PDP-1 COMPUTER
ELECTRICAL ENGINEERING DEPARTMENT
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

PDP-45

CERTAINLY (ABRIDGED)

31 January 1972

Introduction

Certainly assembles source programs written in PDP-1 assembly language into object programs. The source language provides a convenient way of coding algorithms while giving the programmer complete control over the content of the object program. The source program may be read from the drum or the on-line typewriter. The object program is written onto drum field 1.

This is an abridged edition of the Certainly manual. Many advanced and powerful features of the assembler are not described in this manual.

Note: Sections identified with an asterisk (*) may be omitted on a first reading.

Certainly processes the source program twice. During pass 1 address tags and other symbols are defined, and constants and variables areas are allocated. During pass 2 the object program is produced. Macros, repeats, and conditionals are expanded during both passes.

A sample program written in Certainly assembler language is shown below.

```
sum
n=100
100/
a,          law tab
            dap b
            dzm s
b,          lac
            adm s
            idx b
            sas c
            jmp b
            dsm
tab,        tab+n/
s,          0
c,          lac tab+n
start a
```

The first non-blank line is the title, which is printed on the typewriter. The program ends with the start pseudo-instruction, or, if there is no start pseudo-instruction, with the end of the source program.

The Source Language

For clarity, the following symbols are assigned to the invisible characters when needed in examples of parts of source programs.

carriage return (cr)
tabulation (tab)

The source program is considered to be a series of syllables and separators. A separator is one of the following characters - space, tab, cr, +, -, ×, ∧, ∨, <, >, ~, =, comma, (,), [,], and slash. A syllable is a string of alphanumeric characters (digits, letters, and period) preceded and followed by separators.

The most important object in the source language is the expression, which has a numerical value to be used as a storage word of the object program, location assignment, argument, etc. An expression is one or more terms separated by suitable combining operators. The following are some of the forms terms can take -

A symbol is a syllable containing at least one letter. Symbols may be of arbitrary length, but are recognized by their first six characters. If a symbol is undefined, the expression in which it appears is undefined. If it is defined as a macro-instruction or pseudo-instruction, special action is taken. The mnemonics for the PDP-1 machine instructions are initially defined as shown in Appendix I.

A number is a syllable which is a string of digits with an optional decimal point at the end. The value of a number is computed modulo 777777, except that 777777 is not changed to 0. If a number is immediately followed by a decimal point, then it is taken as decimal regardless of the current radix.

The syllable consisting of a single point evaluates to the current location, which is the address at which the current instruction is to be assembled.

A term consisting of upper case characters is a micro-program instruction (see memo PDP-35). The syllable must not contain case shifts.

A double quote (") followed by an upper or lower case character is a term, which has the value of the 7-bit concise code of the character. The 7-bit concise code of a character is its concise code, plus 100 if the character is in upper case.

Certain pseudo-instructions generate terms. See the descriptions of the pseudo-instructions for details.

flexo abc is a term with value 616263

Terms may be combined by use of the following operators. Arithmetic is performed in one's complement.

+ or space means addition. A sum of zero is always plus zero.

- means subtraction. Minus signs count out properly, thus ---3 = -3. -0 is not changed to +0.

V means bitwise inclusive or

Λ means bitwise and

~ means bitwise exclusive or

× means integer multiply. Multiplication is mod 777777.

> means integer quotient. The argument on the left is divided by the argument on the right. Division by zero returns the original dividend.

< means remainder of integer division. Division by zero returns zero.

