction mode indicator
mi, 0 /define indicator
syn, 0 /second three characrs of M-I name
tea, 0 /temporary subroutine exit address
scn, 0 /
scw, 0 /
scn, 0 /
tcn, 0 /
tcn, 0 /
tcc, 0 /
tcc, 0 /
rl, 0 /dummy symbol count
dsl, 0 /temporary for dum sym count
t, 0 t1, 0 /temporary
t2, 0 t3, 0 /registers
constants

/pseudo instruction list and macro names and definitions

psi/    law npi-3
mai/    lac npi-1

    text .repeat. rpt
    text .charac. ch
    text .fle xo. fx
    text .tex t. txt
    text .sta rt. sta
    text .termin. ter
    text .define. dfn
    text .consta. cns
    text .oct al. oct
    text .decima. dec
    text .noinpu. noi
    text .expung. xp
    text .variab. var
    text .dimens. dim

np1,

dss/     1
dsm/     110000
cvl/     pd1

low/     lac low

    start ps5
SYMBOL PACKAGE - macro fio-dec

/MACRO P SYMBO PUNCH 10-27-61

flx/

lsb,        clf 5
senses 1001
jmp 7751

law i 20
jda fee
ls,        listen
swap
senses 1001
jmp 7751
sad (77
jmp ls3
sas (36
jmp pt1-5

ls2,        listen
swap
ls3,        senses 1001
jmp 7751
lio (jmp sps
sad (char rm
lio (jmp mps
sad (char rs
str 5
dio sps-1
lio ls3+2
dio .-2
sas (77
jmp ls2
law i 40
jda fee
iac end-1
jda pnb
law i 40
jda fee
xx
sps,        lac low
dap bpp
law low+1
jda end
szf 5
jmp pse
law i 40
jda fee

mps,        law psi
dap bpp
add (2
jda end
init bpp,npi
lac mai
add (law-lac+1
sad .-4
jmp pse
dap end
jsp pst

pse,  
  law i 30
  jda fee'
  lac (jmp ps5
  jda pnb
  law i 240
  jda fee
  jmp 7751

end,  
  0
pst,  
  dap psx
  clf 4
bpp,  
  law xy
psr,  
  dac org
  dap sor
  and (-77
  add (100
  dac loc
  law pbf
  dap .+2
psu,  
  lac 1 sor
  dac .
  idx .-1
  dap ts
  idx sor
  sad end
  jmp .+4
  sad loc
  jmp psc
  jmp psu
  dac loc
  str 4

pcb,  
  jmp psc
  szf 4
psx,  
  jmp xy
  lac loc
  jmp psr

psc,  
  senses 1001
  jmp 7751
  jmp pun+6

sor,  
  xy

constants

bnp/  
  jmp pcb+1
pt1/  
  jmp pt1+4
pt6-1/  
  jmp ls

start lsb
RESTORE

bnp/ lac wrd
pt1/ lio t
pt6-1/ jmp ptl

/Text printer
pbf/
txp, 0
dap txu
txu, lio .
  ril 6s
tyo
  ril 6s
tyo
  ril 6s
tyo
  idx txu
sub (lio
  sas txp
jmp txu
jmp 1 txp

constants
/init. sym. val

ist, flex 1s 1
      flex 2s 3
      flex 3s 7
      flex 4s 17
      flex 5s 37
      flex 6s 77
      flex 7s 177
      flex 8s 377
      flex 9s 777

char li 10000

flex and 020000
      flex ior 040000
      flex xor 060000
      flex xct 100000
      flex jfd 120000
      flex cal 160000
      flex jda 170000
      flex lac 200000
      flex lio 220000
      flex dac 240000
      flex dap 260000
      flex dip 300000
      flex dio 320000
      flex dzm 340000
      flex add 400000
      flex sub 420000
      flex idx 440000
      flex isp 460000
      flex sad 500000
      flex sas 520000
      flex mus 540000
      flex dis 560000
      flex jmp 600000
      flex jsp 620000

