Advanced
Micro
Computers
A subsidiary of
Advanced Micro Devices

AmSYS™29/10 Microprogram Support Software

User's Manual

059910515-001

$15.00

REVISION RECORD	
REVISION	**DESCRIPTION**
01	Prelimiary Issue
(1/30/81)	
02	Manual Revised
(3/13/81)	
A	Manual Released
(6/12/81)	
Publication No. 05991051S-001	

REVISION LETTERS I, O, Q AND X ARE NOT USED

© 1981 Advanced Micro Computers
Printed in U.S.A.

Address comments concerning this manual to:

ADVANCED MICRO COMPUTERS

Publications Department
3340 Scott Boulevard
Santa Clara, CA 95051

PREFACE

This manual provides the user with detailed information about the generation, debug, and emulator support software used wtih the AmSYS29 bit-slice microprocessor development system. The manual is organized according to the usual sequence employed in program development: definition, assembly, emulation support, post-processing, and error analysis.

The information in this publication is believed to be accurate in all respects. However, no responsibility is assumed for errors that might appear in this publication. Advanced Micro Computers disclaims responsibility for any consequences resulting from the use thereof. No part of this manual may be copied or reproduced in any form without prior written permission from AMC.

These products are intended for use only as described in this document. Advanced Micro Computers cannot be resposible for the proper functioning of undescribed features or parameters.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

FIGURES

CHAPTER 1

INTRODUCTION AND PURPOSE

INTRODUCTION

Microprogramming Software programs used with AmSYS29 support the development and debugging of microcode for prototype systems designed around microprogrammed controllers.

It is assumed that the user has some familiarity with the AmSYS29 hardware and operating procedures. For more detailed information concerning System 29 hardware and operation, refer to the AmSYS29 Hardware Manual or related documentation.

This manual describes the Microprogram Support Software which is composed of the Microprogram Generation Software and the Microprogrammed debug software. Microprogram generation software consists of the AMDASM meta-assembler, AMMAP, AMPROM, and AMSCRAM. Microprogrammed debug software consists of four programs to move microcode between disk storage and bipolar memory and debug support that interfaces the CPU with the clock control logic.

AMDASM is a meta-assembler processor used to generate microprogram object code. An assembler reads another program written in a symbolic form and produces an output of binary words corresponding to the symbolic input. A microprogram assembler is a special kind of assembler formally called a meta-assembler.

A meta-assembler differs from an ordinary assembler in that most of the symbols are defined by the user prior to the assembly process. In an ordinary assembler, the user can define labels for instructions and symbols for particular data words, but the instructions themselves, including their associated word length and format, are generally already defined by the assembler. A meta-assembler format rarely establishes the entire contents of a microinstruction, but rather defines only a few bits of the total word.

A meta-assembler must be far more flexible than a traditional assembler, since it is used with many hardware configurations. Each hardware configuration can require a different format and word lengths (microinstructions) over 100 bits.

AMDASM operates in two phases, the definition phase (PHASE1) and the assembly phase (PHASE2). The definition phase establishes word length and definition of formats and constants (the definition file). The assembly phase is traditional assembly process (assembly file) performed on a program using the formats and constants from the definition file. This phase reads a symbolic program, handles most common assembler features such as labeling and setting the address

counter, and produces a binary output and various listings and cross-reference tables. The definition phase is executed first to set up the table which associates the user's format names and constant names with their corresponding bit patterns.

Following assembly of the user's program, a file is retained which contains the assembled microprogram. This file is then prepared for PROM programming and for debugging in the AmSYS29 microprogrammed controller. The output utility can select columns and rows for a given PROM freeing the user from any restrictions regarding the organization of the microprogram memory.

The program to be assembled may be written using any of the features specified during the definition phase. In the simplest case, the assembly phase source program might be written using just strings of ones and zeros, with the definition phase consisting only of the microinstruction word length. At the other extreme, the assembly phase source program may refer to multiple format names from the definition phase for each microinstruction. Any number of formats may be overlaid to define a single microinstruction, as long as the defined or variable fields of each format fall into the don't care fields of the other formats invoked. A user can overlay a format specifying sequence control operations, another for data control, and a third for memory control.

The AMC assembler has been written to maximize its flexibility and ease of use for hardware designers. Every effort has been made to make the program efficient on the machine and efficient at the human interface, with a minimal knowledge of the host machine's operating system required.

NOTE

Throughout this manual examples often refer to the Am2900 Learning and Evaluation Kit illustrated in Chapter 5.

CHARACTER SET

The following characters are legal in AMDASM source statements.

- The letters of the alphabet, A through Z. Both upper and lower case letters are allowed. Internally, AMDASM treats all letters as though they were upper case, but the characters are printed exactly as they were input in the source files.

- The digits 0 through 9.

- The following special characters:

CHARACTER	MEANING
+	Plus Sign
-	Minus Sign
*	Asterisk
/	Slash
,	Comma
(	Left Parenthesis
)	Right Parenthesis
&	Ampersand
:	Colon
$	Dollar Sign
%	Percent Sign
#	Blank or Space
;	Semicolon
.	Period
cr	Carriage Return
HT	Horizontal Tab

DEFINITION OF TERMS

There are no standard terms associated with microprogram assemblers; the more common terms used in this manual are listed in table 1-1.

IMPLEMENTATION

AMDASM operates on the Advanced Micro Computers' AmSYS29 under the AMDOS29 Operation System. Table 1-2 is a summary of AMDASM commands for both the definition and assembly phase. Included are examples or constraints for each command.

ASSEMBLER OPERATION

AMDASM is placed into execution by control statements from the console input device.

The definition file is processed in PHASE 1 and if it contains no errors the assembly phase begins. PHASE 2 Pass 1 assigns values to source file labels and allocates storage. PHASE 2 Pass 2 translates the source file source program into object code.

TABLE 1-1. COMMON TERMS FOR AMDASM

TERM	DEFINITION
ß	Indicates a required blank character.
Name or label	1-8 characters are assigned a value by the programmer or the assembly process. Labels are used only in the assembly file.
Constant	A specific pattern of 1-16 bits.
Constant Name	A name for a constant.
Field	A group of adjacent bits in a microinstruction
Format	A model for a microinstruction consisting of fields containing constants, variables, and don't cares.
Format Name	A name for a format.
Line	An input line of up to 128 characters on a console or a diskette file.
Modifiers	Symbols (*%:-$) which indicate the data given for a field is to be modified.
Attribute	A modifier which is permanently associated with a field.
Designator	A symbol (V, X, B# Q#, or H#) which indicates the type of field or constant: variable (V), don't care (X), binary (B#), octal (Q#), decimal (D#), or hexadecimal (H#).
Delimiters	A symbol (:ß=,/) used to indicate the end of a name (:ß=), the end of a field (,), or the continuation of a statement (/) on another line.
Default Values	The value which will be substituted if an explicit value is not specified.
Options	Choices available which indicate the input and output devices to be used, the type of output listing desired, and processing of one or both phases (definition and assembly).
[]	Brackets indicate that the enclosed parameter is optional.
cr	Carriage Return.

TABLE 1-2. AMDASM COMMAND SUMMARY

COMMAND	DEFINITION PHASE EXAMPLE OR CONSTRAINT
TITLE	Max 60 characters
WORD n	n ≤ 128
EQU␣	Name:EQU␣constant/expression
SUB␣	Name:SUB␣field,... 10 fields max
DEF␣	Name:DEF␣field,... 30 fields max
NOLIST	Do not print following statements
LIST	Print following statements
END	End of definition source file

COMMAND	ASSEMBLY PHASE
TITLE␣	Maximum 60 characters
EQU␣	Name:EQU␣constant/expression
NOLIST	Do not print following statements
LIST	Print following statements
f.n.␣	Format name␣VFS,...(from DEF)
FF␣	Free format FF␣field,...max 30
SPACE␣n	Spaces n blank lines
EJECT	Ejects page
ORG␣n	Resets program counter (forward)
RES␣n	Reserves n words of code
ALIGN␣n	Sets PC to next even multiple of n
LABEL:	Preceded f.n. or FF, value = PC
LABEL::	Entry point for op code mapping memory
;	Comment statement

NOTES

␣ = Required space
Names = 8 characters, no blanks
Char 1 = A-Z, or Char 2-8 = A-Z, 1-9
.... = Optional

User-selected options determine whether the definition phase is to be executed or if a previous execution of that phase has already established the table of formats on a file which will be used by the assembly process.

The AMDOS29 operation system allocates all necessary input and output resources, such as files, automatically. Table 1-3 is the AMDASM microcode object file format. Table 1-4 is a summary of field and operator information.

TABLE 1-3. AMDASM MICROCODE OBJECT FILE FORMAT

BYTE NUMBERS	DESCRIPTION
0-59	Title record (60 bytes)
60	Microword size (i.e., width in bits)
61-62	Maximum location (program) counter value
63-64	Number of microinstructions in file
65	m = Number of 16 bit words required for each microinstruction
* 66-67 ** 68-(68+2m-1) *** (68+2m) - (68+4m-1)	One microinstruction record

* Location (program) counter value.

** Mask defining don't care fields bit = 1- means this is a don't care bit; bit = 0- means this is a defined bit.

*** Contents of microinstruction. If corresponding bit of mask = 0, this bit is a defined value. Don't care bits = 0.

HORIZONTAL TABS

A horizontal tab may be entered for readability as the user inputs his source files. The assembler places the character following the horizontal tab at the next tab position. Tab stops begin with position 1, and occur every eight positions thereafter as follows: position 1, 9, 17, 25, etc. Thus if data is input at character position 5, a tab will place the next character input as position 9. However, if data is input at character position 17, a tab will place the next character at position 25. Horizontal tabs may be used in both the definition and assembly files.

TABLE 1-4. AMDASM 29 FIELD AND OPERATOR INFORMATION

CONSTANTS, EXPRESSIONS, CONSTANT FIELDS

[n] des digits [mod]

VARIABLE FIELDS

n V [attr] [des] [digits] [mod] (digits are default value)
n V [attr] X (defaults to X)
max n = 16

DON'T CARE FIELDS

n V [attr] X max n = word size

MODIFIERS (mod) and ATTRIBUTES (attr)

*	Inversion
-	Negation
%	Right justify or field has expression
:	Truncation
$	Paging (relative addressing) ATTRIBUTE only, sets % and :

EXPRESSION OPERATORS

+	Add	
-	Subtract	Evaluated left to right
*	Multiply	
/	Divide	

DESIGNATORS

B#	Binary
D#	Decimal
Q#	Octal
H#	Hexadecimal

VARIABLE FIELD SUBSTITUTES (VFS)

Label
Label$
Expression
Digits
Des digits [mod]
Constant name

TABLE 1-4. AMDASM 29 FIELD AND OPERATOR INFORMATION (continued)

NOTES

[]	= Optional
Des	= Designator
Attr	= Attribute
Mod	= Modifier
Digits	= Numbers

All values are binary. Bytes 61 and 62 are stored low order byte first, high order byte second (e.g., if the value is 01FF it would be stored as FF,01). This also applies for bytes 63-64, 66-67, the mask and the microinstruction which are stored and written as 8080 addresses (i.e., 2 bytes with low order first).

If the microcode is not continuous (due to the use of ALIGN, ORG or RES), there is no data stored for the empty words of microcode.

CHAPTER 2

DEFINITION PHASE (PHASE 1)

INTRODUCTION

The definition phase allows the user to define the microword length, constants and formats used to write source programs.

The AMDASM definition phase includes the following features:

- A name from 1 to 8 characters assigned to a constant value.
- A name used to define a format whose fields are given as variables, don't cares, explicit bit patterns (values), or specific addresses by using appropriate designators.
- Blanks used to improve readability.
- Microword length from 1 to 128 bits.
- Modifiers that include inversion, truncation, negation, and designation of a field as an address field to be right-justified (placing a value in a field at the right with leading bits set to zero).
- The ability to set a page size via the attribute $. This permits error detection when the assembly phase calls for a jump or branch to an address which is on a different page of the microcode.

Data from the definition phase may be retained for use with subsequent assembly phase source programs and/or it may be modified as desired.

DEFINITION FILE

Definitions are input via a sequence of instructions called the definition file whose content includes the following items:

```
TITLE (heading to be printed on output listing)
WORD n (defines microinstruction word length)

|                                  |
|  Printing control statements     |
|  Definition statements           |
|  Comment statements              |
|                                  |

END
```

The control statement WORD appears as the first statement in the definition file after the optional TITLE statement. The END statement is the last statement in the definition file.

Other statements (shown boxed) may be interspersed throughout the body of the file.

In the definition file, microinstruction length is defined first. The word can be any length from 1 to 128 bits, which is adequate for all but the most sophisticated processors.

Each user defined symbol has a specific bit pattern associated with it. A format name is used to define all or part of one microinstruction. The format definition can consist of numeric fields defined to contain specific bit patterns, variables filled in when the format is invoked, and don't care states.

Once the definition phase has been executed, its output can be retained and used by future programs. The don't care states are retained until defined, which may not happen until after the assembly process, during a third, or post-processing, phase. At the conclusion of assembly a listing of the microprogram shows an X for every undefined bit. This is extremely useful during the development process before the microword length has been optimized by sharing fields.

To facilitate readability, blanks may appear in most parts of these statements, blanks are not permitted between the letters of the control words TITLE, WORD, END, LIST, NOLIST, DEF, EQU, or SUB. An entire blank line may be inserted by entering a semicolon and a carriage return. The definition file statements TITLE, WORD, and END are described in the following paragraphs.

TITLE

To print a title on definition file statements, the first statement input should be TITLE. The format is:

TITLE␣ title desired by user

TITLE must begin on a new line and is followed by a blank and a maximum of 60 characters.

WORD

WORD statement input is used after the optional TITLE. Its format is:

WORD␣n

WORD␣ is followed by a decimal integer value n that indicates the microword size in bits (range 1-128), and by at least one blank and 1 to 3 decimal digits. It is the first input line (second input line if TITLE was used) and must begin on a separate line.

If WORD is omitted, assembly will halt as the definition phase must know the size of the microword in order to proceed.

END

END indicates the end of the definition file. If END is omitted an error message will be printed but processing will continue. The format is:

```
END
```

END must begin on a new line and be the last statement in the definition file. It is always followed by a carriage return.

PRINTING CONTROL STATEMENTS

The description of printing control statements, LIST, NOLIST, EJECT, and SPACE is as follows:

LIST

LIST indicates that the following statements are to be printed whenever printing of the definition file input is requested. This feature is used when correcting or modifying a definition file. AMDASM selects LIST as the default option. NOLIST must be specified if the user does not wish to print his definition file source statements. The format is:

```
LIST
```

LIST begins on a new line and is always followed by a carriage return. It precedes the definition file statements to be printed and is interspersed between complete definition statements.

NOLIST

NOLIST turns printing off, and printing of the definition file input statements will not occur until LIST is encountered. However, any source statement containing an error will be listed. The format is:

```
NOLIST
```

NOLIST begins on a new line and is followed by a carriage return. It precedes the definition file statements, which are not to be listed, and is interspersed between complete source statements.

SPACE

SPACE indicates that the assembler is to leave n blank lines before printing the next source statement. The format is:

```
SPACE␣n
```

SPACE begins on a new line and is followed by ␣ and a decimal digit indicating the number of succeeding lines to be left blank. It is inserted in the definition file at the point where the spaces are desired.

EJECT

An EJECT statement causes the assembler to generate blank lines on a list device so that any previous lines plus the blank lines equals the specified page length (default is 66 lines). It then generates a new page headed with the title. On a printer a new page is ejected. The format of this statement is:

```
EJECT
```

EJECT begins on a new line and is followed by a carriage return.

DEFINITION STATEMENTS

Definition statements are used to define constants, full microword formats, or partial microword formats. The format for these statements is:

```
name: definition word␣ |field1, field2,..., fieldn|
                       |or                       |
                       |constant                 |
```

DEFINITION WORDS

The definition words and their functions are:

EQU is used to set a name equal to a bit pattern.
DEF is used to define a format for a microinstruction.
SUB is used to define a format for part of a microinstruction.

NUMBER OF PERMITTED EQUs, DEFs, AND SUBs

There is no fixed maximum number of EQUs, DEFs, or SUBs because AMDASM stores all data dynamically. The user of a 64K-byte system has available, in PHASE1, approximately 42K bytes for variable storage; PHASE2 has approximately 40K bytes.

PHASE1 allocates:

12 bytes for each EQU
12 bytes for each format or subformat name
4 bytes for each field in a DEF or SUB

PHASE2 allocates:

12 bytes for each format name, constant name and label
4 bytes for each format field

FIELDS

A field is a contiguous group of bits in a microinstruction (such as branch address, next instruction control, etc.) Each field may be one of three types:

- A constant field whose content is a fixed value or a fixed bit pattern, (for example, the next instruction control).
- A variable field whose content will contain different bit patterns in different situations (for example, an address field).
- A don't care field whose content is not used in this format (for example, the address field for a continue instruction).

The type of data in a particular field is indicated by using designators.

DESIGNATORS

Permissible designators and their meanings are shown in table 2-1.

FIELD RULES

Each field following a definition word contains a maximum of 16 bits except the don't care field which contains a bit length and the designator X. The field is followed by a comma unless it is the last or only field following the definition word.

Define a constant field by using the designators B#, Q#, D#, or H# and the appropriate digits. Some fields contain a variable which gives a bit length and the designator V, the field type defaults to binary. Fields may also contain a constant name or subformat name which has been previously defined.

TABLE 2-1. PERMISSIBLE DESIGNATORS

DESIGNATOR	MEANING	EXAMPLE
B#	A constant or field whose contents will be represented using binary digits (0 and 1). Each digit has an implicit length of one bit.	B#101 (three bits 101).
Q#	A constant or field whose contents will be represented using octal digits (0 thru 7). Each digit has an implicit length of three bits.	Q#32 (six bits 011010).
D#	A constant or field whose contents will be represented using decimal digits (0 through 9). For a constant name definition using EQU, the implicit length for decimal numbers is the number of bits needed to represent the number in binary. Thus, D#3 has an implicit length of 2, D#4 has an implicit length of 3. For fields in a format (DEF or SUB), the D# must be preceded by decimal digit(s) giving an explicit length (number of bits) for the field.	D#4 (three bits 100). 3D#6 (three bits 110).
H#	A constant or field whose contents will be represented using hexadecimal digits (0 through 9, A through F). Each digit has an implicit length of four bits.	H#8A (eight bits 10001010).
X	A don't care field. X must be preceded by decimal digit (s) giving an explicit length for this field (i.e., the bit length).	4X (4 bit don't care field).
V	A variable field. V must be preceded by a decimal digit (s) giving an explicit length for this field (i.e., bit length).	6V (six bit variable field).