Operator priority

Operations of the same priority are performed from left to right. Operations of different priorities are performed in the order given in the table below.

```
unary + and -    (executed first)
<
>
Λ
×
~
V
binary + and -   (executed last)
```

Two consecutive operators are assumed to have zero between them. The following are some examples of expressions, giving the values (in octal) on the right.

expression	value
2	2
262143.	777777
262144.	1
-0	777777
2+3	5
2-3	777776
2x3	6
400000x7	400003
2V3	3
2∧3	2
2~3	1
-5V1	777773
5<3	2
13>5	2
7-2V3	4
add 40	400040
claVcma	761200
+-4	777773
--1	1
3xx2	0
TAZ+30	774032
^{n}x	173

Operations on expressions

An expression enclosed in brackets is a term with the value of the expression. Brackets may thus be used for grouping in order to force evaluation of parts of an expression in a certain order.

2x[3+4] has value 16

Warning - brackets are removed in a repeat range or macro argument list. An extra pair of brackets is sometimes needed to circumvent this.

An expression enclosed in parentheses is a term which evaluates to the address of the register in the next constants area where the expression is stored. See the description of constants for details.

lio (20), or, as usually written, lio (20, assembles an instruction which places 20 in the in-out register by loading it from a register in the constants area in which 20 is stored.

(x) An expression preceded by one of the conditional pseudo-instructions ifp, ifm, ifz, ifn, or ifup, and followed by a slash, is a term with value 1 or 0 depending on the result of a test applied to the expression. See the description of these pseudo-instructions for details.

ifup 2/ has value 1 if sense switch 2 is up, zero otherwise.

ifz a/∧ifp b+2/+3 has value 4 if a is zero and b+2 is positive, and 3 otherwise.

Note that the bitwise and, or and exclusive or operators may be used as logical operators with logical values 0 and 1.

Grouping brackets, constants, and conditionals may be nested to any reasonable depth.

The closing parenthesis, closing bracket, or slash that ends the expression within a constant, conditional, or grouping brackets may be omitted. The assembler will assume that the missing character was placed in the last position that will result in a syntactically correct expression.

```
      examples - lio (20
        The assembler assumes a right paren
        just before the cr.

                 repeat ifz a,foo
        The assembler assumes a slash before the comma.
```

Uses of Expressions

The meaning of an expression to Certainly is determined by the context in which it appears in the source program. The character immediately following the expression usually indicates its use.

Storage word

An expression terminated by a cr or tab is a storage word and is assembled into the object program.

```
     examples - jmp ret
                lac abc
```

The 18 bit number which is the value of the expression is assigned a location in memory determined by the location counter in the assembler. After each word is stored, the location counter is advanced by one. A storage word may be an instruction, a constant, or data. A tab or cr not preceded by an expression, or preceded by arithmetic operators only, with no syllables, does nothing. If a storage word is undefined on pass 2, the usw error message will be given.

Location assignment

An expression terminated by a slash is a location assignment. The current location is set to the value of the expression truncated to twelve bits.

```
     example - 100/     sza
                        jmp 100
```

The above source program part will cause the instructions sza and jmp 100 to be assembled into locations 100 and 101 of the object program, respectively.

An undefined location assignment will give the usl error message.

Address tag

An expression followed by a comma is an address tag. If the tag is a single undefined symbol, that symbol will be defined to be equal to the current location. If it is a defined expression, it is compared with the current location, and a disagreement will cause an mdt error message to be printed. (Use of the same symbol as an address tag twice in one program is a common cause of this error.) If the tag is undefined but more complicated than a single symbol, it is ignored on pass 1 and a ust error is given on pass 2.

```
        example - a,    dzm i tab+n
                        SXXZ
                        jmp a
```

When the assembler defines a symbol as an address tag, if the comma is preceded by a centerdot, the symbol is defined in such a way that it will not be transmitted to ID.

Note the opposite character of location assignments and address tags. A location assignment moves the value of an expression into the location counter, while a tag moves the location counter into the symbol which forms the tag.

```
A sequence such as  tab,
                    tab+n/
```

is frequently used to reserve a block of registers for a table of data. In the above example, the length of the block is n, and tab is defined as the address of the first register in the block.