flex skip 640000
      flex szf 640000
      flex szs 640000

      flex sza 640100
      flex spa 640200
      flex sma 640400
      flex szo 641000
      flex spi 642000
| flex ral   | 661000 |
| flex ril   | 662000 |
| flex rcl   | 663000 |
| flex sal   | 665000 |
| flex sil   | 666000 |
| flex scl   | 667000 |
| flex rar   | 671000 |
| flex rir   | 672000 |
| flex rcr   | 673000 |
| flex sar   | 675000 |
| flex sir   | 676000 |
| flex scr   | 677000 |
| flex law   | 700000 |
| flex iot   | 720000 |
| flex tyi   | 720004 |
| flex rrb   | 720030 |
| flex oks   | 720033 |
| flex lsm   | 720054 |
| flex esm   | 720055 |
| flex cdf   | 720074 |
| flex cfd   | 720074 |
| flex rpa   | 730001 |
| flex rpb   | 730002 |
| flex tyo   | 730003 |
| flex ppa   | 730005 |
| flex ppb   | 730006 |
| flex dpy   | 730007 |
| flex clf   | 760000 |
| flex nop   | 760000 |
| flex opr   | 760000 |
| flex stf   | 760010 |
| flex cla   | 760200 |
| flex hlt   | 760400 |
| flex xx    | 760400 |
| flex cma   | 761000 |
| flex clc   | 761200 |
| flex lat   | 762200 |
| flex cli   | 764000 |

iyi, -0 -0
/CONSTANTS PRINTER

yc,  szs 1 30
     szs 20
     jmp ych
     jmp 7751

ych,  lac cn7
     sad (dac cr2
     jmp 7751
     dap yct
     law yc2
     jda txp
     jmp 7751
     /red, c.r., u.c.
     /c, l.c., o
     text .nstants area.

yc2,  lac pss
     spa
     jmp yc3
     law yc4
     jda txp
     text /, inclusive
     char lo+3477

yc4,  stf 5

yc7,  law cor
     dap ycm
     law cr2

ycr,  dap ycn

ycu,  sad yct
     jmp 7751

ycm,  lac .    /cor
     spa
     jmp ycp
     jda opt
     szf i 5
     jmp ycq
     law 36
     jda tys
     law i 1

ycn,  add .    /cr2
     jda opt

ycq,  law 77
     jda tys

ycr,  idx ycm
     idx ycn
     jmp ycu

yc3,  law yc6
     jda txp
     text / origi/ flex ns +34

yc6,  clf 5
     jmp yc7
yct,    add .
ycp,    law yco
       jda txp
       357145 /red, 1, n
       flex def
       char l:+3477
yco,    jmp yck

constants

start yc
ALPHA SYMBOL PRINTER

yc/
ycs,
  szs i 20
  jmp syx
  law ycl
  jda txp :
  3577
  text /Defined Symbols ALPHA/
  3477
ycl,
  lac low
  sad .-1
  jmp syx
  dap yc8
  lio (77
  iot 4003
ycy,
  law ist
  dap yca
yca,
  lac . /ist
  jda per
yc8,
  sad . /symbol
  jmp ycb
  idx yca
  idx yca
  sas (lac iy1
  jmp yca
  clrf 5
ycz,
  iot i
  szs i 20
  jmp syx
  lac i yc8 /symbol
  jda per
  jda tys
  jsp tb
  idx yc8
  lac i yc8 /value
  jda opt
  szf i 5 /set if print
  jmp ycl
  jsp tb
  lac i yca
  jda opt
yc1,
  lio (77
  iot 4003
  jmp ycv
ycb,  idx yc8
idx yca
lac i yc8
sad i yca
jmp ycc
str 5
law i 1
add yc8
dac yc8
jmp ycz

ycc,  idx yc8
ycc,  idx yc8
ycc,  idx yc8
ycc,  idx yc8

ycc,  ycv,  idx yc8
ycc,  ycv,  idx yc8
sas (sad low
jmp ycy
iot i

syx,  szs i 30
jmp 7751
law syy
jda txp
jda txp
357777

text /Defined Symbols NUMERIC/
3477
syy,  jmp 7751

constants

start ycs
NUMERIC SYMBOL PRINT

yc/
sy,
  szs 30 i
  jmp 7751
dzm t:
  init sy3,1st
  init sy4,1st+1
  lio (77
  tyo-4000

sya,
  lac t
dac t1
clc
dac t
lac low
dap syb
idx syb

syb,
  lac xy
  /value
  lio i syb
  xor t1
  spa
  jmp sq5
  sza i
  jmp syc
  xor t1
  sub t1

sq1,
  spa
  jmp sy1

sq2,
  lac t
  xor i syb
  spa
  jmp sq3
  lac i syb
  sub t

sq4,
  spa
dio t

sy1,
  idx syb
  idx syb
  sas (lac low+1
  jmp syb
  lac t1
  cma
  sza
  jmp sya
  iot i
  jmp 7751

