AMP

PROGRAMMING MANUAL

Theodore Strollo
Air Force Cambridge Research Laboratories (CRBI)
L. G. Hanscom Field
Bedford, Mass.

ACKNOWLEDGEMENT

I wish to thank Mr. Richard Bayles for his listing of Bell Fap. Many of the ideas from which "AMP" developed are those of the people at Bell Laboratories who produced Bell Macro-Fap. My sincere appreciation is extended to Mr. Charles Garman who patiently guided the development of "AMP" and made many suggestions about its format, and to Mr. Charlton M. Walter of the Air Force Cambridge Research Laboratories for making available time on the Experimental Dynamic Processor (DX-1) for the realization of AMP.

Ted Strollo

FOREWORD

In March of 1963 work was initiated at AFCRL on a new programming system for the DX-1 (Experimental Dynamic Processor).[1] The system is named "ASYS"[2] and consists of

1. A control monitor and relocating linking loader
2. An assembler ("AMP"[2])
3. A compiler ("AIDEX"[2])

The system's basic purpose is to provide flexible, easily used programming languages with good error diagnostic communication between the languages and users. Many of the features available in these languages represent the latest state of the art in assembler and compiler technology. This manual will describe the usage of the assembler "AMP".

---

1. A hardware configuration consisting of:
   2 PDP-1's
   6 Magnetic Tape Units with 2 Type "52" controls
   2 Extra Core Memories
   Color, precision, and flicker-free black and white scopes
   magnetic drum storage
   A-D converter

2. The names originate from the compiler name
   Aidex - Algorithm Interpreter for the Dynamic EXperimental Processor Group
   Amp - Aidex Machine-language Programming
   Asys - Aidex SYStem

CONTENTS

I. SOURCE PROGRAM FORMAT

A typical line of AMP coding consists of two fields:

1. The location field which contains a tag (symbol) by which other instructions may refer to this instruction.
2. The variable field which contains a symbolic expression to be decoded and ultimately loaded into core.

The Location Field

The location field begins at the left margin and is terminated by either:

1. An initial tab (which makes the location field null)
2. A string of symbol building characters followed by any symbol delimiter (which places a tag in the location field)

A tag is defined by its appearance in the location field. Some examples of tag definition follow.

```
"symbol     lac    number   "
"    bet    mul      = 12   "
```

The tag "symbol" is defined as the reference name for the location containing "lac number".

The tag "bet" is defined as the reference name for the location containing "mul = 12". Note that initial spaces do not delimit the location field in the line defining "bet".

Defined Symbols in Location Field

If a defined opcode, a macro-call, or a pseudo-op name appear in the location field, AMP will consider the line to have no tag, and the variable field scan will begin at the left margin. This means that defined opcodes, pseudo-ops, and macro-calls may not be used as tags.[1]

The appearance of the same tag in the location fields of multiple lines will be considered a multiple definition error. The tag is not redefined; the definition from the tag's first appearance is retained.

Location Field Format Restrictions.

A tag may not be defined after the first tab on a line is typed; more

1-A complete list of these symbols is presented in Appendices B, C, and D.

than one tag may not be defined on the same line.

e.g. " →| symbol lac number "

The location field is terminated by the initial tab, and the string "symbol" will be considered part of the variable field.

"symbol1 symbol2 lac number "

"symbol1" will be defined as a tag here but not "symbol2" since the location field is also terminated by the first symbol of the line. The string "symbol2" will be considered part of the variable field. Of course multiple tags may be defined to the same core location using "equ", "set", or "bss 0".[1]

The location field may not be preceded by any symbol delimiting character except " Δ ". The location field is often made intentionally blank by typing an initial tab at the left margin.

The Variable Field

The variable field begins after the termination of the location field and may contain expressions using operation codes, symbols, terms with break characters, numbers, literals, qualifiers, and/or comments. The variable field is terminated by the carriage return. If the location field is present but the variable field is absent or null, the location field tag is normally defined and a zero word is assembled in the referenced location (one word of storage is allocated for zero word)

e.g. " symbol loc " will assemble the same as

" symbol "

Operation Codes

An operation code is a three letter mnemonic code for a machine operation which is tabulated in a dictionary with its binary definition. AMP recognizes all the opcodes and input-output commands useful on the DX-1 system.[2] Since the PDP-1 allows the building of some instructions with micro-programming, AMP will interpret more than one opcode in the variable field. If the opcodes build in such a manner as to change the first

1-See the section on AMP's pseudo-ops.

2-See Appendix B

five instruction bits, the line will be considered to have an operation code error.

e.g. " lac dac b " is an error.

A line need not have an opcode if none is applicable.

e.g. " loc 3 " will assemble the same as " 3 "

" loc buffer " will assemble the same as " buffer "

Symbols

A symbol is a string of alpha-numeric characters, the first of which is non-numeric, delimited by a break character. Case shifts may occur before, within, or after a symbol and will, unless redundant or meaningless, appear in the symbol table. As a rule of thumb, symbols which visually appear identical will be considered identical by AMP regardless of case shifts.

A symbol may be of any length. Practical limitations mandate some concern on maximum length. Long symbols are difficult to type, take up more room in the symbol table, take more time to evaluate, etc. On the other hand, long symbols are generally more descriptive, more readable, etc. Each programmer must develop his own tradeoff feelings about symbol length. Three-letter symbols are not recommended for use as tags. They may conflict with defined opcodes and always dictate a search of the opcode table. Non-three letter symbols escape the opcode table search and often take less time to evaluate.

Undefined Symbols in the Variable Field

If a symbol appears in the variable field which is never defined by an appearance in the location field of a line, the symbol (unless it is declared an external system symbol[1]) is automatically defined to reference the next available location at the end of the program. This location is assembled as a zero word and the program length is increased by a count of one.

1-See the Section on Linkage

Terms of the Variable Field

A term is the string of characters between a carriage return, space, or tab and the next carriage return, space, or tab. Terms are evaluated from left to right. Term results are inclusive or'd by the evaluation routine. Spaces and tabs are therefore inclusive-or operators.

n.b. " lac a " will assemble the same as

" lac →|ΔΔ ä Δ "

" lac a+1 " will assemble differently from

" lac a +1 "

In the last example, "a" and " +1" are in separate terms whose values will be inclusive or'd, not summed.

Break Characters

Break characters are used to denote mathematical operations to be performed by the evaluation routine on symbols and numbers in each term of the variable field. A list of break characters and their meanings follows.

Break Character	Meaning
" Δ " or "→|" or ") "	Boolean inclusive - or left and right terms
" ^"	Boolean and left with right term
" + "	Add left and right terms
" - "	Subtract right from left term
" x "	Multiply left and right terms
" / "	Divide left by right term

Special Break Characters

" ' " (Single quote)	Has a value 10000. (The indirect address bit). If this character is used in conjunction with a non-indirectly addressable instruction, the line is considered to have an indirect address error.
" → "	Has the value of the current instruction location counter.

" = "	Initiates a literal.
" __ " (underscore)	Initiates a qualifier.
" - " (overbar)	References an external system symbol.