(x) Formal symbol definition

A symbol followed by an equals sign and an expression is defined to have the value of the expression if the expression is defined. If the expression is not defined, no action is taken on pass 1, and the use error is given on pass 2. A formal symbol definition overrides any previous definition of the symbol, whether it was a numeric definition, an instruction mnemonic, a pseudo-instruction, or macro. If an underbar precedes the equals sign, the symbol will be defined in such a way that it will not be transmitted to ID.

```
examples - n=100
           t=t+t
           smi_=spi i
```

No storage word is generated by a formal symbol definition.

Comments

A slash, when not preceded by an expression, begins a comment. All characters are ignored up to the next carriage return.

The location counter

The location counter records the address at which the current storage word is to be assembled. It is set to zero at the beginning of each pass and is advanced by one after each storage word is assembled. Any attempt to assemble a word, constant, or variable into location 10000 will produce an rpm error.

```
example -        dzm i tab+n
                 SXXZ
                 jmp .-2
```

assembles into the same sequence of instructions as the example given in the section on address tags.

Pseudo-instructions

Pseudo-instructions are special commands to the assembler. They are usually used for generating certain types of data, controlling the assembly process, and defining macros. Each pseudo-instruction has one or more names in the initial symbol table. Certainly acts on a pseudo-instruction whenever it encounters its name followed by any separator other than equals sign. Some of the descriptions below give names that are more than six characters long. Since symbols are recognized by their first six characters only, any pseudo-instruction name may be shortened to six characters (for example, charac instead of character). They may not be shortened further except for character and flexo, for which the alternate names char and flex are defined in the initial symbol table.

Data Generating Pseudo-Instructions

character and char

The pseudo-instruction character (or its abbreviated form char) is used to generate a syllable containing the concise code for a given character. The name of the pseudo-instruction is followed by a separator, the letter l, m, or r, and then the character to be translated. The letter l, m, or r determines whether the following character is to be placed in the left, middle, or right six bits of the word, respectively. The other twelve bits are set to zero. If the character following the separator is not l, m, or r, that character itself is used, and is placed in the right six bits. The term generated by character may be used anywhere within an expression.

```
examples - char ra    =    000061
           char mb    =    006200
           char lc    =    630000
           char d     =    000064
```

flexo and flex

The pseudo-instruction flexo (or its abbreviated form flex) is used to pack three characters into one word. The three characters immediately following the separator after the pseudo-instruction name are packed from left to right. The resulting term may be used anywhere within an expression.

```
example - flexo abc    =    616263

    this is equivalent to char laVchar mbVchar rc
```

text

The pseudo-instruction text is used to assemble an arbitrarily long string of characters. The character immediately following the separator after the pseudo-instruction name is used as the break character. Following characters, up to but not including the next appearance of the break character, are packed three to a word and assembled into the object program. If the break character which ends the string is followed by octal digits instead of a separator, the assembler goes into "octal" mode, in which pairs of digits are taken as 6 bit numbers and packed as if they were characters. When the break character is next encountered the assembler reverts to normal "text" mode. The assembler alternates between text and octal modes until the break character, followed by a separator, is found while in text mode. Note that the string begins and ends in text mode, and there are always an even number of appearances of the break character.

```
examples - text .abc.7652.de.        assembles into
                                      616263
                                      765264
                                      650000

           text //14/abc/13//        assembles into
                                      146162
                                      631300
```

Because text may generate more than one word of data, it should only be used to generate storage words. It should not be used in constants, arguments, etc.