TABLE 2-1. PERMISSIBLE DESIGNATORS (continued)

DESIGNATOR	MEANING	EXAMPLE
V (continued	When a designator B#, Q#, D#, or H# is given after a V, it becomes a permanent attribute of that field and the assembler assumes that any value specified for that field will be given in digits appropriate to that designator. These permanent designators for variable fields may be overridden when using the format during the assembly phase. If a variable field has no designator given, it defaults to binary. For example, if all variable fields are given as nVQ# in the definition phase, all values for this variable field that are octal may be written during the assembly phase by writing only the necessary octal digits. The content of a variable field may be given during the definition phase. The V designator may be followed by the B#, Q#, D#, or #H and these may be followed by appropriate digits called the default value for this field. Thus 6VQ# indicates a 6-bit variable field whose contents will be given in octal. 6VQ#35 indicates that if no value is substituted in the Assembly Phase, this variable field should assume the default value 011101.	

NOTE

The designators B#, Q#, D#, and #H must not have blanks between the letter and the #. The desired value for the field is then given in the appropriate digits as shown in the examples.

NAMES

Names may be user-defined constants, formats, or subformats.

They are the first element in a statement. Each name begins with an alphabetic character (A-Z) or a period (.) and is terminated by a colon (:). A maximum of eight characters not including the colon is permitted. Embedded blanks are not permitted. Names are followed by EQU, DEF, or SUB. Positions 2 through 8 contain only alphabetic characters (A-Z), a period (.), or the digits (0 through 9).

Names that contain more than eight characters will be truncated after the first eight characters. A name may be preceded by blanks or followed by blanks after the colon and before the EQU, SUB or DEF.

Examples of proper names are:

```
NUMBER:
.SHIFT:
REG.3:
```

Improper names are:

```
*ADD          (special character used)
SHIFT LEFT:   (embedded blank, more than eight characters)
3MUXCNTL:     (first character not A through Z or period)
```

CONSTANTS

Constants are used to associate a name with a value or to define a specified fixed bit pattern. They are expressed by using designators and the appropriate digits as follows:

Q#62 This defines the bit pattern 110010 and has an implicit bit length of 6 bits (each octal digit represents 3 bits).

4H#5 A decimal digit precedes the designator the 4 represents the explicit length of the field, and the bit pattern is 1010.

Explicit and implicit lengths are more fully defined later in this chapter. Constants are left justified.

Constants must be represented in 16 bits (i.e., $2^{16}-1$ maximum). The permissible forms for constants are shown in table 2-2.

TABLE 2-2. CONSTANT FORMS

FORM		PERMISSIBLE DIGITS	MEANING
	n	0 through 9	Decimal value (default form)
i	B#n	0 or 1	Binary value
i	Q#n	0 through 7	Octal value
i	D#n	0 through 9	Decimal value
i	H#n	0 through 9 or A through F	Hexadecimal value
where: i represents optional digits specifying the explicit length.			

EXPRESSIONS

When a field contains an expression, the expression may use designators labels, operators and/or digits.

Operators permitted in expressions are:

OPERATOR	DESCRIPTION
+	Add the value of the left operand to the value of the operand on the right of +.
-	Subtract the value of the operand to the right of the minus (-) from the value of the operand on the left.
*	Multiply the left operand by the right operand.
/	Divide the operand on the left (dividend) by the operand on the right (divisor).

All expressions are evaluated from left to right. There is no hierarchy for the operators and no parentheses for nesting are permitted. The result is a value that is a positive constant which is calculated using integers; remainders are discarded.

DEFINITION WORDS

The definition words EQU, DEF, and SUB are described as follows:

EQU

EQU is used to equate a constant name to a constant value or expression. The format is:

name:EQU␣ constant (or expression)

This equates the characters given in the name position to the value of the constant or expression. Only one expression or constant is permitted following the EQU.

The following sets the name R12 equal to the bit pattern 1100:

R12:EQU␣H#C

Future references to the bit pattern 1100 (register 12) may be made by using the name R12.

The default type is decimal if no designator follows the EQU. (R10:EQU␣10 assumes the bit pattern 1010, implicit length 4 bits).

Each EQU statement begins on a new line with a name. The name and colon (:) must be followed by EQU␣ (blanks between the colon and EQU are optional). It contains a constant, an expression or a constant name which represents a bit pattern. The statement defines a value that can be represented in 16 bits (2^{16}-1 maximum).

Each EQU may be followed by a semicolon and comment after the constant or expression. To continue on additional lines use a / (slash) as the first nonblank character in those lines. The statement is used in the assembly file as well as in the definition file.

DEF

DEF is used to define a complete microword format by establishing the contents of unvarying portions of the microword, the position and length of variable and don't care fields. In addition, default values for variable portions of the word may be specified. The format is:

name: DEF␣field1, field2,...,fieldn

Each DEF begins on a new line, preceded by a name. It is followed by one or more blanks, then fields separated by commas. The sum of the lengths of all fields must equal the microword length specified by WORD. Every bit in the microword is specified in terms of constants, don't cares, or variables.

A DEF may contain blanks between name, the colon, and DEF␣. To continue the statement on additional lines use a / (slash) as the first nonblank character in those lines. A semicolon and a comment may follow after any full field is defined. A subformat name or constant name which has been PREVIOUSLY defined, don't care, constant or expression may be part of any field.

A DEF may contain a variable field which specifies a default value for the field. The statement may be overlaid on don't care fields with another format to obtain a complete microword during the assembly phase. Overlaying on other than don't care fields will result in errors, so this feature must be used with care.

SUB

SUB is used to define a subformat of the microword. A subformat is the same as a format except that it contains fewer bits than the full microword. The fields may be constants, variables or don't cares. Its format is:

name: SUB␣ field1, field2, ..., fieldn

Each SUB is preceded by a name: (colon) and followed by one or more blanks, then fields separated by commas. It precedes the DEF in which it is first referenced, begins on a new line and can not be used in the Assembly File.

A SUB may be less than a microword in length bits. To continue on additional lines use / (slash) as the first nonblank character in those lines. It can be followed by a semicolon and a comment after any complete field or contain (for any field) a constant name that was previously defined, or a constant, expression, variable, or don't care specification.

A SUB will be useful when several formats contain identical adjacent fields. In this case, the subformat name may be used in each DEF where these fields occur.

EXAMPLES OF EQU, SUB, DEF (H1)

An EQU is used to associate a bit pattern with a symbol (constant name). An example of this and other applications are as follows:

R2:EQU␣B#010

This defines the name R2 as a 3 bit constant with the bit pattern 010. Whenever the symbol R2 is used, the bit pattern 010 will be substituted.

SHFTRT:SUB⏨3V,B#10110,5X	This defines SHFTRT as a subformat with a 3 bit variable field (3V), a 5 bit constant field (B#10110), and a 5 bit don't care field (5X) for a total of 13 bits.

A DEF is used to associate bit patterns with a symbol (format name). Examples of this and other applications are:

ADD:DEF⏨ 3V,B#10110,5X,B#0011,4X,B#010	This defines ADD as a format with a 3 bit variable field (3V), a 5 bit constant field (B#10110), 5 bit don't care field (5X), a 4 bit constant field (B#0011), a 4 bit don't care field (4X), and a 3 bit constant field (B#101). This gives a total microword length of 24 bits.
ADD:DEF⏨SHFTRT,B#0011,4X,R2	Alternatively, the same format name could be written using the subformat name (SHFTRT) and the constant name (R2) previously defined.
TWOK:EQU⏨2048	This assigns the bit pattern 1000000000000 and a length of 12 bits to the name TWOK. The 2048 is assumed to be decimal and the length is taken from the rightmost bit through the leftmost bit in which a 1 appears.
EIGHT:EQU⏨8	This yields the bit pattern 1000 with a length of 4.

Alternatively, by using different designators, the constant TWOK:EQU⏨2048 could be written:

```
TWOK:EQU⏨B#100000000000
TWOK:EQU⏨B#4000
TWOK:EQU⏨H#800
```

All of these yield the bit pattern 100000000000 and a length of 12.

FIELD LENGTHS

Each field may be given an explicit or implicit length. An explicit length is indicated for a field by using decimal digit(s) before the designator. The maximum length is 16 bits except for don't care fields whose maximum length is the microword size. The expression 3B#101 indicates a field with an explicit length of 3 bits.

Decimal, variable, or don't care designators require an explicit length before the designator D#, V, or X. Table 2-3 shows examples of field length definition. Decimal fields in a format or subformat require an explicit length since there is no direct correlation between the number of decimal digits given and the number of binary bits desired for this field.

TABLE 2-3. FIELD LENGTH DEFINITION

EXAMPLE	DESCRIPTION
4V	Defines a variable field with the explicit length of 4 bits.
5D#16	Defines a constant field with the explicit length of 5 bits and the bit pattern 10000.
R3:EQU#5	Defines a constant using the default type decimal, value 5. The implicit bit length is 3.

CONSTANT LENGTHS

A constant may have an implicit or an explicit length. An explicit length is given by placing the bit length (in decimal digits) before the designator. An example of this is B:EQU#4D#8 which has an explicit length of 4 and the bit pattern 1000.

If an explicit length is not given, the constant is assigned an implicit length determined by the designator used. Table 2-4 shows some examples of the attributes of implicit length constants.

CONTINUATION

Any statement may be continued on additional lines by placing a / (slash) as the first nonblank character in those lines. Continuation should be indicated after a complete field (including comma) has been

given on the preceding line, but not between the designators B, D, Q, or H, and the # sign.

Examples are:

```
SHFTRT:SUB␣3V,B#10110,
/5X
ADDldef33V,B#10110,5X,
/B#0011,4X,B#010
```

TABLE 2-4. IMPLICIT LENGTH ATTRIBUTES OF CONSTANTS

CONSTANT	IMPLICIT LENGTH	BINARY VALUE	DESCRIPTION
AB:EQU␣B#1000	4	1000	Each binary digit yields an implicit length of 1 bit per digit.
BB:EQU␣Q#10	6	001 000	Each octal digit yields an implicit length of 3 bits per digit.
CB:EQU␣H#10	8	0001 0000	Each hexadecimal digit yields an implicit length of 4 bits per digit.
DB:EQU␣12	4	1100	The 12 is assumed to be decimal, and the implicit length is counted from the rightmost bit through the leftmost 1.
EB:EQU␣4	3	100	Same as above. Implicit length 3.

COMMENT STATEMENTS

A comment statement is used to provide information about program variables or program flow. It may be a full or a partial line. The format is:

```
;comment text
```

Comments begin with a semicolon and are placed after a completed field if used within a DEF or SUB. Subsequent fields for that DEF or SUB must begin on a new line with a / (slash) indicating that they are a continuation.

Examples of comment statements are as follows:

```
1.  SHFTRT:SUBØ3V,; this is a shift right subformat.
2.  /B#10110,5X; which is continued on a second line.
3.  ; the ADD given below is a complete microword format.
4.  ADD:DEFØSHFTRT,B#0011,4X,R2.
5.  ; total number of bits for SHFTRT is 13.
6.  ; the bit pattern for SHFTRT will be substituted.
7.  ; in the ADD given above.
```

Statements 3, 5, 6, and 7 are full comment lines. Statements 1 and 2 are statements to be processed but all characters after the 'semicolon' will be treated as comments. The SUB begun in statement 1 is continued in statement 2 where / indicates continuation.

MODIFIERS AND ATTRIBUTES

Modifiers are placed after a constant or after the designator V. When placed after a constant, they alter only the value given. When used after a V, the modifiers are called attributes of that field and are permanently associated with the field. Attributes will modify any default value given with the variable field in the definition file and they will modify any value substituted for this variable field when the format name is used in the assembly file.

Table 2-5 provides a list of permitted modifiers and their actions.

Table 2-6 provides examples of correct use of modifiers with constants.

Table 2-7 provides examples of incorrect fields due to omission of modifiers.

Modifiers must appear after the value of a constant (i.e., 12H#4C% or 5Q#37:), appear after the V but before the (optional) default value for a variable field (12V%Q#46), if they are to be permanent attributes of the field. The % and the Q# become permanent attributes of this variable field and are also modifiers of the default value. To modify only the default value, modifiers must follow the value (12VQ#46%). Modifiers do not appear with don't cares (e.g., 3X% is illegal). The modifiers (*) and (-) may not both be used for the same field.

A more detailed description and examples are given in Chapter 3.

TABLE 2-5. PERMITTED MODIFIERS

MODIFIER	ACTION PERFORMED ON CONSTANTS OR DEFAULT VALUES
*	Inversion (one's complement).
-	Negate the number (two's complement).
:	Truncate on the left to make the value given fit into the number of explicit bits for this field.
%	This field is to be considered an address field. Any value given is to be right-justified in the field and any bits remaining on the left are to be filled with zeros.
$	The field is treated as an address within a paged memory organization. This attribute permits substitution in this regard and initiates out-of-bounds page checking logic. Used only with variable fields as an attribute (may not follow a default value).

TABLE 2-6. CORRECT MODIFIER USE

EXAMPLE	DESCRIPTION
D#5*	Yields bit pattern 010 (101 (5) is inverted).
B#0101-	Yields bit pattern 1011 (0101 is two's complemented).
6Q#357:	Yields bit pattern 101 111 (the left bits 011 (3) are truncated).
12H#A5%	Yields bit pattern 0000 1010 0101 (the A5 is right justified in a 12 bit field).

TABLE 2-7. INCORRECT FIELDS

EXAMPLE	DESCRIPTION
4B#101	Explicit length is 4 bits, only 3 bits follow the B# but no % sign (indicating right justification) is given.
5Q#34	Explicit length is 5 bits but the 34 generates 6 bits and no : has been given to indicate that the leftmost bit is to be truncated.

MODIFIER PRECEDENCE

Modifiers or attributes may appear in any order but will always be processed in the order shown in table 2-8.

TABLE 2-8. MODIFIER PRECEDENCE

MODIFIER	DESCRIPTION
* or -	Inversion or negation.
%	Right Justification.
:	Truncation.
$	Paged addressing.

DESIGNATORS AS ATTRIBUTES

Variable fields may use the B#, Q#, D#, and H# as attributes and once given they are permanently associated with that variable field unless overridden. If a variable field has no radix base specified it will default to binary.

If the user wants to always input assembly variables in octal, each variable field in the definition phase should be written as nVQ#. Then in the assembly phase the value for this field may be given as 27 and the program will assume that these are octal digits. If octal is not desired in the assembly file, the field in the assembly file program could be written as B#010111, H#27, etc., to override the octal attribute.

If a variable field is defined with a default value (4VH#C) the designator (H#) becomes an attribute of that field.

The attribute H#, if given with a variable field in the Definition File, may need to be repeated in the Assembly File. This is necessary since the program can not distinguish hexadecimal values which begin with A through F from names, which may also begin with the letters A through F.

ATTRIBUTES

The $ attribute may be used only with variable fields to indicate paged addressing. When the $ is given with a variable field, the % and : attributes are automatically set for that field. The $ will indicate that this is a field whose remaining upper (leftmost) bits are to be

truncated and compared with the corresponding bits of the current program counter. If the truncated bits do not agree with the corresponding bits of the PC, an error occurs.

The desired length of the page is determined by the number of bits given as the width of this variable field. Thus, if a page is to be 256 words deep, the variable field would be defined as 8V$. Any value substituted for this field will be truncated on the left and the remaining eight right-hand bits will be substituted into the field. If the truncated left bits do not agree with the corresponding bits of the current program counter value, the substitution would attempt to produce a jump to another page; thus an error message is generated.

DON'T CARES

A don't care is used to indicate the bits (a field) whose state (bit pattern) is irrelevant in this microword instruction.

The form is:

```
nX
```

where

n is the number of bits (in decimal), and X indicates don't care.

Don't cares are printed as an X in the assembly phase output and may be assigned the value 0 or 1 during the post processing phase. Don't cares are the only fields that may be greater than 16 bits in length or used in a format that is overlayed (or'ed) with another format containing a constant in the same field.

VARIABLES

Variables are used to define microword fields whose contents need not be assigned until assembly time. A variable field may be assigned a default value in the definition file. The formats are:

```
nV
nV  attributes
nV  attributes default-value
nV  attributes default-value modifiers
nV  default-value modifiers
```

A variable field is preceded by an explicit length (n) which gives (in decimal) the bit length of the field ($n \leq 16$), contains a V after the length, and ends with a comma (,) if another field follows it.
A % is used after the V if an expression or the program counter is to be used as a substitute for this field in the Assembly File.

A variable field may contain attributes immediately after the V such as inversion (*) that inverts any value given for this field. The field contains a designator given with or without a default value that automatically determines the default type for this field. A default value is given in binary indicated by (B#), octal (Q#), hexadecimal (H#), or decimal (D#) followed by the desired digits. Modifiers appearing after the default value modify only the default value and are not permanently associated with this variable field. Default values given as X indicating don't care are used to overlay this field during the assembly phase. Either a default value of don't care or an explicit default value (bit pattern) may be used, but not both.

Examples of the correct use of variable fields with a default value of don't care are:

```
3VX
3V*X
3V-%X
3V*:X
```

EXAMPLES OF VARIABLE FIELDS

Table 2-9 describes variable field contents. It shows example field contents and discusses their meanings.