Numbers

A number is a string of numeric characters delimited by any non-numeric character. All numbers are normally interpreted as octal.[1] Special checks are made for minus zero; a minus zero will go into core when it is specified.

Literals

It is often necessary to reference a core-location which contains a constant value. One could set aside this location and name it with a tag, but the more convenient technique is to let the assembler set aside the location as a literal. Literals are referenced by the appearance of the ' = ' sign in the variable field. The expression to the right of the equals sign is the value of the literal. This expression must not contain:

1. Any pseudo-op except bci, oct, dec
2. Any macro names
3. Any external symbols
4. Another = sign

Typical literals are shown below

```
"     lac  =  7  " which assembles the same as "    lac    =    oct   7"
"     lio  =  bci.      f    .    "
"     sas  =  dzm    first loc + 31    "
```

Note that operation codes and relocatable addresses are allowed in literals.

1-This also applies to numbers used with pseudo-ops. See the modal pseudo-ops, " octal" and " decimal". Numbers following a rotate or shift instruction (0-9) are interpreted as decimal and are automatically converted to the proper bit configuration.

Qualifiers

A qualifier is a symbol which tells the assembler to qualify the present mode of assembly. For example, AMP uses qualifiers to change the interpretation of numbers from octal to decimal or vice-versa. The qualifier's range extends only to the next carriage return or until the next qualifier is encountered. At the carriage return, the mode of assembly is reset to the mode before the first qualifier in the line was encountered. Qualifiers are initiated by the " __ " (underscore character).

In the present version of AMP only two-types of qualifiers are available: the number qualifiers and the "iff" created symbol qualifier (see the section on the "iff" pseudo-op).

Number Qualifiers

" D " in the variable field tells AMP to interpret all numbers in the range of the qualifier as decimal numbers regardless of the mode of number interpretation.

" O " in the variable field tells AMP to interpret all numbers in the range of the qualifier as octal numbers regardless of the mode of number interpretation.

```
"  bss      D 100   "
"  jmp  →  +D9      "
```

Comments

A comment is initiated by the sequence "..." in either field. Comments may be used anywhere in an AMP source program. A line which contains nothing but either spaces and/or tabs and/or a comment is treated as a null line by the assembler. All characters in a comment are completely ignored. The carriage return terminates the comment.

II. RELOCATION AND LINKAGE

All AMP programs are nominally given a load origin of memory cell 0 (unless "org" is used.[1]) Obviously all of the subprograms loaded by the

1- See "org" pseudo-op

ASYS loader in a given job could not be loaded starting in cell zero[1]; they would overlap each other. Therefore, the programs are relocated such that the next program loaded begins where the last loaded program ends. Normally instructions referencing tags must have their address bits relocated since the referenced location has been relocated. Therefore, the assembler must transmit relocation information to the loader. This relocation information is called the relocation value.

Determining Relocation Value

AMP considers only two values of relocation correct; these values are relocation 0 and relocation + 1. The relocation value of a tag is +1. The relocation value of a number or opcode is 0. To compute the relocation value of a line, replace all the tags by the value +1 and all the numbers and opcodes by 0 then perform the operations indicated by the break characters.

e.g. "a" and "b" are tags

"	lac	a + 1
	0	1 + 0
	0	1
	1	relocation value
"	law	a - b "
	0	1 - 1
	0	0
	0	relocation value

The Boolean-and, inclusive-or, multiplication, or division of two relocatable elements is considered a relocation error.

Linkage

Programming systems are generally a collection of smaller programs called by a main program. These smaller programs are separately assembled and debugged. It must be possible for all of these programs to communicate with one another even though they have not been assembled

1-Normally the loader performs an initial relocation to cell 100

together. Therefore, the loader must be capable of linking programs as it loads them into core.

ASYS links programs by permitting the assignment of any memory cell as a communication cell. A cell is assigned a communication cell by tagging it with a system symbol.

e.g.

```
                In program 1
"table'     oct     5005        "
"system'    lac       → +2      "
```

The tags "system" and "table" are assigned as internal system symbols ("entry" 's) by the appearance of the " ' " (single quote) delimiter.[1] The locations represented by the tags may now be referenced by external programs. The external program must indicate to the loader that it needs linkage to these communication cells by declaring the communication cell tags as external system symbols ("externs" 's).

e.g.

```
                In program 2
"       law     table       "
"       jmp     system      "
```

The symbols "table" and "system" are declared external to program 2 by the appearance of the " ‾ " anywhere in the symbol.[2] The overbar itself will not appear in the symbol table. It is only necessary to declare a particular name external once per program. All uses of that name will then be considered references to the external name.

Linkage Restrictions

It is considered an error to declare the same name both an external and internal system symbol in the same program. The first declaration of the symbol name is retained.

AMP will not permit the use of address arithmetic on an external

1-See also the "entry" pseudo-op
2-See also the "extern" pseudo-op

system symbol

" jmp $\overline{\text{s}}$ymbol + 3 "

The address arithmetic is considered an error and will be stripped from the assembled word.

III. PSEUDO-OPS

A number of pseudo-operations (assembler as opposed to machine operations) have been provided to control the assembly process. They are classified as:

1. The linkage pseudo-ops "entry, extern"
2. The storage allocating and assignment pseudo-ops "bss, org"
3. The data generating pseudo-ops "oct, dec, bci, str"
4. The number interpretation modal pseudo-ops "octal, decimal"
5. The assembly control pseudo-ops "dup, end"
6. The defining pseudo-ops "equ, set, xsy, macro, xmacro"
7. The macro control pseudo-ops "iff, irp"
8. The assembly listing pseudo-ops (discussed in Section IV)

A pseudo-op must be used in either the location field or as the first symbol in the variable field to be properly recognized. Only one pseudo-op per line is allowed.

Argument Lists

Many of the pseudo-ops expect a list of parameters or arguments. The format of such a list is:

1. The list begins after the pseudo-op's break character and ends at the next carriage return. The initiation of a comment (by "...") terminates the list.
2. Argument names are separated by any break characters; multiple break characters (such as multiple spaces) are ignored.
3. The list may have any number of arguments.

    1. The linkage pseudo-ops

        "entry"

The "entry" pseudo-op inserts each name from the argument list into the symbol table, and when each name is subsequently encountered in the location field, the name is defined as an internal system symbol - a program entry. The "entry" pseudo-op is useful because one can tell from a glance at the entry list which internal system symbols are defined by the program. Symbols defined by "entry" would be identically defined by using the " ' " (single quote) delimiter on the name at its appearance in the location field.

Symbols appearing in the "entry" list, but not subsequently in the location field are not defined as entries. The "entry" pseudo-op may be used anywhere in the program. A typical technique is to put the entry list very near the beginning of the program before any instruction lines will be encountered.

```
e.g.  "   amp     floating     point     add
                  entry        fadd"
```

"extern"

The "extern" pseudo-op inserts each name from the argument list into the symbol table, and when each name is encountered in the variable field of an instruction, the name is defined as an external system symbol. The "extern" pseudo-op is also useful because one can tell from a glance at the extern list which external system symbols are defined by the program. Symbols defined by "extern" would be identically defined by using the " ‾ " (overbar) anywhere in the symbol at its appearance in the variable field.