(x) text7

The pseudo-instruction text7 assembles characters in 7-bit form. The pseudo-instruction name is followed by a string in the same format as for text. The 7-bit concise codes of the characters are packed five per two words, left justified. Bit 0 of the first word in each pair is zero. In "octal" mode, three digits are used for each character.

```
example - text7 /What??/
          assembles into
          octal    binary
          254703   0 1010110 0111000 011
          044721   0001 0010011 1010001
          242000   0 1010001 0000000000
```

(x) ifp, ifm, ifz, and ifn

These four pseudo-instructions apply a test to a numeric argument and generate one or zero depending on the result. Ifp, ifm, ifz, and ifn generate 1 if and only if the argument is positive, negative, zero, and nonzero, respectively. For the purposes of the test, +0 is positive and zero, and 777777 (-0) is negative and nonzero. The expression to be tested follows the separator after the pseudo-instruction name and is ended by the next unpaired slash (see the section on syntax). If the expression under test is undefined during pass 1, the term generated by the pseudo-instruction is undefined. If the expression under test is undefined during pass 2, the usi error is given instead.

(x) ifup

The pseudo-instruction ifup is used to test a sense switch at assembly time. The expression following the separator after ifup and ended by the next unpaired slash is taken to be the number of the switch. A value of 1 is generated if that switch is up, zero if down.

Radix Control

All numbers not followed by a decimal point are interpreted according to the current radix. At the beginning of each pass, the radix is set to octal.

decimal

Decimal sets the radix to decimal.

octal

Octal sets the radix to octal. These pseudo-instructions may be used anywhere within an expression, hence an expression may be interpreted partly in decimal and partly in octal.

(⋊) radix

Radix is followed by an expression and sets the radix to the value of that expression. The expression must be defined on both passes. The usx error is given if this is not the case.

Automatic Constant Allocation

It is frequently necessary to assemble an instruction whose address part is the address of a register in which a constant is stored. The assembler facilitates this operation by automatically assembling a register containing a constant whenever the constant appears enclosed in parentheses in an expression. The constant with its parentheses then evaluates to the address in which the constant is assembled. The right parenthesis after the constant may be (and almost always is) omitted. A constant does not need to be defined on pass 1. If it is undefined on pass 2 the usc error will be given.

example - sas (13
  assembles into an instruction which skips if
  the accumulator contains 13

constants

The actual constants are saved in a table in the assembler and then assembled in a block at the next appearance of the constants pseudo-instruction. Duplicated constants are combined and stored in the same register. The amount of space allocated for the constants area during pass 1 may exceed the amount actually used on pass 2, since, if constants are undefined on pass 1 the assembler is sometimes unable to determine whether they are duplicated and must assume that they are not.

The pseudo-instruction constants may be used up to 8 times in a program. Each constant is placed in the next constants area regardless of whether the same constant appeared in an earlier constants area. The programmer should not make any assumptions about the order of constants within a constants area.

Automatic Variable and Array Allocation

Certainly will automatically allocate one register of memory for a variable or temporary if the name of the variable appears with an overbar. The overbar may be anywhere within the name. Only one appearance of the name needs an overbar. The symbol will be defined to have a value of the address of the register which is allocated. A variable must have been previously undefined on pass 1. The mdv error will occur if this is not the case.

dimension

The dimension pseudo-instruction declares a symbol as an array or table to be automatically allocated. Dimension is followed by a series of array declarations separated by commas and terminated by a carriage return. Each declaration consists of the array name optionally followed by its length enclosed in parentheses. If the length specification is absent, the length is assumed to be 1. The length may be any expression, which must be defined on pass 1. The usd error will occur if the array size is not defined. Each array name will be defined as the value of the address of the first word of the array. An array name must have been previously undefined on pass 1. The mdd error will occur if this is not the case.

example - dimension a(10),b(20),c(1),d

declares a, b, c, and d as arrays of 10, 20, 1, and 1 words respectively. The declarations for c and d could have been accomplished by their appearance with an overbar in any expression.

variables

All variables and arrays are placed in a variables area, which the assembler constructs when it encounters the variables pseudo-instruction. This pseudo-instruction may be used up to 8 times in a program. Each variable or array is placed in the next variables area after the overbar or dimension pseudo-instruction that declares it. The programmer should not make any assumptions about the order of variables and arrays within an area. The initial contents of variables and arrays are not assigned by the assembler.