To summarize, attributes placed immediately after the V are permanently attached to this field and will operate on any default value given with the field as well as any value substituted for the field in the assembly file. Modifiers placed after a default apply only to the default value. Examples of incorrect variable fields are:

INCORRECT FIELD CONTENT	DESCRIPTION
3VH#&	The H#7 yields 4 bits. No : was given to indicate that the left bit should be truncated to fit the 3-bit field.
3:VH#7	The : is in an incorrect position. It should be 3V:H#7 or 3VH#7: (depending on whether the truncation is a permanent field attribute or a modifier of the default value H#7).

DEFINITION FILE RESERVED WORDS

The following words are used during the assembly phase as assembler control statements and may not be used as format names or constant names in the definition file:

```
ALIGN     EQU       NOLIST    SPACE
EJECT     FF        ORG       TITLE
END       LIST      RES
```

TABLE 2-9. VARIABLE FIELD CONTENTS

FIELD CONTENT	MEANING
3V	A 3-bit field. The content is variable and will be supplied when this format name is used in the assembly file. The field type defaults to binary.
3VQ#	A 3-bit field whose content is variable. The content will be supplied when the format name is used during the assembly file. The content may then be given as one octal digit without using the designator Q#. If the content is to be given in binary, decimal, etc., then the designator B# or D# would be placed before the digit(s) given in the assembly file.
3V*%	A 3-bit field whose content is variable. Any value given for this field within the assembly file will automatically be inverted and right-justified. Since no designator is given, the field defaults to binary. If the content is to be given in octal, etc. in the Assembly File, the appropriate designator (Q#, H#, D#) must precede the digit(s).
3VQ#5	A 3-bit field whose content is variable. If no value is specified for this field in the assembly file, it will assume the default value (specified as Q#5), bit pattern 101.
3VQ#5*	Same as above but the 5 is inverted to yield the bit pattern 010. Values substituted for this field during the assembly file are not automatically inverted.
3V*Q#5	Yields the same pattern as 3VQ#5* but, in addition, any value substituted during the assembly file for this field will also be automatically inverted since the * follows the V rather than the 5.
3V*Q#5*	Yields a 3-bit variable field with a default value of 5, inverted, then inverted again by the * following the V. The resulting bit pattern is 101. Any value substituted for this field in the assembly file will be inverted.

CHAPTER 3

ASSEMBLY PHASE (PHASE2)

INTRODUCTION

The assembly phase provides for input of the microprogram source statements, conversion of format and constant names to their appropriate bit patterns, substitution of values for variable fields in the format, and generation of listing and binary output. The assembly source program references format names and constant names from the definition file. It also contains statements that associate labels with addresses, control assembler operation, and provide program counter control.

The assembly process provides the user with the following features:

- A microword assembled by referring to one or more format names from the definition file.
- A microword whose format was not specified in the definition file specified by using the built-in free-form format command.
- Programmer control of the program location counter to set the origin and/or to reserve storage.
- One of four different output listing formats.
- A constant or variable field defined by using values and/or expressions.
- Errors detected and listed. Severe errors cause processing to halt.

Output of the assembly phase is an object file which contains the complete microprogram. Post processors can directly convert this object file to other forms, such as hexadecimal or BNPF.

The user must input source program statements in an order corresponding to the desired order of executable statements. Blocks of storage may be allocated, a list of print control statements obtained and the program counter set via nonexecutable assembler control instructions that are interspersed with, and do not affect the order of, executable statements.

The source code is input via a sequence of instructions called the source file whose content includes the following:

```
TITLE (heading to be printed on the output listing)

  Printing control words
  Program counter control words
  Constant definition word
  Executable statements
  Comments

END
```

The optional TITLE statement is usually input first so that the desired title appears on the first output page.

The other statements (shown boxed) may be interspersed throughout the body of the file. However, the executable statements must be input in the order that corresponds to the desired sequence of the object (micro) code.

The END statement must be the last statement in the source file. Table 3-1 is a list of assembly phase statements and their categories.

None of the control words (LIST, ORG, etc.) or format names may contain blanks.

TABLE 3-1. ASSEMBLY PHRASE STATEMENTS

STATEMENT	CATEGORY
TITLE LIST NOLIST SPACE EJECT	Printing control words
ORG RES ALIGN	Program counter control words
EQU	Constant definition word
FF	Free-form definition word to establish a microword content
COMMENTS	Used for documentation and program flow
END	End of assembly file

ASSEMBLY FILE STATEMENTS

Each statement contains an optional label followed by a statement type. Some statement types must be followed by an argument which may be a constant name or an expression.

The format of all assembly file statements except comments is:

```
label  |    control word   |
 or    |        or         |
name:  |    format name    |  ␣arguments
       |        or         |
       | definition word   |
```

CONTINUATION

Any statement may be continued on additional lines by placing a / as the first nonblank character in those lines.

LABELS OR NAMES

Labels or names are packed groups of letters and/or symbols which have an associated value.

Labels are permissible with executable statements. Names are required with the definition word EQU. The format for labels or names is:

```
Name:   definition word
          or
label:  format name
```

A name or label's value is determined by the statement type which follows it. Thus, name: EQU␣ equates the symbol name with the value given for n, while label: format name ␣ VFS, VFS... equates label to the current value of the program counter, so that reference may be made to this location in the microcode by using this label.

A label or name begins with an alphabetic character (A through Z) or period (.) and ends with a colon. It contains no more than 8 characters without embedded blanks and, exclusive of the colon, excess characters are truncated on the right. Each label is unique. If duplicates occur, the value at the first occurrence is used and a warning message is issued for each duplicate.

A label or name may precede an EQU, RES, ORG, FF, or an executable instruction. It is used as a variable field substitute (VFS) or as a field in an FF statement but not a reserved word. Labels contain only the letters A-Z, numerals 0-9 or a period in positions 2 through 8.

When a name is defined by an EQU, the definition (source statement) must precede the use of the name as a field or a constant. If the statement AM2909:DEF␣JSR,28X is given, it must be physically located in the source program after the statement JSR:EQU␣H.

A good general rule is to place all EQUs at the beginning of the source file program.

ENTRY POINT SYMBOLS

When a label is followed by a double colon (::) it is called an entry point. Entry points are used when generating mapping PROMs to easily obtain the program counter value associated with certain points in the microcode.

Entry points are indicated in the assembly source file as follows:

```
label:: format name ␣ VFS,...
```

Except for the double colon, entry points are subject to all the rules applicable to labels.

A list of the entry points (symbols and values) may be obtained when AMDASM is executed by requesting the MAP option.

STATEMENT TYPES

The assembly file uses six general types of statements. These are listed below with their permissible control words:

- Printing control statements (LIST, NOLIST, SPACE, EJECT, TITLE)
- Program counter control statements (RES, ORG ALIGN)
- Constant definition statement (EQU)
- Executable instruction statements (format names from the definition phase, FF)
- Comment statements (;)
- END statement

PRINTING CONTROL STATEMENTS

Printing statements for the assembly phase are TITLE, LIST, NOLIST, SPACE and EJECT. They are described as follows:

TITLE

All data input on the line with TITLE will br printed at the top of each page of output. A maximum of 60 characters may be input for a title. When a new TITLE␣ is encountered the list device ejects

blank lines to complete the present page and succeeding pages will contain this title. A page is not necessarily a physical page since the user may specify the length (number of lines) of a page. The format is:

TITLE ␣ alphanumeric data to be printed at the top of the page

LIST

LIST indicates that the following statements are to be printed whenever printing of the source file input is requested. This feature will be most useful when correcting or modifying an Source File. (AMDASM automatically prints the source statements unless NOLIST is specified by the user.) The format is:

LIST

LIST must begin on a new line, be followed by a carriage return, precede the source file statements which are to be printed, and be interspersed between complete assembly statements.

NOLIST

NOLIST turns off printing source statements. Printing of source file input will be suppressed until LIST is again encountered. Any source statement containing an error will be printed. The format is:

NOLIST

NOLIST must begin on a new line, be followed by a carriage return, precede the source file statements which are not to be listed, and be interspersed between complete assembly statements.

SPACE

SPACE indicates that the assembler is to leave n blank lines before printing the next source statement. The format is:

SPACE␣ n

SPACE must begin on a new line, be followed by a ␣ and a decimal indicating the number of succeeding lines to be left blank and be inserted in the source file at the point where the spaces are desired.

EJECT

When EJECT is encountered, the assembler generates blank lines on a list device so that any previous lines plus the blank lines equals the specified page length (default is 66 lines). It begins a new page, headed with the title. On a printer a new page is ejected. The format is:

EJECT

EJECT begins on a new line and is followed by a carriage return.

PROGRAM COUNTER CONTROL STATEMENTS

Program counter control statements for the assembly phase include ORG, RES and ALIGN. They are described as follows:

ORG

ORG is used to set a new program counter (PC) origin. The next assembled microword will be located at the nex origin. The format is:

ORG ␣ n

ORG must be followed by at least one blank and n that is specified using decimal digits, unless one of the designators B#, Q# or H# precedes the digits given. The statement is used only for setting the program counter forward to a value greater than or equal to the current value of the program counter.

ORG may contain an expression instead of n and be used an unlimited number of times in the source file.

If no ORG is specified, the assembly uses an initial PC of 0.

RES

RES is used to reserve n words of memory. This increments the program counter by n. The reserved words will automatically be filled with don't cares by the assembler. The format is:

RES␣ n

RES must be followed by at least one blank and n that is specified using decimal digits, unless one of the designators B#, Q#, or H# precedes the digits given.

RES may contain an expression instead of n and be used an unlimited number of times in the source file.

ALIGN

ALIGN is used to set the program counter to the next value which is an integral multiple of the value of n. It is used to align the program counter to a specific boundary such as that the next microinstruction will be assembled at an address which is the next integral multiple of 2, 4, 8 or 16. The format is:

ALIGN␣n

ALIGN must be followed by at least one blank and n that is specified using decimal digits, unless one of the designators B#, Q#, H# precedes the digits given.

ALIGN may contain an expression instead of n and be used an unlimited number of times in the source file.

CONSTANT DEFINITION STATEMENT

The constant definition statement for the assembly phase is EQU. It is described as follows:

EQU

EQU is used to equate a constant name to a constant value or expression. The format is:

name: EQU␣ constant (or expression)

This equates the characters given in the name position to the value of the constant or expression. Only one expression or constant is permitted following the EQU.

Each EQU must begin on a new line with a new name followed by EQU␣ (blanks between (:) and EQU are optional). A constant or expression represents the bit pattern for one field which must define a value that can be represented in 16 bits (2^{16} -1 maximum).

Each EQU may be followed by a semicolon and comment after the constant or expression. To continue on additional lines use a / (slash) as the first non-blank characters in these lines. An EQU may be used in the source file even if defined in the Definition File. This statement can be equated to the current value of the program counter by using $ as the designator. The $ may be part of an expression.

Examples of EQUs are:

ADD:EQU␣Q#0 defines a 3-bit field whose bit pattern is 000.

This could be an ALU function of ADD for the Learning Kit.

PUSH:EQU␣H#9 defines a 4-bit field, bit pattern 1001 which might represent the next microinstruction control field in the Learning Kit.

EXECUTABLE STATEMENTS

Executable statements form the body of the assembly phase program. When assembled (with appropriate substitution of parameters) they form the binary output code of the assembly phase. They must be input in an order which corresponds to the desired order of the object code.

EXECUTABLE STATEMENTS USING FORMAT NAMES

Most executable instructions will refer to the format names established by the definition phase. The format is:

```
{label:}format name¢VFS,VFS
   (VFS = Variable Field substitution)
```

These formats may be referenced singly (with appropriate VFSs) or they may be combined (overlaid) with other formats (and their appropriate VFSs). All cases result in the formation of a single, complete microword.

Executable instruction statements must begin on a new line and contain a format name from the definition phase. A constant name, a label, a constant, or an expression is substituted for each variable field separated by commas. If a default value was given in the definition phase and is to be used, the VFS may be omitted.

Executable instructions may contain a single format name or an unlimited number of format names to be overlayed. The current value of the program counter may be used as the value for a field if $ is the VFS used for that field. The $ can be part of an expression ($ + n) given for a VFS and be preceded by a label: or a label::.

FREE FORMAT STATEMENT (FF)

Executable statements whose instruction formats were not defined in the definition phase may be defined in the assembly phase by using the built-in free format command, FF. The format is:

```
[label:] FF¢field 1, field2, ..., fieldn
```

An assembly file may contain an unlimited number of FFs. The statements begin on a new line with fields separated by commas. A / is used as the first nonblank character if the statement is to be continued on another line. An explicit length n is given for don't care fields (nX) or for fields defined using decimal (nD#m). FF's do not contain a variable field or a constant name for a field unless that constant has been previously defined in the Source or Definition File. These statements can not be overlayed with another format name.

An FF may be preceded by a label (:) or label (::) and contain an expression for any field, but the expression must be enclosed in parenthesis and be preceded by the field length n. An example of this is FF¢5X,10($-5),B#101. It may also contain a value for an expression which is to be automatically right justified in a field. If the number of bits representing the value is larger than the field length, an error is generated unless the truncation follows the right parenthesis [)] for this expression. A field whose value is the current value of the program counter can be utilized by using $ or an expression containing $ for that field.

If the constants (WORD␣ 48 AZ:), (EQU␣B#01 RB:), or (EQU␣#10) were defined in the definition file, then the source file could contain the following statements:

```
C: EQU␣H#C
XTRA:ff␣ 12H#3%,AZ,18X,C,B#10111,
/1X, RB
```

The microinstruction (binary output) for this FF is shown in figure 3-1. The microinstruction will be printed in the following format:

```
000000000011101XX XXXXXXXXXXXXXXXX 110010111X001000
```

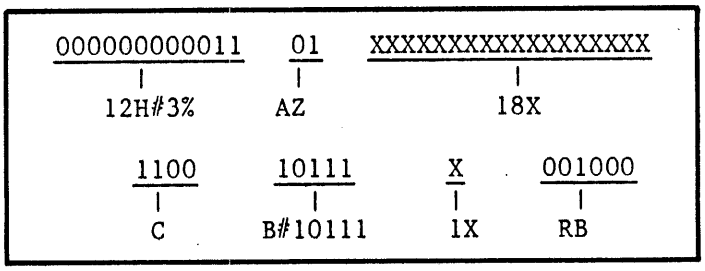

Figure 3-1. Microinstruction Free Form Statement

OVERLAYING FORMATS

The format for overlaying to form a microword is:

```
[label:]format name␣VFS,VFS, &format name␣VFS,VFS..
   (VFS = Variable Field Substitution) (& = overlay)
```

Formats are overlayed with other formats if each bit of format name (#2) contains a one or zero, and that bit is specified as a don't care in the format name (#1) to be overlayed. Subsequent overlays must be on the don't care fields remaining after the overlay of all preceding formats. Each format is a full microword in length. Microword instructions defined using the built-in free format may not be overlayed.

If the definition file contains the following:

```
ADD:  DEF␣ 5X, 8H#A2, 3X
REG1:  DEF␣ B#00001, 11X
CARRY:  DEF␣ 15X,  B#1
```

Then in the assembly phase:

```
ADRGCY:  ADD   &  REG1     &  CARRY
```

yields: 00001 10100010 XX1

COMMENT STATEMENTS

Comment statements are nonexecutable statements which are used to provide information about the program variables or the program flow. A comment may be a full line or may follow a constant definition statement. All characters from the semicolon to the end of the input line are not processed and serve merely as a documentation aid. The format is:

```
;comment text desired
```

END

END indicates that the source file is complete and should be processed. The format is:

```
END
```

END must begin on a new line, be the last statement in the source file and be followed by a carriage return.

ARGUMENTS

An argument follows some types of statements as shown in the executable instruction section.

Permissible arguments are:

- Constants
- Expressions
- Constant names
- Labels

The statements LIST, NOLIST, END, EJECT require no arguments.

Executable instructions which contain format names from the definition file need arguments only if there were no default values given for variable fields. Arguments which are to be substituted in variable fields are called variable field substitutes. All other statement types require arguments.

CONSTANTS

Constants are used as arguments for the commands EQU, ALIGN, RES, SPACE, ORG or as variable field substitutes.

Note that in the assembly file the $ is used to indicate the substitution of the program counter value for the content of a constant or field. Table 3-2 lists the designators that may be used to define constants.

TABLE 3-2. DESIGNATORS USED TO DEFINE CONSTANTS

DESIGNATOR	MEANING
B#	A constant or field whose content will be represented using binary digits (0 and 1).
Q#	A constant or field whose content will be represented using octal digits (0 through 7).
D#	A constant or field whose content will be represented using decimal digits (0 through 9). A D# must be preceded by decimal digit(s) giving an explicit length (number of bits) when representing a field in an FF statement.
H#	A constant or field whose content will be represented using hexadecimal digits (0 through 9, A through F).
$	Use the current program counter as the value for this field or constant.

CONSTANT LENGTHS

Constant lengths are discussed in detail in Chapter 4. However, the length associated with the use of the $ is a special case. When the $ is detected in the evaluation of a constant field or expression, the current program counter value is substituted in place of the $. If the PC = 59 at the instruction preceding NEXTLOC: EQU$+5 then NEXTLOC is equated to 64. If the $ is substituted for a field, the length of the PC is calculated by counting the bits from right to the leftmost significant one bit. The PC length most probably will not agree with the defined field length. When defining fields in a format in the definition phase or in an FF statement, the fields that are to have $ substituted in them should include the percent sign (%) and/or the colon (:) attributes. For example, the field definition 4% will permit any PC value to be substituted into it, but 4V will accept only PC values between 0000_2 and 1111_2.

CONSTANT MODIFIERS

Constants may have modifiers following their given value. They must appear after the constant digits where they may be in any order but will be processed in the order defined in table 3-3. A constant may not be modified by both inversion and negation.

TABLE 3-3. CONSTANT MODIFIERS

MODIFIER	DESCRIPTION
* or -	Inversion or negation
%	Right justification
:	Left truncation
$	Paging

If a constant, including modifiers, is given as a VFS, any attributes (permanent modifiers) given for that field in the definition file will also modify the value of the constant given.

If the definition file contains the expression shown in figure 3-2 and the source file is written TEST: A¥011,9. The binary value 011 is inverted and substituted for field #1, while the 9 (hex) is equated to binary 1001 and right justified for field #2. This results in the microinstruction XXXXX 100 XX 01001 10101. If the source file statement is written TEST2: A¥001*,3* the binary value 001 is inverted by the current *, then inverted again by the attribute in the definition file for field #1. Field #2 hex 3 (binary 0011) is inverted to 1100 and right justified.