Symbols appearing in the "extern" list which never appear in the variable field of any instruction lines are not defined as externs. If the same symbol appears twice or more in an extern list, only the first appearance is entered in the symbol table. The "extern" pseudo-op may be used anywhere in the program. A typical technique is to put the "extern" list near the beginning of the program.

e.g. " ... amp subroutine tradeoff
entry tradeoff crlmat
extern order, matpos, matptr "
⋮

2. The storage allocating and assignment pseudo-ops "bss"

It is often necessary to reserve a block of storage for an array or input-output buffer. The pseudo-op bss will set aside a block of storage by advancing the loader's program origin and the assembler's instruction location counter by the amount expressed in the field to the right of the "bss". The storage area is merely reserved and not zeroed.

A symbol used in the location field of the "bss" pseudo-op is defined before the assembler's instruction location counter is advanced. ("bss" is an acronym for block started by symbol)

e.g. "buffer A bss 100 "

The tag "buffer A" is defined to reference the first location of the storage area.

The "bss" pseudo-op may be used with a zero count.

```
"A      bss     0
 be     bss     0
 done           1    "
```

The statement, "bss" 0", is useful for equating the definition of tags. Tags "A", "be", and "done" will all reference the location containing the 1.

The expression following "bss" must have relocation value 0 (an absolute expression). Tags may be referenced in the bss pseudo-op as long as the net relocation values is 0. No external system symbols, macro-calls, or other pseudo-ops may appear in the expression following "bss". Tags used in the variable field must be previously defined or AMP will consider the use of the tag a phase error.

e.g.

The following "bss" has a phase error

```
"     q          loc
           :
           :
      symboltable      bss      b - q + 1
           :
           :
      b    loc    "
```

The following "bss" is properly used

```
"     q    loc
           :
           :
      b    loc
           :
           :
symboltable     bss     b - q + 1     "
```

When AMP finds a phase error or a relocation error in a "bss", the "bss" is assembled as a "bss 0" (i.e., no storage is allocated) but the tag (if present) in the location field is defined in the normal manner.

"org"

Since the sequence break channels of the PDP-1 use locations 0-77, the ASYS loader will normally load the first assembled program starting at location 100, and subsequent programs will be loaded sequentially. At times it is important to put a particular program in specific area of core. The "org" pseudo-op allows the programmer to set the origin of his program absolutely. It may be used anywhere in the program. Typically "org" is used near the beginning of the program.

The "org" pseudo-op may be used only once in a program. The expression following the "org" must have relocation value 0, and must not contain: previously undefined tags, externs, macro-calls, or pseudo-ops.

3. The data generating pseudo-ops

"oct" and "dec"

The "oct" and "dec" pseudo-ops assemble each number in the parameter list in a sequential core location. The numbers may contain sign

information (" + " or " - ") and must be separated by commas. If a tag is used in the location field, it is defined to reference the first number of the list. After "oct" numbers are considered octal to the carriage return; after "dec" numbers are considered decimal to the carriage return.

e.g.
```
"name   oct 1, 2, -0, 7, +6, 17 "
"       oct    77    "
"       dec    -10, 3, -4, 91   "
```

bci and str

"bci" and "str" pack the BCD characters following the break character[1] three to a word left justified until the next normal " . " (this terminating period is not packed) appears in the string. As soon as one word is filled, packing resumes using the next sequential word. The "," (comma) has a special meaning with these pseudo-ops. The comma means pack the next character regardless of what it is; then resume normal packing. The comma can be used to put the BCD code for "." in the packed word by typing

",." for each period desired in the packing

The comma will appear in the packing only by typing

",," for each comma desired in the packing

Carriage returns may be used in the string and will be packed.

When the normal delimiting period is encountered, "bci" fills the remaining partially packed word with octal "76's" unless the last word is full then it does no more packing. On the delimiting period, "str" fills the remaining word with octal "13's", and, if the last word is full, it will generate a new word with three 13's (packed).

If a tag is used in the location field, it is defined to reference the first packed word generated by the string. The characters between the terminating period and the carriage return are ignored.

```
"     bci.   error   gon
   .   "
"string str.    N    ow   is   the   time,...   "
```

1- A " . " will delimit "bci" and "str" as well as normal AMP break characters.

4. The number interpretation modal pseudo-ops

"octal"

The "octal" pseudo-op sets the mode of assembly to octal. All numbers will then be interpreted as octal unless the numbers are qualified by D , appear in the "dec" pseudo-op field, follow a rotate or shift instruction, or appear in the argument field of a "dup" pseudo-op. The normal mode of assembly is octal.

decimal

The "decimal" pseudo-op sets the mode of assembly to decimal. All numbers will then be interpreted as decimal numbers unless the numbers are qualified by O or appear in the "oct" pseudo-op field.

The "decimal" and "octal" pseudo-ops may be used at any time in an assembly. They may be used multiple times to switch the mode back and forth from octal to decimal.

5. The assembly control pseudo-ops

dup M, N

The "dup" pseudo-op tells the assembler to consider the next block of "M" lines as though it appeared in the source program "N" times. M and N must be absolute expressions (numbers are interpreted as decimal).

```
e.g.        "     dup    1, 39
      buffer  oct          202020 "
```

This will set up a table with 39 (decimal) locations each containing 202020 (octal). The tag will be defined during the first iteration of the dup and will not be multiply defined during the subsequent iterations. The tag will reference the first location in the table.

If M or N are negative or if M is zero the "dup" is meaningless and will be considered an error. A "dup" with N equal to zero is acceptable and very useful inside macro-expansions (see the sections on "macro", "iff"). When N equals zero, the dup means ignore the next M lines.

A "dup" may not be used within the range of another "dup" unless the

iteration count of the inner "dup" is zero.

end

The last statement of the program must be the "end" pseudo-op which signals pass 1 to call pass 2 and pass 2 to return to the ASYS monitor. If the "end" statement contains a variable field with a relocatable (relocation value 1) expression, the ASYS loader will consider the program to be the main program and will assemble an execution jump[1] to the address specified by the variable field.

e.g.
```
"    amp    adaptive        filter design
            entry    adapt
adapt       .        clf    7
            .
            .
       end        adapt  "
```

6. The defining pseudo-ops "equ" and "set"

The AMP language provides the "equ" and "set" pseudo-ops to enable the programmer to:

1. expand the opcode dictionary
2. reference a parameter symbolically which is used very often in the program and may be subject to change.
3. equate the definition of two program tags.

The "equ" and "set" pseudo-ops must have a symbol in the location field. The expression following equ is the definition of the symbol. This expression may have relocation values 0 or 1 defining the symbol as absolute or relocatable, respectively.

e.g.
```
"    cad    equ    iot      1600 ... define Convert A to D "
   "datatape      equ      1  "
```
Every time "datatape" is referenced, the octal value 1 replaces the symbol.
```
" a       lac    =        D      75
  b       equ    a        "
```
—1-The execution jump is described in the ASYS Monitor Manual.

The definition of symbol "b" will be the same as the definition of tag "a", and both will reference the location containing the "lac = D 75 "

If the " → " (right arrow) break character is used in the variable field of the "equ" pseudo-op, the " → " will have the value of the next location to be assigned by "amp".

```
e.g.    "    a     dio     input
             b     equ   → -1    "
```

"b" is again equated to "a" in definition.