The use of dimension, constants, and variables is shown in the program below.

```
sum
n=100
dimension tab(n)
100/
a,          law tab
            dap b
            dzm s
b,          lac
            adm s
            idx b
            sas (lac tab+n
            jmp b
            dsm
variables
constants
start a
```

This will produce the same object program as the example given in the introduction, except that s is not initialized, and the relative order of s and tab in the variables area is unknown. The array tab is not initialized in either example.

(x) repeat

The pseudo-instruction repeat is used to make the assembler process part of the source program a specified number of times. The pseudo-instruction is followed by the count, which may be any expression and is terminated by a comma. The characters following the comma up to and including the next carriage return are the range. The assembler behaves exactly as if the range had been typed a number of times equal to the count.

```
example - repeat 3,ril 6s      ivk 300

      is treated as if it were
         ril 6s      ivk 300
         ril 6s      ivk 300
         ril 6s      ivk 300

another example

         z=0
         repeat 3,z=z+10     y=0     repeat 3,y=y+1     y+z

      is treated as if it were

         z=0
         z=z+10      y=0     repeat 3,y=y+1     y+z
         z=z+10      y=0     repeat 3,y=y+1     y+z
         z=z+10      y=0     repeat 3,y=y+1     y+z

      which is treated as if it were

         z=0
         z=z+10      y=0     y=y+1     y+z
         y=y+1       y+z
         y=y+1       y+z
         z=z+10      y=0     y=y+1     y+z
         y=y+1       y+z
         y=y+1       y+z
         z=z+10      y=0     y=y+1     y+z
        y=y+1       y+z
         y=y+1       y+z

      which assembles into the sequence of words

         11,12,13,21,22,23,31,32,33
```

The count must be definite on both passes, or the usr error will occur. A negative count is taken as zero.

The repeat range ends on the first carriage return not contained within brackets. These brackets are not to be confused with the brackets used for arithmetic grouping. They serve only to "hide" carriage returns and prevent them from ending the repeat range. The brackets are removed, that is, the assembler behaves as if the range without the brackets had been typed the specified number of times. If a bracket is immediately preceded by an upper case shift and followed by a lower case shift, both case shifts are removed also. In order to permit brackets to appear within the range, only the outermost pair is removed. Where repeats are nested, one pair is removed at each level. Thus, in order to place the arithmetic expression 3x[4+5] within three levels of repeats, three extra pairs of brackets must be used even when there are no carriage returns to hide.

```
repeat 1,[repeat 1,[repeat 1,[3x[4+5]]]]

        becomes

repeat 1,[repeat 1,[3x[4+5]]]

        which becomes

repeat 1,[3x[4+5]]

        which becomes

3x[4+5]
```

(x) Macro-instructions

A macro-instruction is a user-defined "abbreviation" for a given string of characters. Macro-instructions are created by use of the define and terminate pseudo-instructions. Subsequent appearances of the macro-instruction name cause the macro to be "called". The assembler behaves exactly as if the characters that form the definition had been typed in place of the call. A macro-instruction call may supply arguments that are inserted into the definition at specified points. The characters that are substituted for the call are the "expansion" of the macro. Macro-instructions must be defined before they are called.

example with no arguments

```
(definition)  define abs
                    spa
                    cma
              terminate

(call)        lac x
              abs
              dac y

         is treated as if it were

              lac x
              spa
              cma
              dac y
```

example with two arguments

```
(definition)  define move a,b
                    lio a
                    dio b
              terminate

(call)        move j,k+3

         is treated as if it were

              lio j
              dio k+3
```

another

```
(definition)  define clear a,b
                     law a
                     dap .+1
                     dzm
                     idx .-1
                     sas (dzm a+b
                     jmp .-3
              terminate

(call)        clear tab,100
```

is treated as if it were

```
                     law tab
                     dap .+1
                     dzm
                     idx .-1
                     sas (dzm tab+100
                     jmp .-3
```