The complete microinstruction is as follows:

XXXXX 001 XX 01100 10101

```
A:  DEF¥  5X, 3V*,2X,5V%H#,B#10101
              |       |
          field#1 field#2
```

Figure 3-2. Definition File Expression Example

EXPRESSIONS

Expressions may be used when the programmer wishes to have a value calculated as an argument or as a field substitution. An expression assumes the format:

Symbol operator symbol operator ...

All expressions are evaluated using integer arithmetic and remainders are discarded. Expressions must result in a positive value which can be represented in 16 bits (2^{16} -1 maximum). Only the operators, +, -, *, / are permitted.

The rules for expression usage are:

- Expressions are evaluated in strict left to right sequence. There is no hierarchy for the operators and no parentheses is for nesting are permitted.

- They may contain the $ as a symbol to indicate that the current value of the program counter is to be substituted.

- They are terminated by a comma or the end of the line except when used as a field in FF where they are enclosed by parenthesis.

- They may be continued on the next line by making the first non-blank character a slash (/). A continuation involving a division would thus require a double slash (//).

- They may contain constants, constant names or labels.

For example: if SBB is a format name, and the first variable field is to contain the value 3, the expression is written as follows:

```
SBB↓1 + 2
```

This is the same as SBB↓3 (1 and 2 are expression symbols, + is an expression operator). The expression JMP↓$-5 yields the current value of the program counter minus 5 as the VFS for the first variable field in the format name JMP. ($ and 5 are expression symbols, - is an expression operator). The expression EIGHT: EQU↓2*2*2 means EIGHT = 8 (2's are the expression symbols, *'s are the operators).

EXAMPLES OF CORRECT CONSTANT USAGE

```
QREG:EQU↓#0
     .
     .
     .
AQ:ERQ↓QREG
DQ:EQU↓4+8/6 (value = 2)
AB:EQU↓REG+1
AM2901:DEF↓4V%D#,5X,AQ,3V,17X
```

Definition File

```
EXOR:EQU↓QREG+6
BEGIN:AM2901↓$+2,EXOR
      AM2901↓$-1,AB
```

Source File

VARIABLE FIELD SUBSTITUTES (VFS)

When a format is defined in the definition file, some of its fields may be designated as variable fields. If these fields are not given a default value during their definition or if one wishes to override the default value, a substitution must be made for these field(s) in the source file statements. These substitutes are called variable field substitutes.

REQUIRED SUBSTITUTIONS

If the variable field(s) are not given values in the definition file, values for these fields must be provided in the source file statements. If omitted, an error message will be provided, and processing of that statement ends.

SUBSTITUTIONS SEPARATORS

Each VFS (whether required or optional) represents a single field and must be separated from other VFSs by a comma. Trailing commas may be omitted but the assembler uses the commas to indicate which fields are to be given substitute values (i.e., VFSs are positional and position is determined by the number of commas), so leading or intermediate commas must be given. If the definition file contains:

```
A:  DEF@ 5X, 3V*B#110,2X,5V%H#,B#10101
                  |          |
              field#1    field#2
```

Then the source file is written as TEST3: a@,4. Field #1 will assume the default value 001 (from 3V*B#11) while field #2 will be equated to 0100 and right-justified in the 5-bit field so that field #2 is 00100. The complete microinstruction will be XXXXX 001 XX 00100 10101. If the comma were omitted and TEST4: A@4 were written, the assembler would try to use 4 as the VFS for field #1. Two errors are present: the 4 is not a binary number as required for field #1, and value is not indicated for field #2. The fact that field #2 has no explicit default value and no VFS given are errors. The indication would be illegal character since the 4 is assumed to go with field #1 which requires binary digits.

If the user wishes to input field #1 as 4 and field #2 as zero, write TEST5: A@Q#4,0 which yields the following microinstruction:

```
XXXXX   011   XX   00000      10101
          |          |
      octal 4     hex 0
      inverted    right-
                  justified
```

If a leading or intermediate variable field is to assume a default value, but a trailing field requires a VFS, each field to be skipped must be represented by a comma when forming the microword definition.

This concept is best explained by assuming a format ADE with three variable fields, each having a default value of zero specified in the definition file as follows:

```
ADE:  DEF↓   2VB#000,   3VB#00,   3VB#000
```

Table 3-4 items 1 through 4 illustrates fields which assume default values that are given override or substitute values.

If the variable field substitutions contain modifiers using the following definition file statement:

```
ADE:   DEF↓   3VB#000,   3VB000,   3VB000
```

Then the source file statements for the previous example could be written as shown in table 3-4 items 5 and 6.

The variable fields may contain attributes in the definition file such as:

```
ADE:   DEF↓   3V:H#0,3V*B#000,   3V%B#000
```

The source file statements shown in items 7 and 8 of table 3-4 illustrate this condition.

FITTING VARIABLE SUBSTITUTES TO VARIABLE FIELDS

Any value given as a variable field substitute must contain exactly the number of bits specified (in the definition file) for the total length of the variable field unless the modifiers % (right justification), : (truncation), or $ (paged addressing) are given.

These modifiers may be supplied as attributes with the original field definition (definition file) or they may be supplied with the field substitution value in the assembly file.

PAGED AND RELATIVE ADDRESSING

Paged and relative addressing are achieved by using a $ in the statement. The $ is used in two ways in the assembly file:

a. To indicate that the current value of the program counter is the value to be substituted into this field. This is called relative addressing.

b. As an attribute to indicate that the value substituted for this field must be on the same memory page as the microword into which it is substituted. This is called paged addressing.

TABLE 3-4. SOURCE FILE STATEMENTS

ITEM	INSTRUCTION	RESULTANT MICROWORD DEFINITION	MEANING
1 or 2	TEST6:ADE⍽,,010 TEST7:ADE⍽,,Q#2	000 000 010	Fields 1 and 2 assume their default values, field 3 contains 010.
3	TEST8:ADE⍽Q#4,,B#101	000 000 010	Field 2 assumes its default value, field 1 is 100, field 3 is 101.
4	TEST9:ADE⍽011	100 000 101	Fields 2 and 3 assume their default values, field 1 is 011.
5	TEST10:ADE⍽,,101*	000 000 010	Fields 1 and 2 assume their default values. Field 3 is 101 inverted.
6	TEST11:ADE⍽H#4:	100 000 000	Field 1 is hex 4 (binary 0100) truncated to 100. Fields 2 and 3 assume their default values.
7	TEST12:ADE⍽,,01*	000 111 010	Field 1 assumes its default value 000. Field 2 assumes its default value 111. (000 inverted). Field 3 is inverted to 10 then right justified to be 010.
8	TEST13:ADE⍽9,Q#3*,1	001 011 001	Field 1 is hex 9 truncated to 001. Field 3 is octal 3 inverted to 100, then inverted by field #2 attribute (*) to 011. Field 3 is binary 1 right justified to 001.

For relative addressing, the $ alone or as part of an expression is used as a VFS.

For paged addressing, the $ may be given as an attribute of this variable field in the definition file, or the $ may immediately follow the VFS in the assembly file source statement.

If the definition file contains the following

```
JSR:DEF␣8X,8V$,H#27,12VH#
JSB:DEF␣8V%D#,8X,8Q#013:,12X
```

The source file could be written as shown in table 3-5.

TABLE 3-5. SOURCE FILE PAGED AND RELATIVE ADDRESSING

```
Line #

  1        JSR␣ BEGIN,OBC
  2        JSB␣ MULTI$+5
  3        JSR␣ MULT.BEGIN$
  4        JSB␣ H#35
  5        JSB␣ $+5
                .
                .
                .
           BEGIN: ADD
                .
                .
                .
           MULT:  MPY
```

Lines 1-3 of table 3-5 are examples of $ used for paged addressing. In line 1, the value of the program counter (where BEGIN: appears) is substituted into the first variable field of the format JSR. This value is truncated on the left if necessary to fit into this 8-bit field, and any truncated left bits must be identical to the corresponding bits of the program counter associated with line 1.

The same type of substitution and/or truncation can occur for lines 2 and 3.

JSB on line 2 needs a $ after MULT if paged addressing is desired since no $ was given with that variable field in the definition file. For expressions such as illustrated on line 2, the constant (5) is added to the value of the label (MULT) before the check is made to ensure that the value substituted is still on the correct page. JSR on line 1 needs no $ with the BEGIN since that variable field contained a $ in the definition file. As illustrated on line 2, a label with a $ may be part of an expression. Line 5 is an example of relative addressing.

the current value of the program counter plus 5 will be substituted for the variable field.

> There is no connection between the $ used for paged addressing, as an attribute for a variable field - and the $ used as a variable field substitute to indicate use of the current value of the program counter (relative addressing).

HEXADECIMAL ATTRIBUTE

The designator H#, if given with a variable field in the definition file, is a permanent attribute but it may need to be repeated in the assembly file. This is necessary since the program cannot distinguish a hexadecimal value that begins with an A through F from a label or format name.

If the definition file contains AM2901:DEF␣8V%H#,Q#0,21X and the assembly file statement contains AM2901␣3A, it is clear to the program that the digits 3A are to be substituted into the variable field. (A label or name cannot begin with a numeral).

However, the statement AM2901␣AB does not clearly indicate whether the constant name AB is meant, or the value of the hexadecimal digits AB is meant. If the programmer wishes the hex value AB, write AM2901␣H#AB. The statement AM2901␣AB will substitute the value of the constant named AB in the first variable field. If there is no constant named AB, an error will be generated.

ASSEMBLER SYMBOL TABLE

The symbol table contains a list of all the symbols (constant names) defined by EQUs and all labels in the assembly file. The symbol table also includes all the constant names and their associated values defined, or if the symbol is a constant name (defined EQU), it is followed by the value of the constant.

A symbol table is useful when errors occur due to misspelling or the omission of the colon after a label.

Table 3-6 is a sample symbol table.

TABLE 3-6. SAMPLE SYMBOL TABLE

SYMBOLS	
A	0001
S	0023
X	0000

Printing of the symbol table is optional and is described in the SYMBOL and NOSYMBOL section of table 3-8.

ASSEMBLER ENTRY POINT TABLE

The entry point table contains a list of all the entry point symbols (labels followed by ::) and their associated program counters. These values are useful for mapping PROMs.

Printing of the entry point table is optional and is described in the MAP and NOMAP section of table 3-8.

SOURCE FILE - RESERVED WORDS

The following are reserved words used by the assembler program during the assembly phase. These words MAY NOT BE USED AS LABELS in the source file statements, format names or constant names from the definition file:

ALIGN	NOLIST
EJECT	ORG
END	RES
FF	SPACE
LIST	TITLE

AMDASM OUTPUT FILENAMES, EXECUTION ASSEMBLER OUTPUT

Assembly phase output includes a choice of one of the four types of printed listings shown in table 3-7.

TABLE 3-7. PRINTED LISTING TYPES

TYPE	DESCRIPTION
I	Interleaved format (INTER). One line of source code is printed with the corresponding line of object code printed directly below it.
II	Source only format (SRCONLY). Only the source file statements are printed.
III	Object code only format (OBJONLY). Only the assembly phase object code is printed.
IV	Block format (BLOCK). All lines of source code are printed followed by all lines of the object code.

Each of these listings contains the program counter associated with each line of source and object code.

A final option is to output the binary object code directly to disc for use as input to the post processing phase. (Disc output is independent of the listing option chosen.) The object code on the disc may then be loaded into the microprogrammed controller where the program is debugged.

FILENAMES

Filenames are used to identify unique files on a diskette. They are in two parts, a primary part and a generic part. The format is as follows:

pppppppp.ggg

where: p represents from one to eight characters in the primary part and g represents from one to three characters in the generic part.

All alphanumerics and special characters except for < > . , ; : = ? 8 or a blank may be used for p or g.

In the following section, p refers to primary filenames for the definition file; q refers to primary filenames in the source file. Normally the user will use the same primary name for PHASE 1 and PHASE 2, thus pppppppp will equal qqqqqqqq.

The user may define names for p's or q's that are meaningful for a particular application. The generics listed below must be used in some cases. Mandatory generics are underlined. Generics not underlined are defaults and will be assigned or assumed if not specified by the user.

pppppppp.DEF	Source input for the definition file (PHASE2)
pppppppp.TBL	Output from PHASE1
qqqqqqqq.TBL	Input for PHASE2 -------usually p = q
qqqqqqqq.SRC	Source input for source file (PHASE 2)
pppppppp.P1L	PHASE1 listing output
qqqqqqqq.P2L	PHASE2 listing output
qqqqqqqq.OBJ	PHASE2 output (object code)
qqqqqqqq.MAP	PHASE2 output entry point symbols and their values

When creating the input files pppppppp.DEF and qqqqqqqq.SRC the DEF and SRC generics must be typed as part of the filename when invoking the editor.

EXECUTION

In examples of execution commands, data to be input by the user is underlined. Other data is output by the system.

After the user creates a definition file and source file using AMDOS29 editor, it is ready to execute AMDASM. After the AMDOS29 operating system has issued a user prompt (i.e., the characters A>), the microassembler is executed by entering the following command:

A> AMDASM␣PHASEn=primaryfilename{␣options% cr

Either PHASE1=primary filename or PHASE␣1primary filename specifies execution of both the definition and assembly phases. Thus, A> AMDASM␣PHASE1␣B:KIT cr specifies execution of only the definition phase using the file (on drive B) called KIT.DEF. or A> AMDASM␣PHASE1=B:KIT␣PHASE2=B:KIT cr specifies execution of the definition and assembly phases using the files (on drive B) KIT.DEF as the definition source file and KIT.SRC as the assembly source file.

Either PHASE1 or PHASE2 or both must be specified following AMDASM␣. P1 and P2 are the alternate abbreviated keywords used for PHASE1 and PHASE2, respectively.

The generic part of the filename must not be typed, and either a ␣ or an = used before the primary filename as a delimiter. For example, the following are permissible execution commands for PHASE1:

```
AMDASM␣P1=pppppppp                |This assumes pppppppp.DEF was the
AMDASM␣PHASE1=pppppppp            |name assigned when the definition
AMDASM␣P1␣ppppppppp               |file was created.
AMDASM␣PHASE1␣ppppppppp
```

Following AMDASM␣P1␣primary filename the user then enters the desired options. Options listed in table 3-8 may be given in any order. The full option may be typed (OBJECT) or the abbreviated option may be typed (O). If an option is not typed, AMDASM uses the default option given.

DISK DRIVE DESIGNATORS

Since the AMDASM program is always loaded from the current drive, the user must precede his file names with a drive designator if his input or output files are not on the current drive.

Thus the format for of all filenames will be as follows:

device: primary.generic

Device: is indicated by an A: or B:. A indicates drive A; B indicates drive B.

TABLE 3-8. AMDASM OPTIONS

OPTION	ABBREVIATED OPTION	DEFAULT	MEANING
DEFTBL⌀filename or DEFTBL=filename	D	pppppppp.TBL or qqqqqqqq.TBL	Specifies the name of the file where output of the definition phase is to be stored. When only PHASE2 is executed executed, this specifies the input file which contains the processed definitions. If no DEFTBL⌀filename is given the default name pppppppp. TBL will be used if PHASE 1 is executed; qqqqqqqq. TBL is the default when only PHASE2 is executed.
LIST1⌀filename or LIST1=filename	L1	pppppppp.P1L	Specified where the definition output is to go. When LST: is given as the filename, the output will be listed on the line printer. If no list1⌀filename is given, the output goes to the file with the default name pppppppp.P1L.
LIST2⌀filename or LIST2=filename	L2	qqqqqqqq.P2L	Same as LIST1 except this specifies where the PHASE2 (assembly) output is to go. The default name is the generic P2L appended to the Source File input name (qqqqqqqq.P2L).
NOLIST	NL	qqqqqqqq.P1L and/or qqqqqqqq.P2L	Suppresses listing of PHASE1 and/or PHASE2 output. If not specified defaults to LIST 1 and LIST2. Output goes to files pppppppp.P1L and qqqqqqqq.P2L.
OBJECT⌀filename or OBJECT=filename	O	qqqqqqqq.OBJ	Specifies that the microcode (object code) is to be output on a file with the name (filename). If not given, the microcode is placed on a file with the default name qqqqqqqq.OBJ.
NOOBJECT	NO	qqqqqqqq.OBJ	Suppresses placement of the microcode onto a file. If block format printing is requested, the object code printing is also suppressed. If not specified defaults to OBJECT and the microcode goes to file qqqqqqqq.OBJ.
INTER	IL	BLOCK	Specifies blocked listing format (all lines of source code, then all lines of object code).

TABLE 3-8. AMDASM OPTIONS (continued)

OPTION	ABBREVIATED OPTION	DEFAULT	MEANING
BLOCK	BL	BLOCK	Specifies source-only listing format (prints only the source code.)
WIDTH/n or WIDTH=n	W	BLOCK	Specifies width n, (a decimal number) of lines per page. If not specified, default is 66 lines (11 inches).
LINES/n or LINES=N	LN	n=80	Specifies width n, (a decimal number) of characters for listing device. Default is 80.
MAP/filename or MAP=filename	M	n=66	Specifies listing of entry point symbols (i.e., label symbols designated as entry points by double colons ::) and their associated program counter values is to be output on the list device or onto a list file.
NOMAP	NM	.qqqqqqqq.MAP	Suppresses listing of entry point symbols. If not specified, defaults to MAP and results are stored on a file with the default name qqqqqqqq.MAP.
HEX	H	HEX	Specifies listing of program counter in hexadecimal format.
OCTAL	Q	HEX	Specifies listing of program counter in octal format. If not specified defaults to HEX.
SYMBOL	S	HEX	Specifies listing of constant names and labels and their associated values.
NOSYMBOL	NS	SYMBOL	Suppresses listing of symbol table. If not specified, defaults to SYMBOL.

Examples assume all files are on the current drive. However, when a drive is designated with an input filename, all output default files will be placed on the same drive as the input file for the associated PHASE.