The pseudo-op "set" is essentially the same as "equ" with the important exception that symbols defined by "set" may be redefined by subsequently using "set".

```
e.g.    "n   set   20
         n   set   31  "
```

This feature is particularly useful for generating tables in conjunction with the "dup" pseudo-op.

```
e.g.    "n   set   1
             dup   2, 18
                 n
         n   set   n x 2   "
```

This sequence will generate a table of powers of two from 2^0 to 2^{17}.

The variable field of the "set" and "equ" pseudo-ops must contain no previously undefined symbols (phase errors), external system symbols, macro-calls, pseudo-ops, or literals. The location field of the "equ" pseudo-op must contain an undefined symbol. The location field of the "set" pseudo-op must contain either an undefined symbol or a symbol previously defined by "set".

"xsy"

The pseudo-op "xsy" is provided to expunge the definition of a list of names from the symbol table and allow these names to be redefined. If an

undefined name is used in the "xsy" list, the name is ignored.

e.g.
```
"    xsy    fadd,    fsub,    fmpy,    fdvd,  float,  ifix
     xsy    gon   "
```

"macro and xmacro"

Often it is necessary to use the same sequence of instructions at many places in program. The "macro and xmacro" pseudo-ops are provided to give a name to this sequence of instructions (called the prototype) and to allow the sequence to be generated by the appearance of this name (the macro call) in the program.

e.g. the sequence
```
"rcl    9
 rcl    9  "
```
which exchanges the contents of the accumulator- and input-output register can be given the name "swap"
```
"swap   macro
        rcl    9
        rcl    9
        end  "
```
So that the subsequent appearance of the name "swap" in the program will cause the sequence
```
"rcl    9
 rcl    9  "   to be generated.
```
The recurring instruction sequence need not always be identical in each appearance, but parts of the sequence can be modified by various parameters. These parameters can be provided by giving the macro-call an argument list.

e.g. The macro-call
```
"     select    2     "
```
can generate the sequence

```
        law     2
        muf
        jmp     →-1   "
```

while the macro-call

```
   "    select      3     "
```

can generate the sequence

```
   "    law     3
        muf
        jmp     →-1   "
```

By defining the name "select"

```
"select     macro   tapeunit   "
"           law     tapeunit
            muf
            jmp     →-1
            end     "
```

Defining Macros

The macro name must appear in the location field of a line whose variable field contains the word "macro" followed by a list of names (which may be blank) separated by any break characters. The total number of names in this list is the expected argument count. These names are called the "dummy arguments" of the macro definition. They are used in the prototype wherever the sequence is to be modified by a parameter in the macro-call. The carriage return or initiation of a comment (by "...") terminates the "dummy argument" list. A maximum of 40 (decimal) "dummy arguments" is allowed. All of the next lines are considered part of the prototype until the line containing the "end" statement is encountered. This "end" statement terminates the prototype (not the assembly).

All of the characters in the line containing "end" are ignored in AMP, (except, of course, for the word "end"). This means arbitrary comments may be put anywhere in the macro "end" statement. Typically, the macro

name is repeated in the location field to show at a glance the beginning and end of each macro definition.

```
e.g.        "zero   macro       beginning,     amount
                    law         beginning
                    dap         → + 1
                    dzm           ..
                    idx         → - 1
                    sas   =   dzm     beginning + amount
                    jmp         → - 3
              zero  end   "
subsequent use of "zero   buffer,      100    "
          produces
          "         law   buffer
                    dap   → + 1
                    dzm     ..
                    idx   → - 1
                    sas = dzm         buffer + 100
                    jmp   → - 3    "
```

Any programming code legal in AMP can be used in the macro prototype with the exception of the pseudo-op "macro" and the pseudo-op "end". Macro-calls can be used in the macro prototype to any level of nesting (until push-down storage is exceeded).

Macro Calls

The macro-call is the subsequent use of the macro by the appearance of its name and an argument list, arguments may contain expressions and must be separated by commas. A tag may be defined in the location field of macro-call; the tag will reference the first location generated by the call.

Created Symbols in Macro Calls

In long macro definitions it is some times useful to use reference tags in some location fields of the prototype lines. However, these tag names

must be supplied as parameters for the macro-call so that they may be unique for each macro-call (otherwise the tag would be multiply defined on multiple calls of the macro name)

e.g. zero could be defined

```
"zero    macro    a, b, c
         law      a
         dap      c
 c       dzm      ..
         idx      c
         sas  =   dzm    a + b
         jmp      c
 zero    end  "
```

and then zero would be used as

```
"zero    buffloc, 237,   name1   "
"zero    buffloc2 , 100, name2   "
```

The necessity of creating unique names can become cumbersome for the programmer; so, if terminal arguments are left off the macro-call, unique names will be created by AMP's macro-compiler to fill out the argument list until the expected argument count is reached. The structure of the created symbols is a " . " followed by numeric characters. The first created symbol is

" .0 "
the next
" .1 "
later
" .127 "
etc.

It is not recommended that programmers use symbols having the form at ".NNN" where the N's are numbers because these symbols may conflict with the created symbols generated by AMP.

Now zero (as defined on page 20) could be used as

```
        zero    first,    amount
```

leaving off the third argument and would assemble as

```
"       law     first
        dap     .210
.210    dzm     .
        idx     .210
        sas  =  dzm       first + amount
        jmp     .210    "
```

External system symbols in macros

A macro may reference an external system symbol by putting an " - " over the symbol in the prototype or using an extern list in the prototype. In some macros, the prototype references, as an external system symbol, the name of the macro itself. These macros are handled by "xmacro"

e.g.

The macro "gdn" (an acronym for get decimal number input from the typewriter) calls the external routine "gdn" and is defined, in its simplest form, as

```
"gdn    xmacro
        jsp     gdn
 gdn    end     "
```

The rules for defining "xmacro" are the same as those for defining "macro"'s.

If a name has been defined as an "xmacro", and that name appears in the variable field of a line, but is not the first symbol of the variable field:

1. the appearance will not be considered a macro-call

2. the name will automatically be considered a reference to an external system symbol.

Therefore, the appearance of a " jsp gdn " in any line of a program where "gdn" is defined[1] as an "xmacro" will cause "gdn" to be considered an external system symbol.

1-"gdn" is a system macro pre-defined in AMP. See Appendix D.

Nesting macro-calls

A macro-call may generate (from the prototype) another macro-call. This next macro-call may generate (from its prototype) still more macro-calls. AMP will handle these nestings.

e.g.

A macro "flac" useful for floating point operations might be defined:

```
"flac    macro       address
         lac         address
         lio         address + 1
 flac    end    "
```

"flac" is used to perform a floating point loading of the accumulator and input-output register.

A macro "flop" might be defined to call the floating point typewriter output routine and could use "flac" to load the floating point number to be typed out.

```
"flop    xmacro      number,      format
         flac        number
         jda         flop
         format
 flop    end    "
```

Note that the "flop" prototype contains a call to the "flac" macro.