define and terminate

The pseudo-instruction define is followed by the name of the macro to be defined and then the list of "dummy symbols", separated by commas and terminated by a carriage return. The following text, up to the appearance of the pseudo-instruction terminate, become the definition. All appearances of dummy symbols within the definition are removed and marked as places where arguments are to be substituted when the macro is called. The actual definition begins with the character after the tab or carriage return that ends the dummy symbol list. It ends on and includes the separator before the terminate pseudo-instruction. In order to permit macro definitions within a macro, appearances of define and terminate are counted. The macro ends on the first terminate not paired with a define. If terminate is followed by a separator other than tab or carriage return, a symbol must follow. It is compared with the name of the macro being defined. A disagreement causes the mnd error. This is sometimes helpful in debugging complicated macros.

In order for the assembler to recognize a dummy symbol in the definition, the symbol must be preceded and followed by separators or non-alphanumeric characters such as overbar, underbar, centerdot, or illegal characters. In some cases it is desirable to substitute an argument adjacent to an alphanumeric character, such as a symbol. This would require adjoining a dummy symbol with another symbol, which makes it impossible for the assembler to determine where one symbol ends and the other begins. To prevent this difficulty, the separator single quote is provided. A single quote separates the symbols, permitting recognition of the dummy symbol. The single quote is then removed and does not appear in the expansion. If it is immediately surrounded by case shifts, they are removed also.

```
     example - define type x
                       law char r'x
                       ivk 100
               terminate

               type q     then becomes

                       law char rq
                       ivk 100
```

The use of rx without the single quote would have prevented recognition of x. Where the count of defines is nonzero, i.e. in a definition within a macro, single quotes are not removed, since they will presumably be needed again.

macro calls

A macro is called whenever its name appears followed by a separator other than equals sign. If the separator is tab or carriage return, there are no arguments. Otherwise the following characters, up to the next tab or carriage return, form the argument list. The arguments are separated from each other by commas. They do not include the commas, the separator after the macro name, or the tab or carriage return after the last argument. In order to permit comma, tab, and carriage return in an argument, these characters may be hidden inside brackets in the same way that carriage returns are hidden in a repeat range. The outermost pair of brackets is removed from each argument. The arguments are then substituted as character strings for the dummy symbols in the definition, and the resulting expansion is substituted for the macro call. After the expansion has been processed, assembly resumes with the character after the tab or carriage return that ended the argument list.

If more arguments are supplied than the number of dummy symbols in the definition, the extra arguments are ignored. If too few arguments are supplied, the empty character string is used for the missing arguments, unless a symbol is generated.

generated symbols

It is sometimes helpful to have a macro generate one or more symbols to be used as address tags, etc. within the macro. For this purpose dummy symbols may be declared to be candidates for generated symbols. If a slash appears in the dummy symbol list, all the following symbols are candidates for symbol generation. If, at the time the macro is called, the argument corresponding to such a symbol is missing, the assembler will generate a symbol and use it for the argument. A new symbol is enerated for each call. Generated symbols are of the form .g0001, .g0002, etc. If the argument is supplied, it overrides the generated symbol.

```
example - define ifzero x/y
                  sza
                  jmp y
                  x
              y,
              terminate
```

The generated symbol provides an address for the instruction to jump over x without knowing how many words x will become.

```
        ifzero [lac a
                dac b
                lio c]

            becomes

             sza
             jmp .g0001
             lac a
             dac b
             lio c
        .g0001,
```

stop

The pseudo-instruction stop causes an immediate exit from the most recently entered macro. The assembler behaves as if it had reached the last character of the definition, and continues from the character after the call.

Miscellaneous Pseudo-instructions

start

The start pseudo-instruction indicates the end of the source program. It is optionally followed by an expression to be used as the starting address for the program. If Certainly was started at location 102 (as by "N" from Expensive Typewriter), the starting address is placed in the program counter when control is returned to ID.