When the user specifies a filename but no drive designator, the file(s) will be placed on the current drive.

EXAMPLES OF AMDASM EXECUTION

Options need to be separated by at least one blank character from other options in the execution command. Whenever a user does not specify an option in his execution command AMDASM will use the default values given in the table 3-8.

The command language for executing AMDASM is illustrated as follows (current drive is assumed to be drive A):

```
A> AMDASM␣P1=2900␣P2=2900 cr
```

This command specifies execution of both PHASE1 and PHASE2 using 2900.DEF as the input file for PHASE1 and 2900.SRC for PHASE2. Defaults are selected for all other options as follows.

```
A> AMDASM␣P1=2900R1 cr
```

This specifies execution of PHASE1 with 2900.DEF as the input source file and 2900R1.TBL as the definition table output file. The command A> AMDASM␣P2=SYSTEM1␣D=2900R1␣IL␣NS cr specifies execution of PHASE2 with SYSTEM1.SRC as the input source file and 2900R1.BL as the definition table input file, interleaved listing format, no symbol table listing, and a list of entry point symbols (by default).

The primary default name for the DEFTBL option may assume the PHASE1 (pppppppp) filename or the PHASE2 (qqqqqqqq) filename as illustrated in Table 3-7. Thus, if the execution command is A> AMDASM␣P1␣AM2900 cr the program will indicate an error since it will be looking for SYSTEM 1.TBL as the filename for the DEFTBK input.

The user may, prior to executing the above command, rename his AM2900.TBL file to be SYSTEM1.TBL. Alternatively, he may execute the command A> AMDASM␣P2␣SYSTEM1␣D␣AM2900 cr indicating the name AM2900.TBL is the DEFTBL input filename.

In either case, PHASE 2 will output files with the following default names (including generics):

SYSTEM1.OBJ	object code generated
SYSTEM1.P2L	PHASE2 listing
SYSTEM1.MAP	mapping PROM file (entry point symbols and their values)

The user may assign only a primary filename to the DEFTBL option. All other options may be given a primary or a primary and generic default option is not used.

SUBMIT FILES

To have AMDOS29 automatically execute an AMDASM command, create a SUBMIT file as follows:

```
A > ED␣name, SUB  cr
NEW FILE
* I  CR  cr
AMDASM␣P1-$␣P2-$2  cr
Control Z
* E  cr
```

SUBMIT files assume the name.SUB file is on the current drive, thus it must be created on the diskette which contains AMDASM and this diskette must be mounted on the current drive.

For execution of the above SUBMIT file, type the following:

```
A> SUBMIT␣name␣ppppppppp␣qqqqqqqq
```

AMDOS29 automatically substitutes ppppppppp for $1, qqqqqqqq for $2.

SUBMIT files are similar to batch jobs since more than one execution command may be part of the SUBMIT file. The user may create a SUBMIT file for one or multiple jobs and need not remain at the console.

A multiple job SUBMIT file is most convenient when the user has a long execution command and/or when he wishes to execute several consecutive assemblies without staying at the console and/or when he wishes to execute the same type of command using many different files. For more detailed information about SUBMIT files, please refer to the AMDOS29 manuals.

SAMPLE OF AMDASM PROCESSING

The capabilities of AMDASM can be demonstrated by microprogramming one of the exercises from the Am2900 Learning and Evaluation Kit. This kit provides a simple but complete example of a microprogrammed system.

The architecture of the kit is shown in figure 3-3. The dashed lines outline the two LSI components, the Am2909 microprogram sequencer and the Am2901 four-bit slice microprocessor. Each microinstruction in the microprogram memory consists of 32 bits divided into fields to control the sequencer, branch address, shift multiplexers, and all the inputs to the Am2901. Figure 3-4 defines the fields and their functions.

The first step in using AMDASM is the creation of a set of definitions reflecting the microprogram hardware statements. Table 3-9 defines, mnemonically, the kits fields. They implement the fields and their functions for microprograms operating in this architecture. Figure 3-4 is an example of those fields and functions. Figure 3-5 is a flow chart of the program to be written. Figure 3-6 is the AMDASM output in block format.

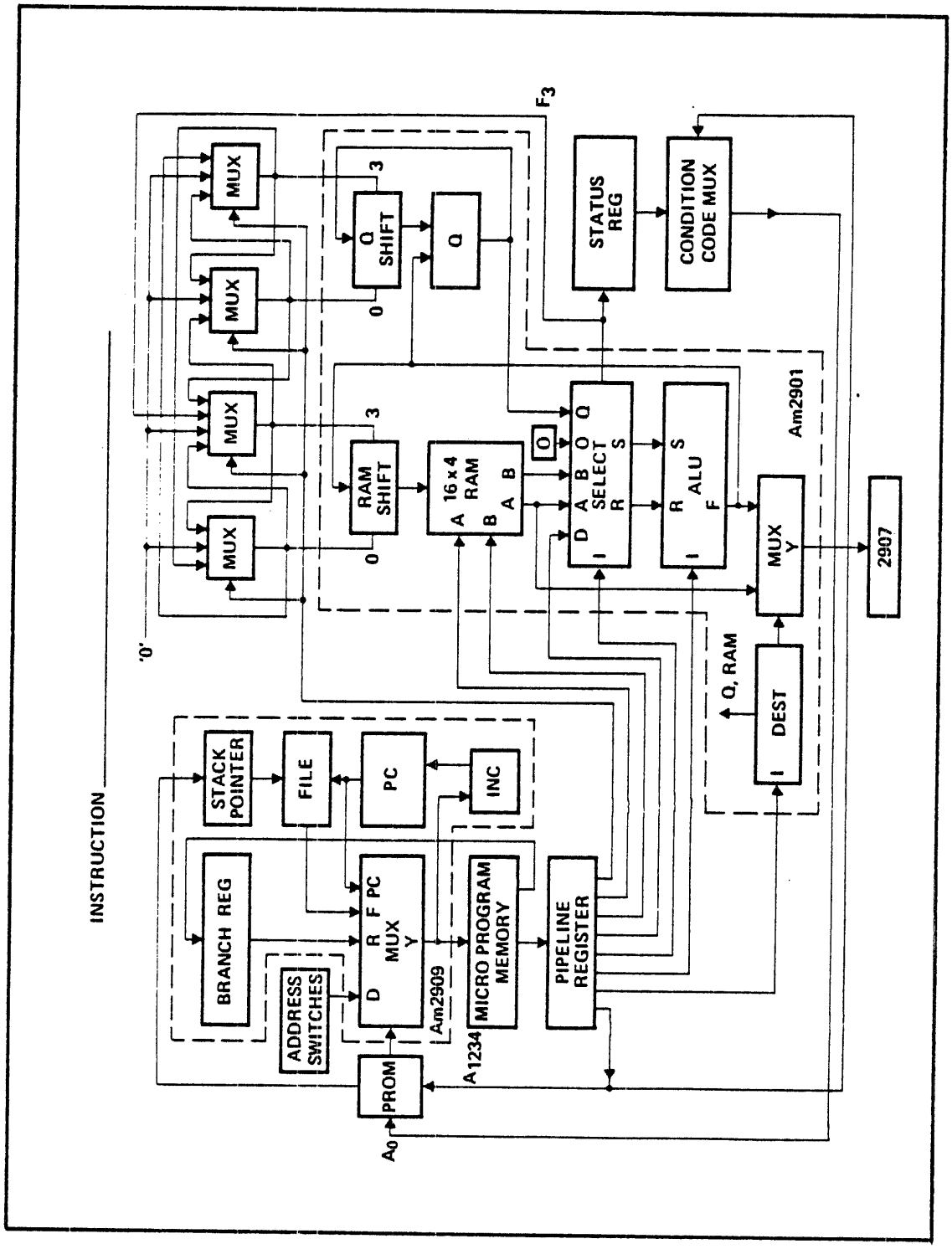

Figure 3-3. Am2900 Learning and Evaluation Kit Architecture

TABLE 3-9. DEFINITION FILE

```
TITLE  AM 2900 KIT DEFINITIONS

;
WORD 32
;
REGISTER DEFINITIONS
;
R0:  EQU H#0
R1:  EQU H#1
R2:  EQU H#2
R3:  EQU H#3
R4:  EQU H#4
R5:  EQU H#5
R6:  EQU H#6
R7:  EQU H#7
R8:  EQU H#8
R9:  EQU H#9
R10: EQU H#A
R11: EQU H#B
R12: EQU H#C
R13: EQU H#D
R14: EQU H#E
R15: EQU H#F
;
;AM2901 SOURCE OPERANDS  (R S)
;
AQ: EQU Q#0
AB: EQU Q#1
ZQ: EQU Q#2
ZB: EQU Q#3
ZA: EQU Q#4
DA: EQU Q#5
DQ: EQU Q#6
DZ: EQU Q#7
;
;AM2901 DESTINATION CONTROL
QREG:  EQU Q#0
NOP:   EQU Q#1
RAMA:  EQU Q#2
RAMF:  EQU Q#3
RAMQD: EQU Q#4
RAMD:  EQU Q#5
RAMQU: EQU Q#6
RAMU:  EQU Q#7
;
;SHIFT  MATRIX CONTROL
:
SHIFT:  DEF 8X,B#0,3XB#0,19X
ROTATE: DEF 8X,B#0,3XB#1,19X
DBLROT: DEF 8X,B#0,3XB#0,19X
ARITH:  DEF 8X,B#0,3XB#1,19X
;NEXT  MICROINSTRUCTION ADDRESS SELECT
;
```

R0 to R15 are using the equate statement. The H# means the numbers following H# are a digit representing 4 bits.

ALU source Q# designates digit.

Defines the two bits which control the left-right shift; x's are don't-care bits in between the defined bits.

TABLE 3-9. DEFINITION FILE (continued)

```
BRFNO;   EQH#0 ;BRANCH REGISTER IF F NOT EQUAL TO ZERO
BR;      EQU H#1 ;BRANCH REGISTER
CONT:    EQU H#2 ;CONTINUE
BM:      EQU H#3 ;BRANCH MAP
JSRFNO: EQU H#4 ;JUMP-TO-SUBROUTINE IF F NOT EQUAL TO ZERO
JSR;     EQU H#5 ;JUMP-TO-SUBROUTINE
RTS:     EQU H#6 ;RETURN FROM SUBROUTINE
STKREF: EQU H#7 ;FILE REFERENCE
LOOPFNO:   EQU H#8 ;END LOOP AND POP IF F=0
PUSH;      EQU H#9 ;PUSH AND CONTINUE
POP:       EQU H#A ;POP AND CONTINUE
LOOPCOUT: EQU H#B ;END LOOP AND POP IF  CN+4
BRFEQO;    EQU H#C ;BRANCH REGISTER  IF F=0
BRF3:      EQU H#D ;BRANCH REGISTER  IF F3
BROVR:     EQU H#E ;BRANCH REGISTER IF OVR
BRCOUT:    EQU H#F ;BRANCH REGISTER IF CN+4
;
;OTHER STUFF
;
CNO:  EQU B#0
CN1:  EQU B#1
LOW:  EQU B#0
HIGH: EQU B#1
ZERO: EQU B#0
ONE:  EQU B#1
;
AM2901: DEF 9X,3VQ#1,1X,3VX,4VX,4VX,4X
AM2902: DEF 4VX,4VH#2,24X
DIN:    DEF 28X,4VH#
;
END
```

Definitions for the sequence control instructions used in the second field of the microinstruction.

Format definitions are made for the ALU fields, the sequence control fields, and the data input. Formats contain don't cares (x) and variables (v). Each variable can have a default value. For example, in AM2902, trhe second four-bit variable defaults to hex 2, and the first four-bit variable defaults to x.