sq5,
  lac t1
  jmp sq1
sq3,  lac t
   jmp sq4

syc,  law i 1
      add syb.
      dap syz'

sy4,  lac xy
      xor i syb
      spa
      jmp sy5
      sza i
      jmp syf
      lac i syb
      sub i sy4

sy5,  spa
      jmp syp

syd,  idx sy4
      dap sy3
      idx sy4
      jmp syg

syf,  lac i sy4
      jmp sy1

syp,  iot i
      szs i 30
      jmp 7751

syz,  lac xy
      jda per
      jda tys
      jsp tb
      lac i syb
      jda opt
      lio (77
      tyo-4000
      jmp sy1

sy3,  lac xy
      jda per
      sas i syz
      jmp syp
      idx sy4
      dap sy3
      idx sy4
      jmp sy1

      constants

start sy

/ist value

/mai symbol

/ist table
/restore macro

dsm/
rm,
Jmp 7751
load mai,lac npi-1
load psi,law npi-3
load low,lac low
init rm2,ist-2

rm4,
idx rm2
idx rm2
add (1
dap rm3

rm2,
lac xy
sad lye
Jmp 7751
jda per
dac sym

rm3,
lac xy
dac t3
jsp vsm
Jmp rm4

constants

start rm
/final "where to go routine"

```plaintext
dsm/ 110000  /permuted char lr
    szs 40
    jmp ps5
    lac pss
    sma+szf 6-skp
    jmp s6
    sma
    jmp s4
    szf 6
    jmp 1st
    jmp s5

 collaborate

dss/ 1

cv1/ pdl

start dsm+1
```
APPENDIX 2

MACRO INSTRUCTION EXAMPLE
Appendix 2: Macro Instruction-Example

The sample program on the next page is analyzed in detail to illustrate most of the features of the macro processor. We illustrate first how a programmer might analyze the macros. Each successive level of macro expansion is indented one column from its predecessor.

On the next page is listed an English transliteration of the macro structure from MACRO's point of view. Internal dummy symbol numbers correspond to the letters used as shown by the chart below. The most important changes to the dss table are shown also, but the reader should remember that any dummy symbol parameter assignment will in general alter the dss table. Note particularly how the extra argument of second is lost.

Finally there is an octal and binary dump of the mai table for these macros. The octal numbers are in the left hand column, and on the right appear the binary forms of the same numbers divided off according to their significance. Numbers in parentheses are value words associated with the zero-nonzero indicator bits immediately preceding them. Periods represent word boundaries, and semicolons represent statement boundaries. Each statement corresponds precisely with one entry in the mai table as listed on the preceding page. The pseudo-instruction data is shown also.

Table of Dummy Symbols

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
</table>
Sample program: June, 1962, RAS.

define first A, B, C
law A
add B
dac C
term

define second X, Y
Z=105
dac Z
X=X+(Y
first 1, (X, X+X
lac Z
Z=X
add Z
term

define third J, K
second 100, J+(K+200, K
term

a, first a, b, c
second 1, 2
third 10000, (40000
dac d
hlт
b, 0
c, 0
d, 0

const

start a
Expansion of Sample Program

<table>
<thead>
<tr>
<th>Source tape</th>
<th>Intermediate results</th>
<th>Word</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>a,</td>
<td>first 4, 25, 26</td>
<td>law 4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add 25</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dac 26</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>second 1, 2</td>
<td>dac 105</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>z=105</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dac z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=1+(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=1+30</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first 1, (31), 62</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first 1, 31, 62</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lac z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z=x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z=31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>third 10000, (40000)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>third 10000, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>second 100, 10000+(32+200), 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>second 100, 10000+33, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>second 100, 10033, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z=105</td>
<td>dac 105</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>dac z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=100+(10033)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=100+34</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=134</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first 1, (134), 270</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first 1, 35, 270</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lac z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z=x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z=134</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dac d</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hlt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dac 27</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hlt 24</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>const</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>40000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>232</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10033</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>134</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Sample Program Macros as Seen by MACRO