Program Format

While Certainly has few requirements on format, many programmers have found that adherence to a fairly rigid format is helpful in writing and correcting programs. The following suggestions have been found useful in this respect.

Place address tags at the left margin, and run instructions vertically down the page indented one tab stop from the left margin.

Use only a single carriage return between instructions, except where there is a logical break in the flow of the program. Then put in an extra carriage return.

Forget that you ever learned to count higher than five. Let Certainly count for you. Do not write "dac .+16", use an address tag. This will save grief when corrections are required.

Have a listing handy when assembling or debugging a program. Carefully note corrections thereon as soon as they are found so as to maintain an up-to-date listing.

As macro-instructions must be defined before they are used, put these definitions at the beginning of the program.

If the pseudo-instructions variables and constants are used, place them at the end of the program, just before start.

Assembly Procedure

Certainly normally reads the source program from Expensive Typewriter's text buffer and places the object program on drum field 1. However, many variations in procedure are possible by typing control characters on the typewriter.

When Certainly is started at location 102 (as it is when the "N" command is given in Expensive Typewriter), it automatically goes through both passes of the assembly and returns to ID as if the sequence z, s, s, and b had been typed. It directs ID to place the starting address of the program in the program counter, read the symbol table, and unsave drum field 1 (which contains the object program) into core.

When Certainly is started at location 104 (as it is when the "M" command is given in Expensive Typewriter), it listens for control characters from the typewriter. After each pass on a program section, it listens for more control characters.

Whenever sense switch 1 is up, Certainly types out every character of the source program, including expansions of repeats and macros. This is useful when debugging macros.

Control Characters

input medium

e	Expensive Typewriter text buffer
y	online typewriter

output medium

d	drum field 1
w	without output (just check for errors)

assembly control

s	begin next pass
f	forget (initialize everything)
z	assign and zero drum field 1
k	print constants and variables areas

exit

b	back to ID, leaving symbol table in core where "2T" command can read it
m	meliorate source program (back to Expensive Typewriter)

Error Messages

Upon detecting an error, Certainly will print a line in the following format.

aaa p,l c d e

where aaa is a three letter code indicating the error, p,l is the page and line numbers at which the error occurred (if input is from Expensive Typewriter text buffer), c is the symbolic address (relative to the last tag), and d is the name of the last pseudo-instruction or macro. In the case of an error caused by a symbol, e is the symbol. Following is a list of error messages and the action taken if assembly is continued.

sce	Symbol table capacity exceeded. No recovery.
pce	Pushdown capacity exceeded (nesting of repeats and macros is too deep). The pushdown list is cleared and assembly starts over at the top level.
cce	Constants capacity exceeded (more than about 400 constants). The current constant will evaluate to zero.
mce	Macro capacity exceeded and the garbage collector could recover no space. No recovery.
ich	Illegal character. It is ignored.
rpm	Wrap around memory. The location counter has exceeded 7777. It will be reset to zero.
ilf	Illegal format. Characters are ignored to the next tab or carriage return.
ipi	Illegal pseudo-instruction. A pseudo-instruction is used in an illegal context. Same recovery as ilf.
mdv	Multiple definition of a variable (a symbol with an overbar was previously defined). The old definition remains.
mdd	Multiple definition in dimension (a symbol in a dimension declaration was previously defined). The old definition remains.
mdt	Multiple definition of a tag. A defined tag does not match the location counter. The tag is not redefined.