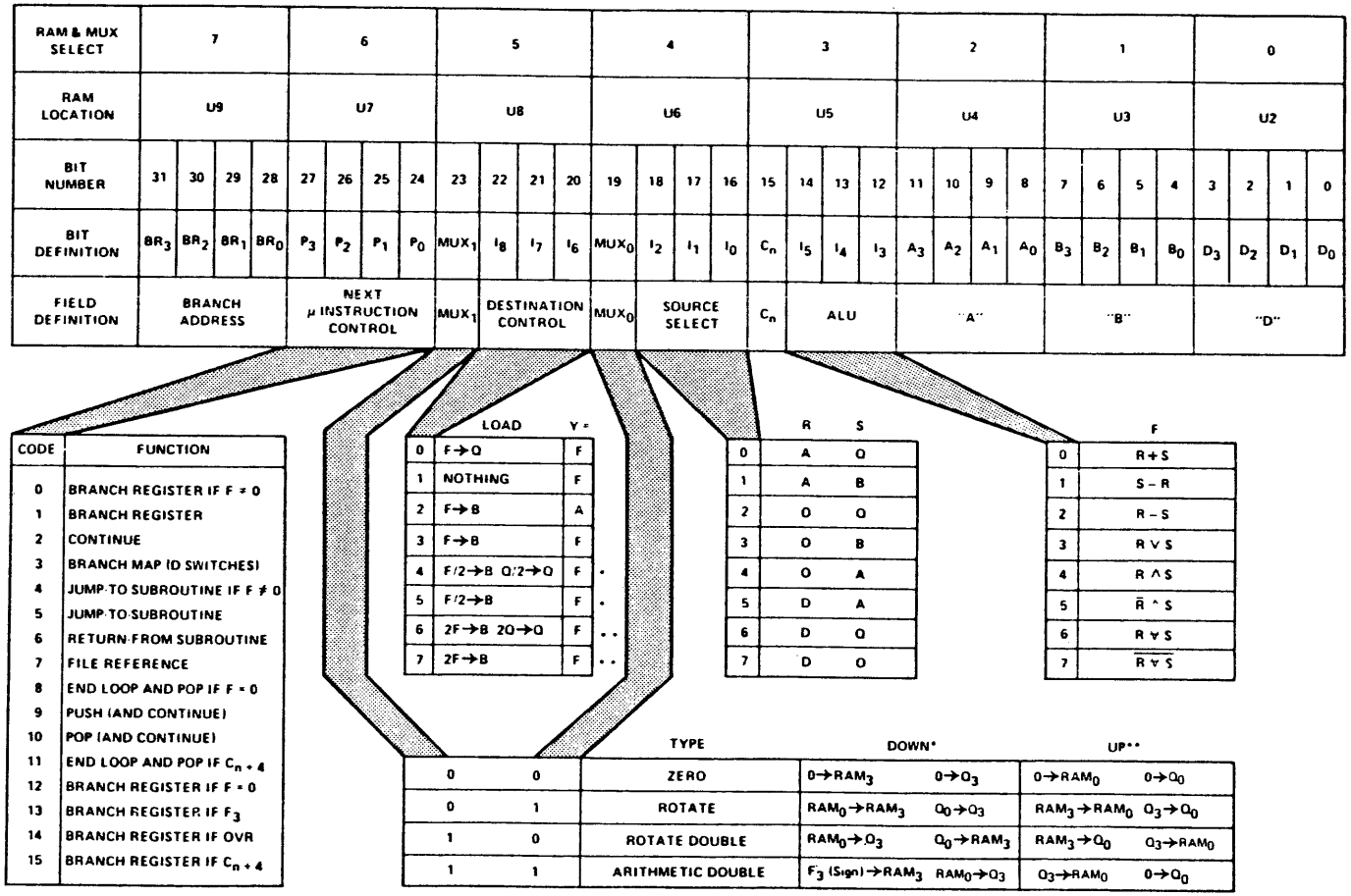

Figure 3-4. Example of Fields and Functions

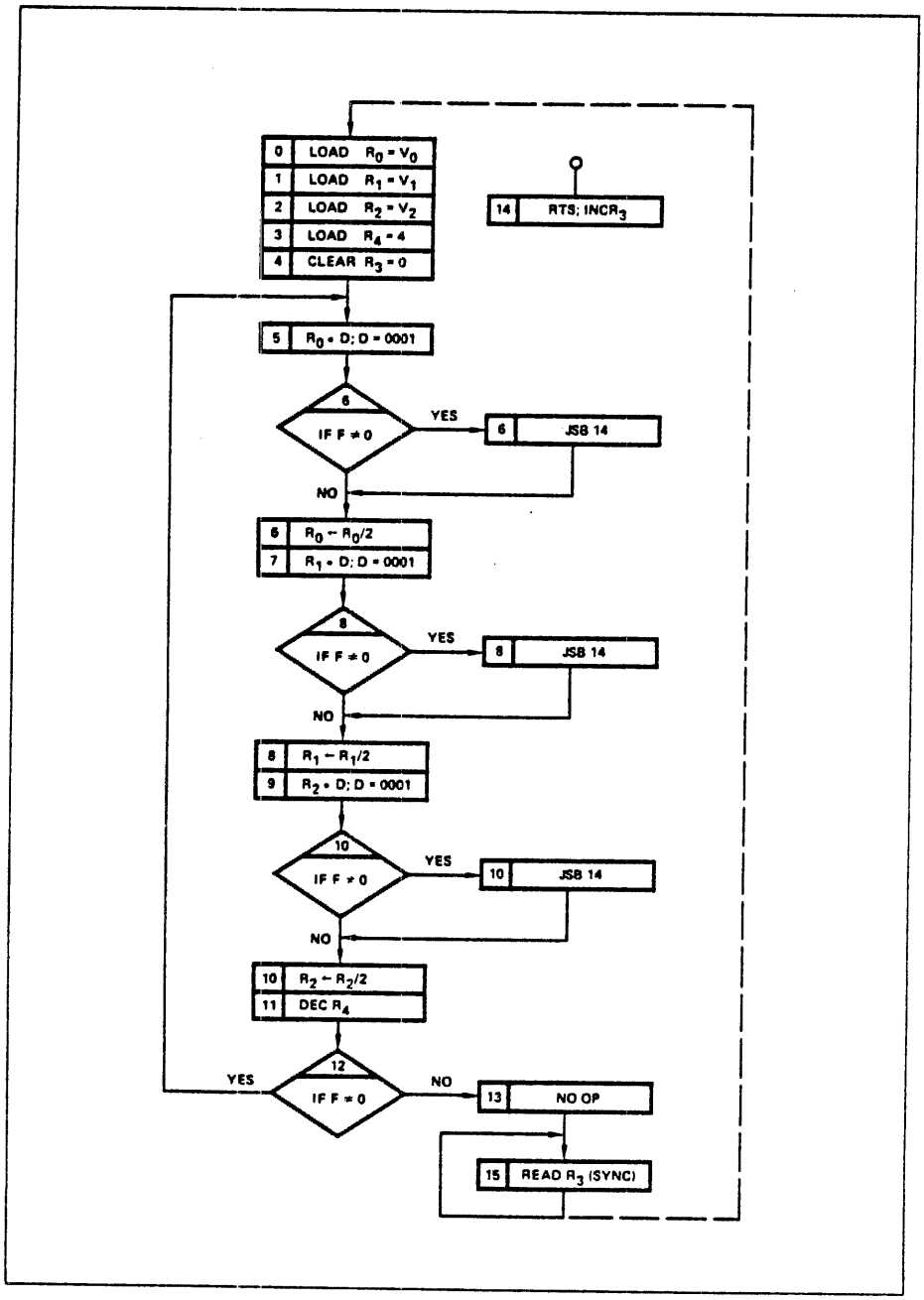

Figure 5-4. Flow Chart of Example

Figure 3-5. Flow Chart of Example

```
0000      AM2909 & AM2901   RAMF, DZ,,OR,,RØ & DIN H#F
0001      AM2909 & AM2901   RAMF, DZ,,OR,,R1 & DIN 9
0002      AM2909 & AM2901   RAMF, DZ,,OR,,R2 & DIN Ø
0003      AM2909 & AM2901   RAMF, DZ,,OR,,R4 & DIN 4
0004      AM2909 & AM2901   RAMF, ZB,,AND,,R3
0005 A5:  AM2909 & AM2901    ,DA,,AND,RØ,RØ & DIN 1
0006      AM2909 A14,JSRFNØ & AM2901  RAMD, ZB,,OR,,RØ
0007      AM2909 & AM2901   ,DA,,AND,R1,R1 & DIN 1
0008      AM2909 A14,JSRFNØ & AM2901  RAMD, ZB,,OR,,R1
0009      AM2909 & AM2901   ,DA,,AND,R2,R2  & DIN 1
000A      AM2909 A14,JSRFNØ & AM2901  RAMD, ZB,,OR,,R2
000B      AM2909 & AM2901  RAMF, ZB, CNØ,SUBR,,R4
000C      AM2909 A5,BRFNØ & AM2901
000D      AM2909 A15,BR & AM2901
000E A14: AM2909 ,RTS & AM2901  RAMF, ZB,CN1,ADD,,R3
000F A15: AM2909 A15,BR & AM2901   ,ZB,,OR,,R3
     END

0000 XXXX0010X011X111 X011XXXX00001111
0001 XXXX0010X011X111 X011XXXX00011001
0002 XXXX0010X011X111 X011XXXX00100000
0003 XXXX0010X011X111 X011XXXX01000100
0004 XXXX0010X011X011 X100XXXX0011XXXX
0005 XXXX0010X001X101 X100000000000001
0006 11100100X101X011 X011XXXX0000XXXX
0007 XXXX0010X001X101 X100000100010001
0008 11100100X101X011 X011XXXX0001XXXX
0009 XXXX0010X001X101 X100001000100001
000A 11100100X101X011 X011XXXX0010XXXX
000B XXXX0010X011X011 0001XXXX0100XXXX
000C 01010000X001XXXX XXXXXXXXXXXXXXXX
000D 11110001X001XXXX XXXXXXXXXXXXXXXX
000E XXXX0110X011X011 1000XXXX0011XXXX
000F 11110001X001X011 X011XXXX0011XXXX
```

Figure 3-6. Assembly Output in Block Format

CHAPTER 4
EMULATOR SUPPORT SOFTWARE

INTRODUCTION

The microprogramming software consists of a number of programs, each having its own command. Each command has a multitude of operands and/or subcommands with operands. Any number of operands and/or subcommands, up to a maximum of 127 characters, may be input on a single command line at the CRT console. The basic microprogramming software commands consist of the following:

Load Bipolar Memory (LBPM)

Verify Bipolar Memory (VBPM)

Save Bipolar Memory (SBPM)

Restore Bipolar Memory (RBPM)

Dynamic Debugging Tool 29 (DDT29)

Any number or combination of the legal delimiters may be used between a command and the associated subcommands and operands. Thus, the user can structure a command line in a form that is easily readable.

Once the initial version of the prototype microcode is input at the CRT console and assembled by the AMDASM assembler, the microprogramming software is used. Figure 4-1 shows the interrelationship between AMDASM and part of the microprogramming software. The LBPM (Load Bipolar Memory) command is used to load the assembled object file into WCS (Writable Control Store). The load is automatically verified unless the NOVerify option is selected. Alternatively, the load can be separately verified using the VBPM (Verify Bipolar Memory) command. Bits selected as don't care bits in the microword can be loaded as zeroes, ones, or not altered in memory. In addition, any portion or all of the object file can be loaded anywhere in memory. In a similar manner, the microcode address entry points can be assembled and loaded into the mapping memory on the microprogram sequencer (MSC) card. As with the WCS card contents, the mapping memory contents can be verified by invoking the VBPM command.

After the prototype microcode is stored in the WCS memory and address entry points are stored in the MSC card mapping memory, the DDT29 program is used to debug the user's microcode. That is, if the hardware responses are inappropriate for a portion of the microcode execution, a microcode address breakpoint can be set into a register in the Clock Control Logic (CCL) and the microcode can be rerun. When the selected address occurs, microcode execution stops. At that point, the

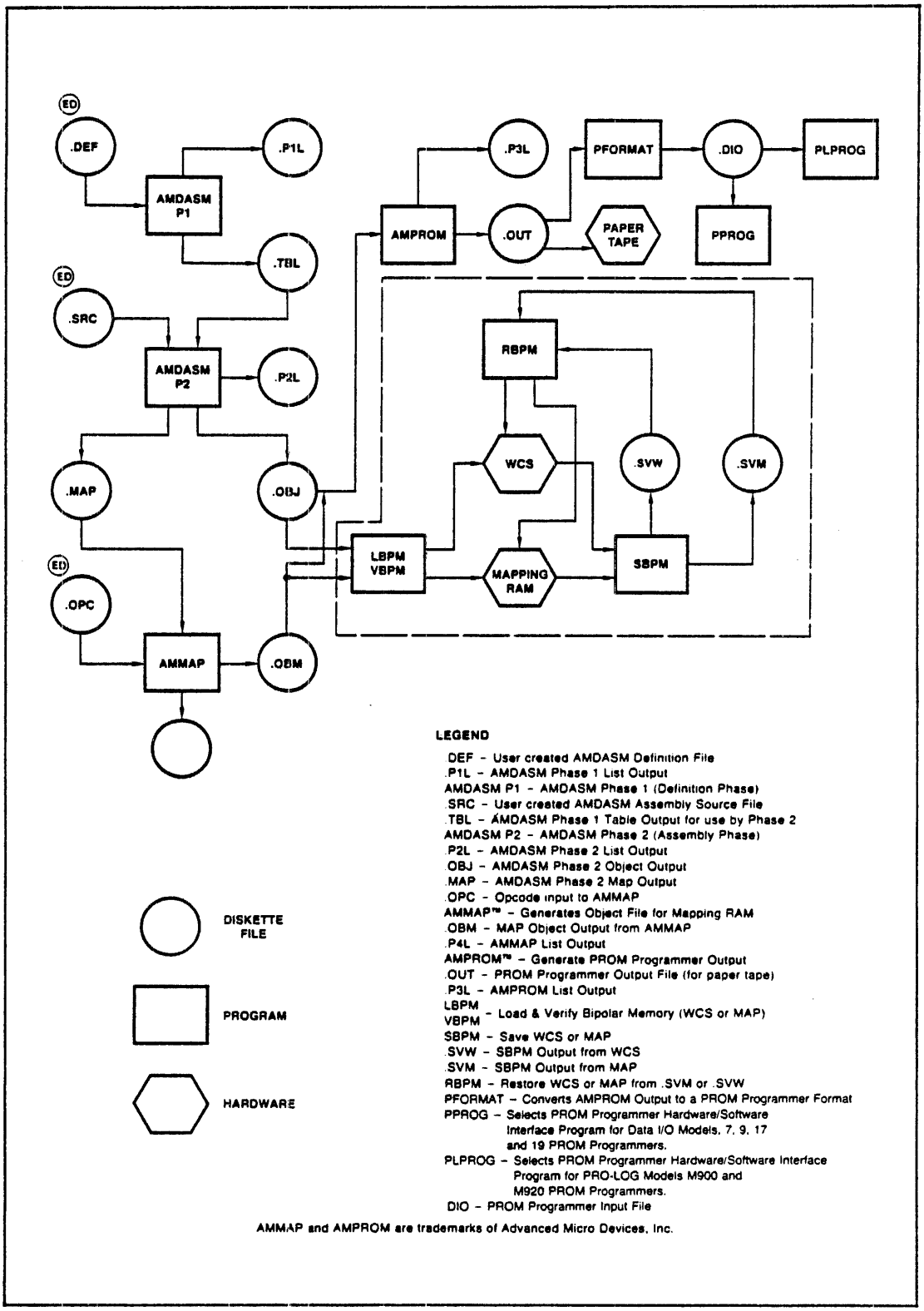

Figure 4-1. AMDASM/Microprogramming Software Relationship

user can examine up to 20 selected (user-wired to connector P4 of the CCL; refer to drawings in hardware manual) test points or data bits in the prototype hardware circuit.

If results are not as anticipated, DDT29 can be used to display the last microinstruction or any desired continuous number of microinstructions on the CRT console. If the user decides to alter the microcode, the modification can be entered at the CRT console via DDT29 or assembled as previously described.

The assembled output creates a new object file on diskette. Then, the LBPM command is used to move the new microcode file to the WCS. A DDT29 subcommand can be invoked to jam a microcode address onto the prototype system WCS address bus.

Thus, the entry point of the last microcode routine or microinstruction executed can be addressed and re-executed. While testing the portion of modified microcode, the user can again monitor the selected test points to verify that the modification is successful.

If microcode operation is still not satisfactory, more of the many DDT29 subcommands can be used to examine both prototype microcode and hardware. For example, the microprogrammed controller clock (MPC) can be controlled from the CRT console by inputting run, halt, or single-step (SS) subcommands. The SS subcommand permits the microprogrammed system clock to be stepped one phase at a time while observing hardware responses to the various stages of microcycle execution.

Once the prototype microcode and hardware operation are satisfactory, the developed and debugged microcode can be stored in a diskette file through the use of the SBPM command. If, at a later date, more testing or development is required, execution of the RBPM command causes the saved microcode to be restored in the WCS memory.

Alternatively, the saved microcode can be used by AmSYS29 to drive a PROM programmer, or output as a listing on a printer. When the user is finished with microcode development, the DDT29 exit subcommand is used to return AmSYS29 operation to the AMDOS 29 program.

Table 4-1 provides a summary of the microprogramming software commands. A command format summary is included in chapter 3 of the user manual.

TABLE 4-1. MICROPROGRAMMING SOFTWARE COMMANDS

COMMAND/SUBCOMMAND	GENERAL DESCRIPTION
Load Bipolar Memory (LBPM)	Loads machine language microcode from AMDASM generated diskette file into WCS memory or MSC card mapping memory. Can load any portion of diskette file into any portion of WCS or MSC card memory.
Verify Bipolar Memory (VBPM)	Verifies that loaded contents of WCS or MSC memory corresponds to originating file on diskette. Can verify any portion of AMDASM generated diskette file against a corresponding portion of WCS or MSC card memory.
Save Bipolar Memory (SBPM)	Saves WCS or MSC Card memory contents in new diskette file created by SBPM command. Any specified portion of the WCS or MSC Card memory contents can be saved.
Restore Bipolar Memory (RBPM)	Restores specified diskette file to either the WCS or MSC card memory.
Dynamic Debugging Tool 29 (DDT29)	Entry to Microprogrammed System debug environment.
	The following DDT29 subcommands perform specific functions.
Display WCS Contents (DW)	Display WCS contents from specified address for a specified number of words or through a specified address.
Display MSC Mapping Memory Contents (DM)	Display MSC card mapping memory contents from a specified address for a specified number of words or through a specified address.
Status (STatus)	Display contents of CCL status register.
Store Data in WCS Memory (SW)	Specified number of data bytes are stored in WCS memory beginning at specified address and specified byte. Multiple data bytes replace microword bytes beginning at specified byte and continuing to right until right-most byte of microword is reached.
Store Data in MSC Memory Mapping (SM)	Specified number of data bytes are stored in CCU card mapping memory beginning at specified address and specified byte. Multiple data bytes replace previous bytes beginning at specified byte and continuing to right until right-most byte is reached.
Halt	Stops microprogrammed controller
Run	Enables microprogrammed controller to run.
Single-step CCL (SS)	Permits one or more clock phase steps to be run by clock control logic.

TABLE 4-1. MICROPROGRAMMING SOFTWARE COMMANDS (Continued)

COMMAND/SUBCOMMAND	GENERAL DESCRIPTION
Microcyle Step CCL (MS)	Permits one or more clock phase steps to be run by CCL.
Store Value in CCL Control Register (CTL)	Stores specified value in CCL control register.
Jam Address (Jam)	Jam CCL address register contents onto WCS card address bus.
Store Jam Address (IR)	Store specified address into CCL address register.
Z (n)	Stop executing DDT29 subcommands for the specified number of milliseconds.
Macro (n)	Execute one or a string of DDT29 subcommands for the specified number of times.
Display Last Address (DLA)	Display contents of CCL register that contains the address of the last microinstruction executed.
Display Monitor Bits (DMB)	Display 20 monitor bits of user selected and wired test point or other data in hexadecimal format at CRT console.
Display Trace (DTR)	Display microprogram address, data at that address, and monitor bits of the last microcyle.
Leave DDT29 Program (Exit)	Return AmSYS29 operation to AMDOS 29 program.

MICROPROGRAMMING SOFTWARE COMMANDS

In the following command description, all default values are underlined. The use of each command or subcommand is presented along with execution format, definitions of associated operands and usage rules pertaining to a particular command.

The following constants are used in the descriptions:

X = hexadecimal
d = decimal (0-9)
b = binary (1 or 0)
N = required radix indicator where:

H	= hexadecimal
Q or O	= octal
B	= binary
Nothing or D	= decimal

Capital letters included in the execution formats are required for proper command execution. Lower case letters are optional.

To load, modify, display, or in any other way access the WCS, the clock control logic clock must first be halted. The clock can be stopped by invoking the DDT29 halt subcommand or by setting the system mainframe front panel RUN ENABLE/HALT switch to the HALT position. To restart the MPC clock system, the mainframe front panel switch or a software command can be used.

System mainframe RUN ENABLE and MICRO CYCLE STEP can be pressed to restart the clock. When the DDT29 halt subcommand is used to stop the clock, then the DDT29 run command is used to restart it.

LOAD BIPOLAR MEMORY (LBPM)

The LBPM command is used to transfer an AMDASM machine language object file from diskette to either the WCS memory or the mapping memory.

The execution format of the LBPM command is as follows:

```
                |   |      |     |           |       |LOwer|         |
LBPM filename|WCs|FRom X|TO X |WA X  LSb d|NOClear|UPper|VErify   |DC b SBft
             |MAp|      |FOr X|           |CLear  |UL   |NO Verify|
```

During the AMDASM assembly process, a start address is assigned by the user to the first microword in the diskette file being built. If the user does not assign a start address, AMDASM assigns default address 0000 to the first microword. A program counter is maintained by AMDASM so the address for each successive microword can be incremented by one. The PC address associated with each microword is then recorded on the diskette along with the microword.

When the LBPM command is invoked to transfer microcode from diskette to the WCS or mapping memory, the LBPM program first locates the appropriate file by the filename specified in the LBPM command. Then, the LBPM program searches for the starting PC address (which can be specified by the FRom X operand) from which the data transfer begins. The LBPM program begins to transfer microcode from that address to the correct address in the WCS or mapping memory formed by adding the PC

address to the WA address specified. The default for WA is zero if unspecified. Microcode transfer from the diskette file to the WSC or mapping memory continues until the ending PC address (as specified by the TO X operand in the LBPM command) is reached, or the correct number of microwords are transferred (as specified by the FOr X operand in the LBPM command) or, until end-of-file on the input file is reached. An example of the LBPM command is presented below and graphically illustrated in figure 4-2.

```
LBPM JIM WCs FR 200 TO 300 WA 200 NOC UL NOV cr
```

In the LBPM command example, the diskette file called JIM.OBJ is accessed and the block of microcode stored between PC addresses 200H and 300H is loaded into the WCS starting at WCS address 400H (FR of 200H plus WA of 200H = 400H). Any data previously residing in WCS addresses 400H thru 500H is not cleared before the diskette file data is loaded into WCS from diskette file and JIM.OBJ is not to be verified.

NOTE

File type (OBJ) is always read even if different extension is typed.

The Don't Care (DC b) and Set Bipolar Format (SBft) operands are not used in the preceding LBPM command example. Consequently, the DC b operand, which is used to specify whether don't care bits are set to 1 or 0, automatically defaults to 0 unless a default value of 1 exists in the bipolar format table for the WCS.

The bipolar format tables for the WCS and MAP reside in the uppermost area of the 64K byte RAM associated with the system processor. Operand default values that supercede the built-in default values for the commands can be loaded into the bipolar format tables. The bipolar format tables are loaded with operand default values by specifying the SBft operand with the LBPM command.