The macro-call

```
"flop    sum,   =    5005    "
```

would then generate

```
"        lac         sum
         lio         sum + 1
         jda         flop
         =   5005    "
```

"flop" would be considered an external system symbol (note that the "xmacro" pseudo-op defines "flop") and "5005" is the code which "flop" will use to establish the typing format.

7. The macro control pseudo-ops

"iff"

AMP provides the capability for conditional assembly of portions of the macro prototype. Sections of the prototype will be assembled or not, conditioned on the argument list of the individual macro-call.

e.g.

Using the "flop" macro as it was defined in the previous section, some redundant machine code would some times be generated. Suppose the number to be typed out were already loaded into the accumulator and in-out register. Then it would be useful if "flop" could conditionally assemble the call to the "flac" macro.

e.g. " flop , = 5005 "

should assemble as

```
"    jda    flop
       =    5005    "
```

The first argument of the "flop" macro-call has been omitted. Note that this is not a terminal argument of the call and symbols are not created to fill the omission. However, the missing argument is a null string.[1] In order to detect this null string the "flop" macro may be defined.

```
"flop   xmacro      number,      format
        iff     0,  number
        flac    number
        jda     flop
        format
"flop       end     "
```

The "iff" line means: assemble the next line if, and only if, it is false that the argument "number" is a null string.

Specifically if the string is null, the "flac" call will not be assembled.

If the string is non-null, the "flac" call will be assembled.

1-A string which contains either no characters or just "space" and "tab" characters.

The complete format of "iff"

"iff truth value, argument, bcdstring"

The "iff" pseudo-op expects three fields (the third field may be blank). The truth value is any absolute expression.

A truth value of zero means "false"

A non-zero truth value means "true"

The argument is any of the names appearing in the macro definition dummy argument list. The bcdstring is any string of characters.

The full meaning of "iff" is: assemble the next line if, and only if, it is true (false)[1] that the argument is the same character by character as the bcdstring.

e.g. A macro to display a point on any scope might be defined.

```
"display    macro   x, y, scope
            lac     x
            lio     y
            iff     1, scope, bw
            dpy'
            iff     1, scope, precision
            dpp'
            iff     1, scope, color
            lot'    710
 display    end  "
"display    x, y, bw   "
```

would assemble as

```
"           lac     x
            lio     y
            dpy'    "
"display    x, y, color   "
```

would assemble as

```
"           lac     x
            lio     y
            lot'    710   "
```

1-Depending on the truth value.

Third field blank in "iff"

When "iff" is used without the third field, AMP will do conditional assembly based on whether or not the argument specified is null.

"iff truthvalue, argument" has the meaning:

Assemble the next line if, and only if, it is true (false) that the argument is null.

The "C" qualifier is useful with "iff" when the third field is absent. This qualifier tells AMP to determine whether or not the argument was a created symbol and conditionally assemble on this basis.

The C qualifier must appear immediately before the argument name in the second field of the "iff".

"iff truthvalue, C argument" means

Assemble the next line if, and only if, it is true (false) that the argument is a created symbol.

e.g.

```
"divide     macro   arg,     option
            div     arg
            iff     1, C option
            hlt
            iff     0, C option
            option
divide      end     "
```

The divide macro could be used in one of three ways:

1. "divide number"

   would assemble as

```
"       div     number
        hlt     "
```

2. "divide number, nop "

   would assemble as

```
"       div     number
          nop   "
```

3. "divide number, jda diverr "

would assemble

```
        div   number
        jda   diverr   "
```

"dups" and "macro"'s

A macro-call may appear within the range of a "dup", and a "dup" may appear within a macro prototype. A "dup" with a zero iteration count is very useful in conjunction with conditional assembly in macros.

e.g. A macro to conditionally type an error diagnostic then halt can be defined.

```
"error  macro   ac, io, option
        iff     1, option, notype
        dup     2, 0
        lio     = bci.err.
        jsp     type3chars
        lac     ac
        lio     io
        hlt
error   end     "
```

"irp"

It is possible for a macro to iterate a set of instructions within the prototype an indefinite number of times by using the "irp" (indefinite repeat) pseudo-op. The number of iterations is determined by the number of sub-arguments supplied to the macro-call. Sub-arguments are separated by commas with the entire list of sub-arguments placed between parentheses.

e.g.

```
" clear   macro   register
          irp     register
          dzm     register
          irp
  clear   end     "
```

The "clear" macro will store zero in each register referenced by the sub-arguments.

```
                clear (a, b, c, d)
will assemble:
        "  dzm    a
           dzm    b
           dzm    c
           dzm    d      "
                both
        "  clear (a)     "
                and
        "  clear  a      "
will assemble:
        "  dzm    a      "
```

The first "irp" statement marks the beginning of the repeat range. The argument name in the variable field of the first "irp" statement determines which argument conditions the repeat. The second "irp" statement marks the end of the repeat range and initiates a repeat of the range using a new sub-argument until all of the sub-arguments are exhausted. If no parentheses enclose the repeat conditioning argument, the range will not be repeated, and the first argument will be used.

A macro-call nested within the range of an "irp" may itself open a new "irp" range. The macro-compiler strips the outermost pair of parentheses from the macro argument at each new level of macro-nesting. To transmit sub-arguments to deeper level macro's, it is necessary to enclose them in multiple matching pairs of parentheses. (one additional matching pair/ nesting).

e.g.

A macro named SUBSQ2 to compute

$$(a^2 + b^2 + c^2 + \ldots)^2 \quad + \quad (d^2 + e^2 + f^2 + \ldots)^2 \quad + \quad (g^2 + h^2 + i^2 + \ldots)^2 + \ldots$$

would be called by:

```
"    SUMSQ2     ( (a, b, c) (d, e, f) (g, h, i) )    "
```

and would be defined:

```
"    SUMSQ2     macro    arglist2
                dzm      sum2
                irp      arglist2
                SUMSQ1   arglist2
                dac      temp
                mul      temp
                scl      9
                scl      8
                add      sum2
                dac      sum2
                irp
     SUMSQ2     end

     SUMSQ1     macro    arglist1
                dzm      sum1
                irp      arglist1
                lac      arglist1
                mul      arglist1
                scl      9
                scl      8
                add      sum1
                dac      sum1
                irp
     SUMSQ1     end      "
```

An "irp" pseudo-instruction cannot occur explicitly within the range of an "irp". The first nested appearance of the "irp" is used to terminate the range and cannot initiate a new range. The only way to nest "irp" ranges with AMP is through nested macro-calls. The "irp" pseudo-op is not

defined outside the range of a macro-call.

IV. THE ASSEMBLY LISTING AND ERROR DIAGNOSTICS

The optional printed output of the AMP language is called the assembly listing. It consists of a print out of the input program together with the octal representation of the binary words produced by AMP and error diagnostics. A portion of a typical assembly listing appears here;

```
Page 3    Initialization and Main Loop Pass - 2
 O          0157  20 0000   115          lac     dac    a
            0160  60 0240   116          jmp     xp5
            0161  70 0005   117   xp12   law     symtab
            0162  60 0210   118          jmp     extension
 R                         119          bss     symtab
```

The left most portion of the page contains the error diagnostics in the form of single, upper case letters which are defined in the error flags discussion. The next column is the octal value of the assembler's instruction location counter. The ilc for printouts is started at 0 unless the "org" pseudo-op is used in which case the ilc for printouts is started at the specified origin. The following column is the octal representation of the assembled word. Next appears the decimal line count specifying the line number in the source program. The remainder of the line is the repeat of the original text associated with this line number in the source program.