<table>
<thead>
<tr>
<th>English input</th>
<th>Read from mai</th>
<th>Stored into mai</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>define</code></td>
<td><code>term</code></td>
<td></td>
</tr>
<tr>
<td><code>first A, B, C</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>law A</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add B</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>dac C</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>A=A+(B)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>term</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>second A, B</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>C=105</code></td>
<td></td>
<td><code>C=105</code></td>
</tr>
<tr>
<td><code>dac C</code></td>
<td></td>
<td><code>C+2400000</code></td>
</tr>
<tr>
<td><code>A=A+(B)</code></td>
<td></td>
<td><code>D=(B+0)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>A=A+D+0</code></td>
</tr>
<tr>
<td><code>first 1, (A), A+A</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>lac C</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>C=A</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add C</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>term</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>second 100, A+(B+200), B</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>third A, B</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>set dss[2] to 0</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>term</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `A+7000000`
- `B+4000000`
- `C+2400000`
- `A+700000`
- `B+400000`
- `C+240000`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `C+200000`
- `A+700000`
- `B+400000`
- `C+240000`
- `700001`
- `F+400000`
- `G+240000`
- `C+200000`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
- `E=(A+O)`
- `F=E+O`
- `G=A+O`
- `A=A+D+O`
- `C=A+0`
- `C+400000`
Octal and Binary Dump of main Table

<table>
<thead>
<tr>
<th>FIRST</th>
<th>SECOND</th>
</tr>
</thead>
<tbody>
<tr>
<td>667151</td>
<td>226853</td>
</tr>
<tr>
<td>002223</td>
<td>464564</td>
</tr>
<tr>
<td>705026</td>
<td>705031</td>
</tr>
<tr>
<td>[pointer]</td>
<td>[pointer]</td>
</tr>
<tr>
<td>420314</td>
<td>721041</td>
</tr>
<tr>
<td>10 0010000 0 1(700000), 10 0110000 0 1(400000), 111.1</td>
<td>1110 1(105) 0001000, 10 0001.000</td>
</tr>
<tr>
<td>700000</td>
<td>105</td>
</tr>
<tr>
<td>060417</td>
<td>031414</td>
</tr>
<tr>
<td>0 1(400000), 10 0001000 0 1(240000), 111.1</td>
<td>0 1(240000), 10 0110000 110 0.(0)</td>
</tr>
<tr>
<td>400000</td>
<td>240000</td>
</tr>
<tr>
<td>240000</td>
<td>242102</td>
</tr>
<tr>
<td>243450</td>
<td>210303</td>
</tr>
<tr>
<td>10 0110000 110 0(0) 0011.000,</td>
<td>10 0010000 110 0(0) 1001.000</td>
</tr>
<tr>
<td>043070</td>
<td>704204</td>
</tr>
<tr>
<td>10 0011000 1110 0(0) 0.111000,</td>
<td>10 0010000 10 0.010000</td>
</tr>
<tr>
<td>207004</td>
<td>207004</td>
</tr>
<tr>
<td>1110 0(0) 0000100,</td>
<td>0 1(700001), 10 0110000 0 1(400000), 10 000.0100</td>
</tr>
<tr>
<td>316060</td>
<td>700000</td>
</tr>
<tr>
<td>0 1(700001), 10 0110000 0 1(400000), 10 000.0100</td>
<td>0 1(240000), 10 0001000 0 1(200000),</td>
</tr>
<tr>
<td>700000</td>
<td>400000</td>
</tr>
<tr>
<td>240000</td>
<td>240000</td>
</tr>
<tr>
<td>200000</td>
<td>041622</td>
</tr>
<tr>
<td>1.0 0010000 1110 0(0) 10010.00,</td>
<td>10 0001000 0 1(400000), 1111/</td>
</tr>
<tr>
<td>102076</td>
<td>400000</td>
</tr>
<tr>
<td>10 0001000 0 1(400000), 1111/</td>
<td>400000</td>
</tr>
</tbody>
</table>
extended pdp-1 ops and macros, Jan 1962

lap=cla 100
ioh=iot 1
clo=651600
spq=650500
szm=640500

define
  sensewitch A
  repeat 3, A=A+A
  szs A
  term

define
  initialize A, B
  law B
  dap A
  term

define
  index A, B, C
  idx A
  sas B
  jmp C
  term

define
  listen
  cla+cli+clf 1-opr-opr
  szf 1 1
  jmp .-1
  tyi
  term

define
  swap
  rcl 9s
  rcl 9s
  term

define
  load A, B
  lio (B
  dio A
  term

define
  setup A, B
  law 1 B
  dac A
  term

define
  count A, B
  isp A
  jmp B
  term
define  
               move A, B
               lio A
               dio B
               term

define  
               clear A, B
               init .+2, A
               dzm
               index .-1, (dzm B+1, .-1
               term

start