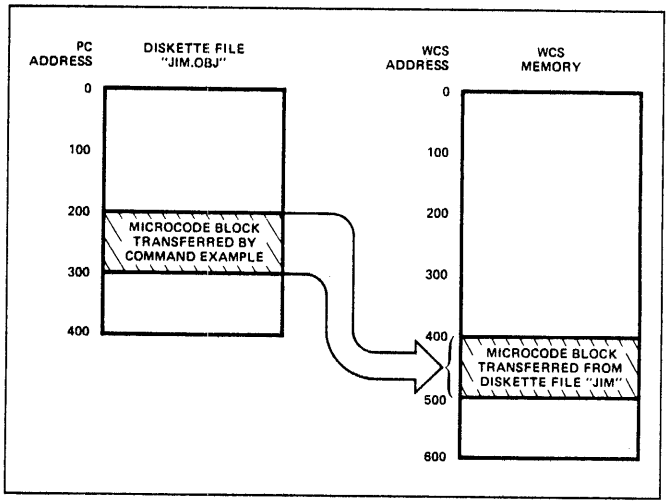

Figure 4-2. LBPM Command Example

When the SBft operand is used with the LBPM command, the values given for the remaining operands in the LBPM command are stored in the bipolar format table for the WCS or MAP. Since the SBft operand is not used in the previous LBPM Command example, the other operand values presented in the example are not stored in the WCS bipolar format table.

The operand descriptions for the LBPM command are as follows:

- WCs indicates that the WCS memory is to be loaded from diskette filename.OBJ. The .OBJ is assumed and need not be entered.

- MAp indicates that mapping memory is to be loaded from diskette filename.OBM. The .OBM is assumed and need not be entered.

- Filename designates name of diskette file to be loaded into WCS or mapping memory.

- FRom designates the value of the program counter address on the diskette file from which the transfer to bipolar memory is to start. The default value is the first PC address on the diskette file.

- TO designates the ending PC address within the diskette file. Default value is the end of file.

- FOr designates the number of microwords to load. Default value is the entire file. TO and FOr are mutually exclusive.

- Writable address (WA) designates the starting address in the WCS memory or the mapping memory offset. The offset is added to the PC address to form the memory address.

- Least significant bit (LSB d) designates the position d in bipolar memory of the least significant bit (127-0) of the microword. The position starts from right to left. Default value is right justified (d=0).

- NOClear indicates that each microword in the WCS or MSC card memory area to be loaded is not cleared before loading a new word from the diskette file. Default is NOClear.

- CLear designates that each microword in the WCS or MSC card memory area to be loaded is cleared before loading a new word from the diskette file. Default is NOClear.

- LOwer indicates that only bits 63-00 of WCS memory are implemented. Default is LOwer.

- UPper indicates that only bits 127-64 of WCS memory are implemented. Default is LOwer.

- Upper/Lower (UL) indicates that all 128 bits of WCS memory are used. Default is LOwer.

- VErify causes the VBPM program to be automatically executed after the WCS or mapping memories are loaded. VBPM verifies that memory is correctly loaded. Default is VErify.

- NOVerify indicates that VBPM is not to be executed after loading memory. Default is VErify.

- Don't Care (DC) specifies whether don't cares are set to 0 or 1. Default value is 0.

Set bipolar format table (SBft) places the given operands into the appropriate bipolar format table as subsequent default values. The SBft operand is for user convenience. That is, the bipolar format table can be loaded with default values and the bipolar memory can be loaded with microcode by using only one command (LBPM).

The following usage rules apply to the LBPM command:

1. Either the WCs or MAp operand must always be specified.

2. Two bipolar format tables exist to service either the WCS memory or the mapping memory. Once a bipolar format table is loaded with default values, those values supercede the default values given in the execution format paragraph.

VERIFY BIPOLAR MEMORY (VBPM)

The VBPM command is used to verify that the data loaded into the WCS or mapping memory is identical to that contained on the diskette source file. A byte-by-byte comparison is made between the WCS or mapping memory contents and the original file on the diskette. If an error exists, the VBPM program causes VERIFY ERROR to be printed out at the CRT console and/or printer along with the WCS or MAP data and the diskette data. An example of verify error is as follows:

The execution format of the VBPM command is as follows:

```
                |   |       |     |             |       |LOwer|
VBPM filename|WCS|FRom X|TO X |WA X  LSb d|NOClear|UPper|DC b
                |MAP|       |FOr X|             |CLear  |UL   |
```

The operand descriptions for the VBPM command are as follows:

- Filename designates name of diskette file against which WCS or mapping memory is verified.

- WCs indicates that the WCS memory is to be verified against the diskette file.

- MAp indicates that the mapping memory is to be verified against the diskette file.

- FRom designates the starting PC within the diskette file that is used for verification. Default is the starting file PC.
- TO designates the ending PC within the diskette file that is used for verification. Default is the entire file.
- FOr designates the number of microwords verified. Default is the entire file. TO and FOr are mutually exclusive.
- WA designates the WCS or mapping memory offset.
- LSB designates the least significant bit number of the microword. Default value is 0, right justified.
- NOClear causes only the microword width to be verified. Default is NOClear.
- CLear causes the entire WCS or MAP width to be verified. Default is NOClear.
- LOwer indicates that only WCS bits 63-00 are verified. Default is LOwer.
- UPper indicates that only WCS bits 127-64 are verified. Default is LOwer.
- UL indicates that all WCS bits are verified. Default is LOwer.
- DC specifies whether don't cares are set to 0 or 1. Default value is 0.

The following usage rules apply to the VBPM command:

1. Either the WCs or MAp operand must always be specified.
2. If WCS and/or mapping memory default values are loaded into the bipolar format table, then those default values supercede the default values given for the VBPM command in the execution format paragraph.

SAVE BIPOLAR MEMORY (SBPM)

The SBPM command creates a file on diskette from the WCS or mapping memory.

The execution format of the SBPM command is as follows:

```
                |WCs|       |TO X
SBPM filename|MAp|FRom X|FOr
```

The operand descriptions for the SBPM command are as follows:

Filename specifies the name of the file created on diskette to receive WCS or mapping memory contents. A three-letter name is appended to the specified filename to form the complete filename as follows: Filename.SVW for WCS memory and Filename.SVM for mapping memory.

Where:

WCs specifies contents of WCS memory are to be saved.

MAp specifies contents of mapping memory are to be saved. The entire 256 word map is saved.

FRom specifies the starting microword address of the microcode in the WCS to be saved. Default value is 0.

TO specifies the ending microword address (in the WCS) of the microcode to be saved. Default value is FRom; that is, a single microword is saved.

FOr specifies the number of microwords to be saved. Default value is 1. TO and FOr are mutually exclusive.

The following usage rules apply to the SBPM command:

1. Either WCs or MAp must always be specified.

2. The FRom, TO, and FOr operands are valid for WCs only.

RESTORE BIPOLAR MEMORY (RBPM)

The RBPM command is used to restore a specified saved file to either the WCS or mapping memory.

The execution format for the RBPM command is as follows:

```
                |SVW
RBPM filename|SVM
```

The operand descriptions for the RBPM command are as follows:

- Filename.SVM indicates name of saved mapping memory file on diskette.

- Filename.SVW indicates name of saved WCS memory file on diskette.

NOTE

The complete filename must be specified for RBPM.

DYNAMIC DEBUGGING TOOL 29 (DDT 29)

The DDT 29 command consists of a number of subcommands that are used during the development of the prototype microcode.

The command DDT29 places AmSYS29 in the DDT29 environment so the various subcommands can be used during the development process.

The subcommands can be entered on the same or subsequent lines after the . prompt. Any DDT29 subcommand string can be up to the 128 characters in length. If more than 128 characters are entered, the DDT29 subcommand invoked responds as if a carriage return (cr) is initiated. The subcommands include:

DISPLAY
STORE
STATUS
HALT
SINGLE STEP
MICROCYCLE STEP
RUN
CONTROL REGISTER STORE
ADDRESS REGISTER STORE
JAM ADDRESS
SLEEP
DISPLAY TRACE
MACRO
DISPLAY LAST ADDRESS
DISPLAY MONITOR BITS
EXIT

The following paragraphs individually cover the DDT29 subcommands.

DISPLAY

The display subcommand permits microcode in the WCS or mapping memory to be displayed from a specified address for a specified number of words or through a specified address. The execution format of the display subcommand is as follows:

```
  |X.Y  |      |X.Y |
Dw|X  ØY|      |X  Ø|Y
```

Where:

- Dw indicates the WCS memory is to be displayed.
- DM indicates the mapping memory is to be displayed.
- X.Y specifies that Y words are to be displayed from starting address X. Initially, default is first 16 words. Thereafter, default is 16 words from last address displayed.

- X¢Y specifies that microcode from address X through address Y is to be displayed.
- X and Y are in hexadecimal notation.

STORE

The store subcommand stores one or more data bytes into the WCS starting at a specified address and byte. Multiple bytes replace bytes in the WCS by beginning at the specified byte and continuing to the right until the right-most specified byte is reached. Mapping memory data is loaded in three-byte fields, beginning with the specified field address and continuing in sequence for each additional three-byte data field entered. Leading zeroes need not be entered. For a mapping memory store, a delimiter must always be used between data byte entries. Figure 4-3 shows the relationship between the data bits and bytes as typed in at the CRT console. The execution format of the store subcommand is as follows:

Sw X Y D or SM X D

Where:

- SW indicates store data in WCS.
- SM indicates store data in mapping memory.
- X is a hexadecimal value that indicates the WCS or mapping memory address for the data store at which the loading starts.
- Y is a hexadecimal value that indicates the starting byte for the data store.
- D is a hexadecimal value that represents the data nibble, byte, string, or field.

Bit Number	127	126	125	124	123	122	121	120	119..8	7	6	5	4	3	2	1	0
Byte Number	0								1....E		F						
Nibble Typed-In At CRT Console	8				F				8		6						
Binary Value Loaded	1	0	0	0	1	1	1	1	1	0	0	0	0	1	1	0	

Figure 4-3. Relationship Between Data Bits and Bytes for DDT29 Store Subcommand

For example, if the following is typed at the CRT console:

```
Sw  100  C  1A9EF38  cr
```

then data loading begins at WCS address 100, byte number 12 in accordance with the information presented in Table 4-2.

TABLE 4-2. DATA LOADED INTO WCS BY STORE SUBCOMMAND EXAMPLE

WCS ADDRESS	HEXADECIMAL BYTE NUMBER	HEXADECIMAL DATA INPUT AT CRT CONSOLE	BINARY DATA EQUIVALENT LOADED INTO WCS
100	C	1A	00011010
100	D	9E	10011110
100	E	F3	11110011
100	F (high order nibble only)	8	1000

A special delimiter must be used to terminate the data input D whenever the command is not followed by a carriage return. This is the case when a number of DDT29 commands are entered on a single line.

The special delimiter is a slash. It must follow the data D included in the SW or SM command. The slash must be followed by at least one standard delimiter such as ⁄, = ().

Example:

```
SW  5  8  1E2A/Dw  5.1
```

changes byte 8 to 9 of microword 5 to 1E and 2A, then displays the changed microword.

Only high order nibbles can be entered as nibble-only increments. If only the low order four-bit nibble of a byte is to be changed, the existing high order four-bit nibble must also be entered at the CRT console to prevent changing that nibble value.

NOTE

All 128 microword bits must be accounted for at all times.

NOTE

References made in the following paragraph to the control register can be clarified by reading the paragraph entitled Control Register Store Subcommand that is presented later in this chapter.

STATUS

The status subcommand is used to display the contents of the CCL status register. The status register contents indicate what caused the CCL clock to stop. Causes include: one of four user-wired trap bits is active, an address comparison occurred, control register mask bit 5 is low, a hardware stop, or a software stop.

When the status subcommand is invoked, the status register contents are displayed at the CRT console as a two-character hexadecimal number. The hexadecimal number represents the status of the eight status register bits. Any low bit indicates what stopped the CCL clock. Table 4-3 summarizes the status register bit assignments.

TABLE 4-3. STATUS REGISTER BIT ASSIGNMENT SUMMARY

STATUS BIT	FUNCTION	DESCRIPTION
0 1 2 3	Trap bit 0 Trap bit 1 Trap bit 2 Trap bit 3	When any user-wired trap bit is low and the associated control register mask bit is low, the CCL clock is stopped and the associated status register bits 0 thru 3 and trap bits 0 through 3.
4	Address Compare	When control register mask bit 4 is set low and an address comparison occurs, the address com pare status bit 4 is set low to indicate an address comparison stopped the CCL clock during address breakpoint operation.
5	Clock Disable	When control register mask bit 5 is low, the CCL clock cannot run and status register bit 5 is low.
6	Software Clock Enable	When the CCL clock is stopped by execution of a software halt command, status register bit 6 is set low.
7	Hardware Clock Enable	When the CCL clock is stopped by setting the front panel RUN ENABLE/ HALT switch in the HALT position, status register bit 7 is set low.

Figure 4-4 shows an example of a status subcommand display at the CRT console and how the two-character hexadecimal presentation is interpreted. In the example of Figure 4-4, an address comparison stopped the CCL clock.

The execution format of the status subcommand is as follows:

STatus								
HEX NUMBER DISPLAY AT CRT CONSOLE	E				F			
BINARY EQUIVALENT OF HEX NUMBER	1	1	1	0	1	1	1	1
STATUS REGISTER BIT NUMBER	7	6	5	4	3	2	1	0

Figure 4-4. Example of Status Subcommand Display and Interpretation

HALT

The halt subcommand stops the microprogrammed controller clock. The execution format is as follows:

```
Halt
```

SINGLE-STEP

The single-step subcommand permits the microprogrammed controller clock to be stepped one or more phases at one time. The execution format for one single-step subcommand is as follows:

```
SS n
```

Where:

- SS denotes single-step.
- n specifies number (1-65535) of clock phases to be stepped. Default for n is one.

MICROCYCLE STEP

The microcycle step subcommand permits the microprogrammed controller clock to be stepped one or more microcycles at a time. The execution format for the microcycle step subcommand is as follows:

MS n

Where:

- MS denotes microcycle step.
- n specifies number (1-65535) of microcycles to be stepped. Default for n is one.

RUN

The run subcommand causes the microprogrammed controller clock to run. The execution format of the run subcommand is as follows:

Run

CONTROL REGISTER STORE

The control register store subcommand is used to store mask bits that control various functions on the CCL. The subcommand stores the binary equivalent of a two character hexadecimal value (input at the CRT console) into the clock control logic control register. Table 4-4 describes the assignments of the eight control register mask bits.

The execution format of the microprogrammed controller subcommand is as follows:

CTL XX

Where:

XX is a hexadecimal byte representing the eight mask bit values.

Example:

CTL 6F cr

Figure 4-5 shows the relationship between the two-character hexadecimal value entered at the CRT console in the example and the individual mask bits. In the example, all trap bits are inhibited, address comparison is enabled, the MPC clock is enabled to run, CCL interrupt 5 is serviced whenever the MPC clock stops, and the contents of the clock control logic address register can be jammed onto the WCS card address

TABLE 4-4. CONTROL REGISTER MASK BIT ASSIGNMENTS

MASK BIT	FUNCTION	DESCRIPTION
0	Trap Bit 0 Enable/Inhibit (0) (1)	
1	Trap Bit 1 Enable/Inhibit	When any of the first four mask bits (0 thru 3) is high, the associated user-implemented trap bit is inhibited. That is, the trap bit is unable to stop the MPC clock.
2	Trap Bit 2 Enable/Inhibit	
3	Trap Bit 3 Enable/Inhibit	
4	Address Comparison Enable/Inhibit (0) (1)	When mask bit 4 is high, the output of the CCL address comparator is inhibited. That is, address breakpoint operation is inhibited.
5	MPC Clock Enable/Inhibit (1) (0)	When mask bit 5 is high, the MPC clock is enabled to run, providing all other clock conditions are met. When mask bit 5 is low, operation of the MPC clock is inhibited.
6	Interrupt 5 Enable/Inhibit (1) (0)	Mask bit 6 is used to enable or inhibit the generation of interrupt 5 CCL whenever the MPC clock is stopped. If mask bit 6 is low, interrupt 5 is inhibited and consequently, not serviced when the MPCclock stops. If mask bit 6 is high, interrupt 5 is enabled for servicing when the MPC clock stops.
7	Address Jamming Control Clock/ DDT29J (1) (0)	Mask bit 7 controls the address jamming function. if mask bit 7 is high, the contents of the CCL address register are jammed onto the WCS address bus whenever the System Clock signal goes high. If mask bit 7 is low, the CCL address register contents are jammed onto the WCS address bus whenever the DDT29 jam subcommand is invoked.

bus by invoking the DDT29 jam subcommand. The example given could be useful for setting address breakpoints to examine the prototype microcode and hardware operation at significant points.

	ADDR. JAM	INTR.5	M.S. CLOCK	ADDR. COMP.		TRAP BITS		
MASK BIT	7	6	5	4	3	2	1	0
ENTERED AT CRT CONSOLE	6				F			
BINARY VALUE LOADED INTO CONTROL REG.	0	1	1	0	1	1	1	1

Figure 4-5. Clock Control Logic Control Register Mask Bit Example

ADDRESS REGISTER STORE

The address register store subcommand stores a specified address into the clock control logic address register. The execution format is as follows:

IR X

Where:

X indicates the address value (000-FFF).

JAM ADDRESS

The jam address subcommand jams the address stored in the CCL address register onto the WCS card address bus. The execution format is as follows:

Jam

The user must consider the type of microinstruction located at the jammed address and take the steps shown in table 4-5. This is because the jam address bypasses the sequencer and accesses WCS directly. The effect is that the jam address is forced to the address inputs of the WCS for one microcycle and then cleared. The microprogrammed architecture then executes whatever microinstruction was pointed at by the jam address. The PC held by the sequencer is not changed unless the microinstruction at the jam address changes it.

TABLE 4-5. JAM ADDRESS MICROINSTRUCTIONS STEPS

TYPE OF MICROINSTRUCTION*	USER STEPS**
Stack Operation (Push/Pop)	Push---Note that the address being pushed is the PC address. PC may have to be reset. Pop---User must verify that the popped address corresponds to the intended pushed address.
Jump-to-Pipeline Operations (From D Input)	No limitations. (The jammed instruction sets the PC and next address.)
Continue (i.e., LDCT, CONT)	Not allowed. Instruction must be changed to one that updates the PC (e.g. JUMP) and next address.
External Inputs (OR inputs)	Since OR inputs only affect the least significant 4 bits (with AmSYS29 MSC), the user must reset/set the most significant 8 bits of address in the next address and PC.
Conditional Instruction	Allowed except if a continue operation results. (See continue above.)