Generated Lines

Lines which are internally generated by the assembler print with special formats. A line generated by iterations of a "dup" pseudo-op (after the first iteration) will neither index nor print the decimal line count. A line generated by a macro will neither index nor print the decimal line number but will, in place of the line count, print a decimal number (in the format ".ON" where N is a decimal number) which indicates the nesting level of the macro-call, the nest count begins at ".01", lines generated by a macro-call within a macro-call index the nest count to ".02". Lines

generated by a macro-call within a macro-call which is itself within a macro-call index the nest count to ". 03".

The decimal line count will, because of the special printout rules for assembler generated lines, always be the correct line number to associate with that line of the source program. This feature is very useful for editing tapes with Expensive Typewriter.

Pseudo-ops which Print with Special Formats

Non-Output Pseudo-ops

Entry, extern, org, octal, decimal, macro, xmacro, dup, iff, xsy, irp, ttl, rem, and eject all print with no listing of the i.l.c. or assembled word since these pseudo-ops generate no binary output. In addition, all of the lines of a macro definition, all lines which are null strings, as well as the line of a macro-call will print without the i.l.c. or assembled word again because no binary output is generated in these cases.

Data Generating Pseudo-ops

The data generating pseudo-ops (bci, str, oct, dec) are capable of generating more than one output word. When this is the case each word and the new i.l.c. appear on a different line. These additional lines do not affect the decimal linecount.

Storage Allocating Pseudo-ops

The "bss" pseudo-op prints the value of the i.l.c. before the bss is evaluated but does not print an assembled word. In the position of the i.l.c. column the "end" pseudo-op prints the program length (in octal).

Symbol Defining Pseudo-ops

The "equ" and "set" pseudo-ops do not print the i.l.c. but in the assembled word column print the location field symbol definition.

The Assembly Listing Pseudo-ops

"ttl"

Listings are always page numbered, and, if the "ttl" pseudo-op is used

a title will appear on the same line as the page number. More than one "ttl" line may be used in a program. Each new "ttl" line wipes out the old title and ejects the paper to the top of a new page.

The title itself follows the word "ttl" and its break character then continues until the carriage return.

"eject"

The "eject" pseudo-op causes the assembler to begin printing the remainder of the listing at the top of a new page. As many "ejects" as desired may be used in a program.

rem

The "rem" pseudo-op initiates a remark, but, unlike, the "..." which initiates comments, the word "rem" will not appear in the assembly listing. This pseudo-op is provided to create neat assembly listings.

ERROR FLAGS

AMP will in almost all cases attempt to continue the assembly when it finds syntactical errors. However, when a paper tape parity error is detected or when a table is exceeded, the error is considered fatal. Some special checks in AMP have been inserted to detect fatal errors in the language itself or machine errors. Fatal errors of any type are very rare, but when a fatal error occurs, one of three characters is typed on-line, then the computer comes to an irrecoverable halt.

"p" - paper tape parity error

"t" - table size exceeded

"e" - machine error or error in AMP.

When either a "t" or an "e" error occurs, record the status of the accumulator (which represents the program counter contents where the error was found) and refer to the assembly listings of AMP itself.

On non-fatals errors AMP will unconditionally type (on-line) the line of the assembly in question. This line will contain all of the elements of the assembly listing (It is a segment of the assembly listing). The one-letter

diagnostics or error flags are described in detail here. They are tabulated and described briefly in Appendix E.

"K"

"K" means kill the assembly of this line and treat it as a null string. This flag is always followed by another flag which tells the type of serious error which caused this line to be killed.

"F"

Field or format error. The format of the line receiving this flag conflicts with the basic input format required by the assembler; for example, the presence of a number (not part of a symbol) in the location field is a field error. A use of the pseudo-op "equ" without a preceding symbol in the location is a format error. The attempt to use two pseudo-ops on the same line is another format error.

"X"

This line contains an extern-intern linkage error. The attempt to declare a name both external and internal to the same program is flagged by "X". The attempt to declare a defined opcode, pseudo-op, or macro-call as either an extern or entry will be flagged "X".

"P"

The use of the pseudo-ops "equ", "set", "bss", or "org" with a previously undefined name in the variable field will be flagged with "P" meaning phase error.

"R"

An expression whose relocation value differs from 0 or +1 will be flagged as a relocation ("R") error. In addition, pseudo-ops such as "org" requiring an absolute expression in the variable field will flag as "R" errors any value of relocation except 0.

"M"

An attempt to redefine a previously defined symbol (except the redefinition of a "set symbol" by the "set" pseudo-op) will be flagged "M" - multiply defined symbol.

"L"

Lines with errors in literals are flagged "L". Such an error would be the use of a data generating pseudo-op in a literal which threatens to generate more than one word or the use of a macro-call in a literal.

"N"

A number which is being interpreted as octal and contains an "8" or "9" or numbers which would not fit in a PDP-1 word size of 18 bits are flagged with the "N" flag - number error.

"D"

When attempts are made to nest "dup" 's, to give the "dup" a 0 or negative line count, or to give the "dup" a negative iteration count, the "dup" is flagged with a "D" or dup-error flag.

"Q"

An illegal qualifier or the illegal use of a qualifier is flagged with a "Q".

"A"

Attempts to perform address arithmetic on external system symbols are flagged with the "A" flag.

"O"

When several machine operation codes are included on the line (for micro-programming), the assembler performs a compatibility check. If the operations are not compatible, the assembler presents the "O" flag. In addition, a rotate or shift command which specifies more than 9 shifts is flagged as an operation code error.

"I"

An indirect address " ' " specified with a non-indirectly addressable operation code is flagged with an "I".

FOREWORD TO SECTION V

Section V of the Manual discusses the use of AMP on the DX-1 installation. First, a brief description of the use of the monitor and loader is presented. This discussion appears in considerably more detailed form in the write-up on the ASYS-1 monitor. The section does, however, refer to the monitor only in connection with AMP and may clear up some points on monitoring AMP assemblies. The next section discusses the sense switch options of AMP which control the symbol table dump and assembly listing features.

V. USING AMP ON THE DX-1 INSTALLATION

AMP is used on the DX-1 in conjunction with the ASYS-1 monitor. The monitor is the first record of the ASYS-1 magnetic system tape. Once the monitor is read into core 1*, it handles all of the control necessary to initiate and complete AMP assemblies.

Required Magnetic Tapes

AMP and the AIDEX System require a total of three magnetic tapes. Two of these tapes are scratch tapes and are available for user scratch immediately after the assembly and loading operations are completed. The ASYS-1 system tape must be mounted on a transport defined as Unit 1. Two additional tapes must be mounted and defined as Units 2 and 3. Tape Unit 2 will contain the relocatable binary output of AMP. Tape Unit 3 is the intermediate tape for AMP and the monitor 'save' and 'restore' tape. (The monitor control options are explained in the write-up on the ASYS-1 monitor).