usw	Undefined symbol in a storage word. The symbol is taken as zero. All error messages beginning with "us" refer to undefined symbols and are identified by the third letter as follows:
usl	In a location assignment.
usc	In a constant.
usi	In a conditional (if).
uss	In argument for start.
ust	In an address tag that is not a single symbol.
usr	In a repeat count.
usd	In an array size for dimension.
use	In a formal symbol definition (with equals sign).
usx	In an argument for radix.
nca	No constants area. The constant is assembled as zero.
ipa	Illegal formal symbol assignment. It is ignored.
mnd	Macro name disagrees with name after terminate The original name is used.
uer	Micro-program error (upper case letters do not form a micro-program instruction). Same recovery as ilf.
vld	Variables location disagrees between passes 1 and 2. The location is forced to agree.
tmv	Too many variables areas. The pseudo-instruction variables is ignored.
cld	Constants location disagrees between passes 1 and 2. The location is forced to agree.
tmc	Too many constants areas. The pseudo-instruction constants is ignored.
ctl	Constants area too long (longer on pass 2 than on pass 1). The constants area is truncated.
eot	End of text reached in improper context (e.g., in the middle of a macro definition). The current pass is ended.

Appendix I

Initial Symbols

Machine Instructions

1s	1
2s	3
3s	7
4s	17
5s	37
6s	77
7s	177
8s	377
9s	777
i	10000
and	20000
ior	40000
xor	60000
xct	100000
lxr	120000
jdp	140000
cal	160000
jda	170000
lac	200000
lio	220000
dac	240000
dap	260000
dip	300000
dio	320000
dzm	340000
adm	360000
add	400000
sub	420000
idx	440000
isp	460000
sad	500000
sas	520000
mul	540000
div	560000
jmp	600000
jsp	620000
skp	640000
szf	640000
szs	640000
sza	640100
spa	640200
sma	640400
szm	640500
szo	641000
spi	642000
sni	644000
spq	650500
clo	651600
sft	660000
ral	661000
ril	662000

rcl	663000
sal	665000
sil	666000
scl	667000
rar	671000
rir	672000
rcr	673000
sar	675000
sir	676000
scr	677000
law	700000
lan	707777
iot	720000
tyi	720004
ckn	720027
cks	720033
dsc	720050
asc	720051
cac	720053
lsm	720054
esm	720055
cbs	720056
dra	720063
rbt	720237
wat	722477
sdl	723477
lei	724577
lea	724677
rer	724777
tyo	730003
dpy	730007
ivk	740000
opr	760000
nop	760000
clf	760000
stf	760010
lia	760020
lai	760040
swp	760060
cmi	760100
cla	760200
cma	761000
clc	761200
lat	762200
cli	764000
lok	770040
ulk	770041
frk	770042
qit	770043
bpt	770044
eem	770046
lem	770047
rpf	770050
lpf	770051
ram	770052
bam	770053

iam	770054
dam	770055
aam	770056
e2m	770060
e1m	770061
mta	770070
hlt	770074
dsm	770077

Pseudo-instructions
ccoun7
ccount
char
charac
consta
decimal
define
dimens
equals
flex
flexo
functi
if2
ifd
ifm
ifn
ifp
ifsym
ifup
ifz
irp
irps
irpinf
noinpu
octal
offset
ones
printc
printo
printx
radix
readin
repeat
return
spell
squoze
start
stop
termin
text
text7
twos
variab
word

Appendix II - Concise Codes

Character		Concise Code
a	A	61
b	B	62
c	C	63
d	D	64
e	E	65
f	F	66
g	G	67
h	H	70
i	I	71
j	J	41
k	K	42
l	L	43
m	M	44
n	N	45
o	O	46
p	P	47
q	Q	50
r	R	51
s	S	22
t	T	23
u	U	24
v	V	25
w	W	26
x	X	27
y	Y	30
z	Z	31
0	→	20
1	"	01
2	'	02
3	~	03
4	⊃	04
5	∨	05
6	∧	06
7	<	07
8	>	10
9	↑	11
(	[	57
)	]	55
_	|	56
-	+	54
•		40
,	$\overline{=}$	33
.	×	73
/	?	21
downshift		72
upshift		74
space		00
backspace		75
tab		36
carriage return		77
black shift		34
red shift		35
stop code		13