Selecting the AMP Language for Assemblies

AMP is most conveniently selected by placing the line:

" ... amp (arbitrary text for identification) "

at the beginning of the source program. By starting the computer at 12000 (octal) with an AMP program in the reader, the monitor will read this first line and recognize that an AMP program follows. The monitor then reads in and initiates Pass 1 of AMP from the magnetic system tape. At the completion of Pass 1 the monitor automatically reads in and initiates Pass 2 of AMP from the magnetic system tape. At the end of Pass 2, AMP returns control to the monitor which will look for more control statements.

Multiple Assemblies

Additional assemblies are made by repeating the process just described. A new program is placed in the reader, and the computer is started at 12000 (octal). If more than one program is to be presented on the same paper tape

*-By using the read in monitor paper tape which is always near the console, the paper tape must be read into Core 0.

1. Each program must be terminated with a FIO-DEC stop code.

2. Sense switch 3 must be up so that AMP (which normally ejects paper tape) will stop reading paper tape on FIO-DEC stop codes.

Loading the Binary Output Tape

When all the assemblies have been completed, the user may type 'load' on the on-line typewriter. The assembled programs will be loaded into core 0 in the order by which they were assembled. Constants are automatically defined and linked during the loading process. Finally, after the loading operation is completed, the user must type 'tra address' where 'address' is either an octal number or a system symbol referencing the start of the program. The monitor will now search the library file (part of the system tape), if necessary, and then transfer control to the specified 'address'.

Sense Switch Control of AMP Printouts

Normal operation of AMP involves all the sense switches and all of the test word switches being off (down). * This procedure will yield:

1. An on-line type-out of the first line of the program used to select AMP.

2. No assembly listing except for the unconditional assembly listing of error lines on the on-line typewriter.

3. An on-line typeout of the symbol table dump.

Assembly Listings

The AMP language encourages the use of its assembly listing feature by offering this listing using a variety of input-output media; the listing is available on:

1. The on-line punch if SS 4 is up for subsequent flexowriter listing.

2. The on-line typewriter if SS 5 is up.

3. A magnetic tape for off-line (1401) listing if SS 6 is up. This 1401 listing feature assumes the availability of yet another magnetic tape transport (Unit 4).

*-The use of sense switch 3 is discussed under Section V - Multiple Assemblies

All of the media may be used simultaneously; AMP uses the sequence break system in its I-O routines. However, the listing will be made only as fast as the top speed of the slowest input-output device being selected. Of course, the typewriter is slowest, the punch is next, and the mag-tape is the fastest. Indeed, magnetic tape listings add very little time to the overall assembly time with no listing.

Symbol Table Dumps

When the time required to get an assembly listing is not available, AMP offers a symbol table dump. This dump consists of a heading for the type of symbol followed by a tabulation of each symbol of this type and its octal definition. The headings are:

1. 'und' - undefined symbols
2. 'rel' - relocatable symbols (relocation value 1)
3. 'abs' - absolute symbols (relocation value 0)
4. 'ent' - entry symbols (internal system symbols)
5. 'ext' - extern symbols (external system symbols)

It is possible to selectively dump only symbols of certain types (1 - 5) by putting up test word switches.

Bit 17 up means dump type 1

Bit 16 up means dump type 2

Bit 15 up means dump type 3

Bit 14 up means dump type 4

Bit 13 up means dump type 5

If bits 11 - 17 are all down, all of the symbol types are dumped.

On-line typewriter symbol table dumps may be terminated at any time by putting up SS 1. This results in a return of control to the monitor.

When assembly listings are being made on the punch or 1401 mag-tape, the symbol table dump is automatically printed on these media.

Sense switch 2 is presently unused by AMP.

SECTION VI

AMP Programming Manual Appendices

Appendix A	A Brief Description of the Assembly Process
Appendix B	A Table of AMP Operation Codes
Appendix C	A Table of AMP Pseudo-Ops
Appendix D	A Table and Description of the AMP System Macros
Appendix E	A Table of AMP Error Flags

APPENDIX A

A Brief Description of the Assembly Process

Pass 1

During Pass 1 AMP reads the program from the source paper tape copying it with auxiliary operational and error diagnostic information on to the collation or intermediate magnetic tape. Location field tags are defined while storage is allocated by Pass 1. Macro definitions are entered into the symbol table, and macro-calls are expanded. Iterations of dup's and irp's are completed. Pass 1 is terminated by the end statement which rewinds the collation tape and instructs the monitor to read in Pass 2.

Pass 2

During Pass 2 AMP reads the symbolic program from the collation tape, converts symbolic to binary, writes the assembled core words with relocation information on the relocatable binary magnetic tape (the punch tape), and creates the assembly listing (if requested).

Undefined symbols and relocatable constants are assigned to storage locations at the end of the program. Symbolic and absolute constant linkage information is written on the punch tape. Pass 2 is terminated by the end statement which writes the end of file on the punch tape, back spaces the punch tape one record (in case additional assemblies or compilations follow), then returns control to the ASYS Monitor.

APPENDIX B

Table of AMP Operation Codes

Operations with an * are
unique to the DX-1 Installation.

mnemonic	code	mnemonic	code
add	400000	lac	200000
and	020000	lat	762200
asc	72xx51	lap	760300
* bks	720375	law	700000
cac	720053	lem	720074
cal	160000	lio	220000
cbs	720056	loc	000000
cks	720033	lsm	720054
cla	760200	mel	720036
clf	760000	mes	720035
cli	764000	mic	720075
clo	651600	mri	720066
cma	761000	mrf	720067
dac	240000	mul	540000
dap	260000	muf	720076
dio	320000	nop	760000
dip	300000	opr	760000
div	560000	ppa	720005
dpp	720407	ppb	720006
dpy	720007	ral	661000
dsc	72xx50	rar	671000
dzm	340000	rcl	663000
eem	724074	rcr	673000
* emn	724074	* rew	720175
* emf	720074	ril	662000
esm	720055	rir	672000
* fws	721175	rpa	720001
* gwe	721266	rpb	720002
* gwo	721366	rrb	720030
hlt	760400	* rte	721675
idx	440000	* rto	721775
ior	040000	sad	500000
iot	720000	sal	665000
isb	72xx52	sar	675000
isp	460000	sas	520000
jda	170000	scl	667000
jmp	600000	scr	677000
jsp	620000	sft	660000

	mnemonic	code		mnemonic	code
	sil	666000		sza	640100
	sir	676000		szf	640000
	skp	640000		szo	641000
	sma	640400		tyi	720004
	spa	640200		tyo	720003
	spi	642000		usk	640600
*	sre	721666	*	wte	721275
*	sro	721766	*	wto	721375
	stf	760010		xct	100000
	sub	420000		xor	060000

APPENDIX C

Table of AMP Pseudo-ops

mnemonic	brief description
bci	packs bcd character strings into core locations. Fills the last location with octal 76's.
bss	reserves a block of storage locations.
dec	converts decimal numbers to binary words for core locations.
decimal	sets the number interpretation mode to decimal.
dup	repeats the assembly of the next specified number of lines a specified number of times.
eject	modifies the assembly listing by ejecting the listing to the top of a new page.
end	marks the termination of the source input.
entry	defines symbol s as internal system symbols.
equ	makes symbol definitions equal to arbitrary expressions.
extern	defines symbols as external system symbols.
iff	uses Boolean expressions to condition the assembly or non-assembly of the next line.
irp	enables the iteration of a group of instructions in the macro prototype with the number of iterations dependent on the argument list of the macro-call.
macro	defines a macro name to represent a macro prototype
xmacro	same as macro but used where the macro name is itself the name of an external subroutine called by the prototype.
oct	converts octal numbers to binary words for core locations.

octal	sets the number interpretation mode to octal.
org	sets the origin of a program to an absolute value.
rem	treats the entire line as a null line. The "rem" and its break character will not appear in the assembly listing.
set	same as equ but allows symbols to be redefined by subsequent use of set.
str	same as bci but fills with octal 13's and will generate an entire word of 13's (131313) if the last packed word needs no filler.
ttl	titles each page of the assembly listing.
xsy	expunges symbol names from the symbol table allowing the name to be redefined.

APPENDIX D

AMP System Macros

The AMP System Macros have been pre-defined and retained in the AMP symbol table. These macros are available for any AMP assembly and must be expunged (with the "xsy" pseudo-op) to be redefined. All operations with an * apply only to the 28 bit mantissa 7 bit exponent floating point package.[1] Other floating point macros apply to both FPT-18[2] and FPT-28.

Macro Name	Use and Description
* f'abs (floating absolute)	takes the absolute value of the floating-point AC-IO leaving this result in the AC-IO. If fabs is given an argument, this argument will be the address of a floating point number which will be loaded into the AC-IO at the beginning of fabs.
f'add (floating add)	calls a subroutine which performs a floating point addition of the AC-IO to the floating point number whose address is the argument of fadd.
* f'com (floating complement)	takes the complement of the floating-point AC-IO leaving the result in the AC-IO. If f'com is given an argument, this argument will be the address of a floating point number which will be loaded into the AC-IO at the beginning of fabs.
* f'co.	is used by fabs and fcom
f'dac (floating deposit)	performs a deposit of the AC into the address specified by the argument and a deposit of the IO into the specified address + 1. This is useful to floating-point deposit the AC-IO.

1-FPT-28 is a floating point package developed by Lt. R. D. Smallwood 7/63
2-FPT-18 is the original AFCRL 18 bit mantissa 18 bit exponent floating point package.

.fdvd (floating divide)	calls a subroutine which performs a floating point division of the AC-IO by the floating point number whose address is the argument of fdvd.
f'lac (floating load)	performs a loading of the AC from the address specified by the argument and a loading of the IO from the specified address + 1. This is useful to floating-point load the AC-IO.
f'lip (floating input)	calls a subroutine which converts a typed in decimal number to a 36 bit (2 word) floating point number returned in the AC-IO. If f'lip is given an argument, there will be a floating point deposit of the AC-IO into the address specified by the argument at the end of f'lip.
f'loat	calls a subroutine which converts the number in the AC to a floating point number in the AC-IO. If float is given an argument, the accumulator is loaded with the contents of the address specified by the argument at the beginning of float.
f'loc (floating location)	leaves 2 zeroed-locations in core for floating point number storage.
.flop (floating output)	calls a subroutine which types out the floating point number in the AC-IO according to a format statement whose address is given in the second argument of the flop macro. If the first argument of the flop macro is non-null, the AC-IO is loaded with the floating point number whose address is given by the first argument at the beginning of flop.
f'mpy (floating multiply)	calls a subroutine which performs a floating point multiplication of the AC-IO by the floating point number whose address is the argument of f'mpy.
f'sub (floating subtract)	calls a subroutine which performs a floating point subtraction of the floating point number, whose address is the argument of f'sub, from the AC-IO.

gdn (get decimal number)	calls a subroutine which converts a typed in decimal number to binary. If gdn is given an argument, the contents of the AC at the end of gdn are stored in the address specified by the argument.
gon (get octal number)	(same as gdn but for octal numbers)
ifix (integer fix)	calls a subroutine which converts the floating point number in the AC-IO to a fixed point result in the AC. If ifix is given an argument, this argument will be the address of a floating point number which will be loaded into the AC-IO at the beginning of ifix.
if.	is used for conditional assembly by fabs, fcom, flip, float, gdn, gon, tdn, ton
swap	exchanges the contents of the AC and IO.
szs (skip on zero switch)	assembles the proper skip command for the specified switch. The indirect bit may be specified for the "skip not" command.
tdn (type decimal number)	calls a subroutine which types the contents of the AC in decimal. If tdn is given an argument, the AC is loaded with the contents of the address specified by the argument at the beginning of tdn.
ton (type octal number)	(same as tdn but for octal numbers)
tpo	calls a subroutine which types character strings to the 13 character. The string must immediately follow the call to tpo.
tst	(same as tpo but the string is remote and its address must be specified in the argument of tst).

```
fabs        macro mantissa, address
            if. flac mantissa
            sma
            jmp address
            fco.
address     bss 0
fabs        end

fadd        xmacro number
            jda fadd
            number
fadd        end

fcom        macro mantissa, address
            if. flac mantissa
            sza'
            jmp address
            fco.
address     bss 0
fcom        end

fco.        macro
            cma
            swap
            xor =0777400
            swap
fco.        end

fdac        macro mantissa
            dac mantissa
            dio mantissa+1
fdac        end

fdvd        xmacro number
            jda fdvd
            number
fdvd        end

flac        macro mantissa
            lac mantissa
            lio mantissa+1
flac        end
```

```
flip        xmacro mantissa
            jsp flip
            if. fdac mantissa
flip        end

float       xmacro number
            if. lac number
            jda float
float       end

floc        macro
            oct 0, 0
floc        end

flop        xmacro mantissa, format
            iff 0, mantissa
            flac mantissa
            jda flop
            format
flop        end

fmpy        xmacro number
            jda fmpy
            number
fmpy        end

fsub        xmacro number
            jda fsub
            number
fsub        end

gdn         xmacro a
            jsp gdn
            if. dac a
gdn         end

gon         xmacro a
            jsp gon
            if. dac a
gon         end

ifix        xmacro mantissa
            if. flac mantissa
            jda ifix
ifix        end
```

```
if.         macro a
            iff 0, Ca
            a
if.         end

swap        macro
            rcr 9
            rcr 9
swap        end

szs         macro switch
            skp switchx@10^70  switch^10000
szs         end

tdn         xmacro number
            if. lac number
            jda tdn
tdn         end

ton         xmacro number
            if. lac number
            jda ton
ton         end

tpo         xmacro
            jsp tpo
tpo         end

tst         xmacro string
            jsp tst
            string
tst         end
```

APPENDIX E

Table of AMP Error Flags

Flag	Meaning
A	address arithmetic error
D	dup or irp error
F	source program format error
I	indirect address specification non-normal
K	the assembly of this line was killed
L	literal error
M	multiple definition of the same tag attempted
N	number error
O	operation code error
P	phase error
Q	qualifier error
R	relocation error
X	extern-intern error