* Refers to Am29811 next address control.
** Assumes AmSYS29 MSC is used for microprogram control.

SLEEP

The sleep subcommand permits the user to stop DDT29 subcommand processing in a variety of ways:

```
Z  n
Z  SP
Z  TGL
```

n operand---n specifies the number (1-65535) of milliseconds that DDT29 subcommand processing will be stopped. Default value for n is 1.

SP operand---Each time a Z SP subcommand is encountered, DDT29 subcommand processing will stop. Pressing the space bar on the System 29 console will cause DDT29 subcommand processing to resume. The user may optionally enter a numerical value n (1-65535) followed by the space-bar. Subsequent subcommand processing will halt on every nth Z SP subcommand.

TGL operand---The TGL operand of the Z subcommand will cause DDT29 subcommand processing to stop the first time it is encountered. Pressing the space-bar on the AmSYS29 console will cause DDT29 subcommand processing to resume. The user may optionally enter a numerical value n (1-65535) followed by the space-bar. Subsequent subcommand processing will halt on every nth Z SP subcommand.

TGL operand---The Z subcommand will cause DDT29 subcommand processing to stop the first time it is encountered. Pressing the space-bar on the AmSYS29 console will cause DDT29 subcommand processing to resume. Now, the Z TGL subcommand will be ignored until the console space-bar is pressed again. TGL, therefore, is a toggle which, on each depression of the console space-bar, will alternate between stopping or not-stopping on the Z.

DISPLAY TRACE

The DTR subcommand displays the address of the last microinstruction executed (see DLA subcommand), the contents of WCS at that address, and the monitor bits (see DMB subcommand). By using DDT29's macro facility, the user may be use the DTR subcommand to trace microprogram execution. An example follows:

```
DTR M Z SP MS DTR  cr
```

Each time the console space-bar is pressed, the microprogram will advance to the next microinstruction, display DTR information as described above, and stop. Replacing the SP operand with a numerical value will cause the microprogram to automatically slow-step at any rate between 1 and 65535 milliseconds.

MACRO

The macro subcommand permits the user to repeatedly execute a string of DDT29 subcommands. The string of subcommands can be executed as many times as desired through the use of the n operand as described below. The DDT29 subcommands repeatedly executed are contained between the M subcommand and the MEnd term or the carriage return cr terminating the command line. The execution format for the macro subcommand is as follows:

```
M n DDT-29-subcommand,.....DDT-29-subcommand, MEnd  cr
```

Where:

n specifies the number of times the string of DDT29 subcommands is to be executed. Default value for n is infinity. The subcommands following the MEnd term are executed when the n count is completed. Should the n count not be entered (infinit loop), a DElete from the CRT console terminates the macro loop. Any DDT29 subcommands entered after the MEnd term are then executed.

NOTE

Execution of the macro subcommand can be terminated at any time by pressing the CRT console DEL key.

DISPLAY LAST ADDRESS

The display last address subcommand permits the user to display the contents of clock control logic register that contains the address of the last microinstruction executed. The address is displayed in hexadecimal at the CRT console. The execution format for the display last address subcommand is as follows:

DLA

DISPLAY MONITOR BITS

The display monitor bits subcommand permits the user to display the 20-bit contents of the clock control logic monitor registers. The monitor registers contain monitor bits such as test points or other data that are user selected and then, user wired to the instrumentation card. The 20 monitor bits are displayed as five hexadecimal characters at the CRT console display and the monitor bits. Figure 4-6 shows the relationship between a typical CRT console display and the monitor bits. The execution format for the display monitor bits subcommand is as follows:

DMB

EXIT

The exit subcommand causes AmSYS29 to leave the DDT29 environment and return to AMDOS29 control.

Exit

HEX DISPLAY AT CRT CONSOLE														
BINARY EQUIVALENT	MSB 0	1	1	1	1	0	0	0	0	0	0	0	1	0
MONITOR BIT NUMBER	19	18	17	16	15	14	13	12	11	10	9	8	7	6
CCL CARD P4 PIN ASSIGNMENT	40	38	36	34	32	30	28	26	24	22	20	18	16	14

Figure 4-6. Typical Monitor Bit Display and Clock Control Logic Card Pin Assignment

CHAPTER 5

POST PROCESSING ROUTINES

INTRODUCTION

The AMDASM meta-assembler has three post processing routines: AMSCRM for engineering applications, AMPROM to produce an object code tape and AMMAP to generate non-microinstruction PROM data.

AMSCRM DESCRIPTION

AMSCRM bit scrambling post processor reassigns the bit positions of the microword contents by simply specifying the source and destination bit positions and the length of each field to be moved. In so doing, a reorganized microcode object file is produced. At the conclusion of an AMDASM assembly, the object code generated by AMDASM is the input to AMSCRM. The leftmost bit in the object code is assumed to be position 0; thus the rightmost bit position will be microword size-1.

After execution begins, the transformation parameters are entered. These indicate the source bits to be moved, their destinations and the length of the field to be moved.

After execution of AMSCRM the microcode is in its new bit order and is available on a file to be used as input to AMPROM.

AMSCRM is used by the microprogrammer to assign microword fields that differ from those in the actual hardware implementation. This is important when the programmer allocates bits according to functions and needs to translate the object code produced by AMDASM to be consistent with the hardware microprogram memory design.

The ability to shift bit assignments is important to engineers. As a product evolves, bits may be added or deleted from the original microword format. At the time that PROMs need to be blown, bits often need to be reassigned to be consistent with the hardware implementation.

EXECUTION AND FILENAMES FOR AMSCRM

After the AMDOS29 operating system has issued a user prompt (e.g., the characters A>), AMSCRM is executed by entering either of the following commands:

A > AMSCRM␣OLD=filename1␣NEW=filename2 cr

A > AMSCRM␣OLD␣filename1␣NEW␣filename2 cr

Filename1 is the name given to the file containing the microcode generated by AMDASM and is assigned name qqqqqqqq.OBJ if AMDASM was executed without specifying OBJECT=filename.

Filename2 is a user-defined name for the file on which the reordered microcode is to be placed. It is recommended that the user make the primary part of Filename2 the same as Filename1, but that he use a different generic. Filename2 must be different from Filename1. There are no required generics for AMSCRM, but if Filename1 does not specify a generic, the generic defaults to .OBJ. Likewise, the default generic for Filename2 is .XOB.

After the execution command and a carriage return is entered, AMSCRM issues a prompt:

ENTER TRANSFORMATION PARAMETERS:

The user enters:

S0, D0, W0, cr
S1, D1, W1, cr
Sn, Dn, Wn, cr
. cr

where:

S0 = starting (leftmost) bit position for the first source field to be moved.

D0 = destination bit position for the first (leftmost) bit of the first group of bits.

W0 = width of the field to be moved.

S1 = starting (leftmost) bit position for second source field to be moved.

.
.
.

Wn = width of the last field to be moved.

Each group of parameters is ended by a carriage return. A period and a carriage return are used to terminate input. For all microwords the leftmost bit position of the AMDASM printout is considered to be zero; thus the rightmost bit position will be the width of the microword -1.

It is the user's responsibility to see that all bits are properly shifted. Thus, if the user enters 13,28,4 cr. Indicating that 4 bits beginning at bit position 14 are to be moved to bit positions 28,29,30,31; 28,X,4 cr must also be entered. X indicates the new starting bit position for the bits originally in positions 28-31.

AMSCRM EXAMPLE

The MUX control bits in the Evaluation Kit are physically separated in the hardware configuration. However, it would be much more convenient to program them as contiguous bits when writing the microcode.

The bit numbers shown in figure 3-4 are numbered right to left; AMDASM and AMSCRM count bit positions from left to right. Thus, if the MUX control bits were assigned to the bit positions 8 and 9 during AMDASM, then AMSCRM would require the following command to put them into the positions shown in Figure 5-1. The AMDASM output is assumed to be on the file SYSTEM1.OBJ. SYSTEM1.XOB is the name to be assigned to the AMSCRM output as follows:

```
A > AMSCRM↓OLD=SYSTEM1↓NEW=SYSTEM1  cr

ENTER TRANSFORMATION PARAMETERS:

    9,12,1  cr
    10,9,3  cr
    .  cr
```

```
Bit No.    0  1  2  3  4.....................................47
-------->

Executable    1
Instruction   2
Number        3
              4
              .
              .
              .
              1023
```

Figure 5-1. Bit Matrix

AMPROM DESCRIPTION

When an AMDASM assembly and an optional AMSCRM execution is complete, AMPROM is used to output binary object code in a form that corresponds with the PROM's organization and/or to be used as input to a PROM burner.

PROM ORGANIZATION

In order to understand post processing one must know how the PROMs are organized in the computer memory space.

If AMDASM has been executed using the command

A > AMDASM␣P1=2900␣P2=2900 cr

AMDASM generates binary object code for the executable statements in the file names 2900.SRC. This binary object code is output to a file called 9000.OBJ. Assume that the microword is 48 bits wide and the number of executable statements is 1024.

This gives us a matrix 48 wide by 1024 deep as shown in figure 5-1.

After PROM width and depth are specified, the bit matrix is subdivided to yield a PROM map where each PROM is n bits wide by m bits deep. If we assume that the initial program counter is zero the actual PROM map printed appears as shown in figure 5-2.

	PC	C1	C2	C3	C4	C5	C6	C7	
R1	0000	1	2	3	4	5	6	7	
R2	0100	8	9	10	11	12	13	14	PROM
R3	0300	15	16	17	18	19	20	21	No.
R4	0380	22	23	24	25	26	27	28	

where:

Pc represents the initial program counter value for that PROM row. The PC value is given in hexadecimal.

Figure 5-2. Sample Printout of a PROM MAP

Physical size and organization are as shown in figure 5-3.

Each executable instruction has a program counter associated with it by virtue of its position in the program and/or the origin(s) that were set during the assembly execution.

This breakup of the matrix is now called a PROM map that has associated with it, not only the PROMs shown, but also rows and columns as shown in figure 5-3. Thus, we may now refer to PROM 19 by using the digits 19, or by referring to R3 for row 3 or C5 for column 5.

All PROMs in row 1 are 256 (instructions) deep. PROMs 1, 3, 5, and 6 are only 4 bits wide, while PROMs 2 and 7 are 8 bits wide and PROM 4 is 16 bits wide.

In row 2, all PROMs are 512 (instructions) deep; PROMs 15, 22, 17, 24, 19, 26, 20 and 27 are 4 bits wide; PROMs 16, 23, 21, 28 are 8 bits wide; and PROMs 18 and 25 are 16 bits wide.

A printing request (or punching) for PROM #1 will obtain data that is 4 by 256.

A printing request for row 3, will obtain data (i.e., the contents of PROMs 15 through 21) in the following form:

4 x 128, 8 x 128, 16 x 128, 4 x 128, 4 x 128, 8 x 128

A printing request for column 4 will obtain data (i.e., the contents of PROMs 4, 11, 18, and 25) that is as follows:

16 x 256, 16 x 512, 16 x 128, 16 x 128

POST PROCESSING FEATURES

AMPROM allows the user to specify the depth (number of instructions) and width (bits of the microword) for each PROM. Listing or suppression of listing of the PROM map is available by PROM number and listing of PROM content by PROM rows or PROM columns or by PROM number. Optional automatic inversion of all bits except the don't care bits and specification of don't care bits to be 0 or 1 is available.

EXECUTION COMMAND FOR AMPROM

The format for AMPROM execution command is:

A> AMPROM80=qqqqqqqq.ggg[options] cr

The primary part of the object code filename must be typed. If the generic part is not specified, the default .OBJ is assumed.

Options and their default values are shown in table 5-1.

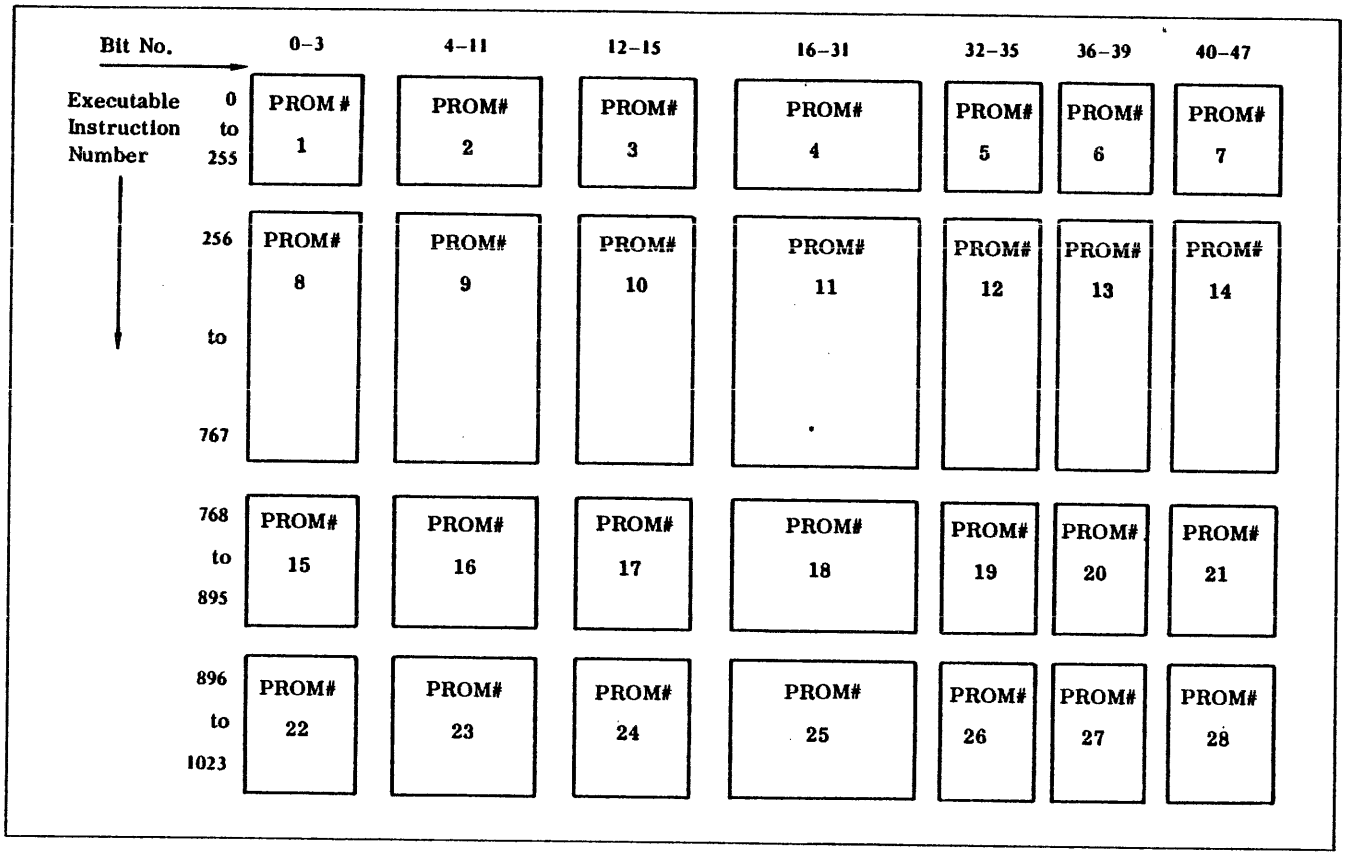

Figure 5-3. Physical Organization of PROMs

TABLE 5-1. AMPROM 29 OPTIONS

OPTION	ABBREVIATED OPTION	DEFAULT	MEANING
OBJECT⌀filename1 or OBJECT=filename1	O	NONE This is a required input.	Specifies the name of the file on which the AMDASM object code is located. If only the primary part of filename1 is input, the default generic .OBJ is assumed.
MAP	M	MAP	Print the PROM map.
NOMAP	NM	MAP	Suppress printing the PROM map. If NOMAP is not specified, the program automatically prints the PROM map.
HEX	H	HEX	Punch the PROM output hexadecimal format.
BNPF	B	HEX	Punch the PROM output in BNPF format. If BNPF is not specified, the output is automatically punched in hexadecimal.
INVERT	I	No inversion	If INVERT is specified, all ones are inverted to zeros, and zeros to ones, except for bits specified as don't cares. If INVERT is not specified there is no modification to the binary object code.
PUNCH⌀filename2 or PUNCH=filename2	P	filename1.OUT	Specifies the name of the file or device where punch data is to be output. If not specified there is no modification to the binary object code.
NOPUNCH	NP	filename1.OUT	Suppresses punching the PROM contents. If not specified, defaults to PUNCH.
LIST⌀filename3 or LIST=filename3	L	filename1.P3L	Specifies the name of the output file device where the AMPROM output listing is to be placed. If not specified, the output automatically goes to the default file named filename1,P3L.
NOLIST	NL	filename1.P3L	Specifies that the output is not to be listed. This would be used when only punching of the output is desired. If not specfied the program defaults to LIST using the default file named filename1.P3L.

AMPROM FILENAMES

As part of the options, the user may need to specify filename information. Whether filename information is needed will depend on whether or not the user wishes to receive his output at a printer console or punched on paper tape or stored on files with or without default assignments.

The PUNCH␣filename and LIST␣filename must each be preceded by a blank and may be specified in any order. The filename may be any AMDOS 29 device.

If the user executed AMDASM with the following command:

A > AMDASM␣P1␣2900␣P2␣2900 cr

The binary object code is stored on a file called 2900.OBJ. When executing AMPROM, only 2900 must be given as the input filename.

Thus the command to execute AMPROM is A > AMPROM␣O␣2900 cr and, since no LIST or PUNCH is specified, all output will be to the default filenames 2900.OUT and 2900.P3L.

AMPROM EXECUTION EXAMPLES

The command A > AMPROM␣NOLIST␣PUNCH␣PUN:␣OBJECT␣2900 cr
specifies the PROM map and the PROM content are to be printed on the list device, the content of the PROMs is not to be punched, but will be stored in hexadecimal on the file with the default name qqqqqqqq.OUT.

However, A > AMPROM␣O=qqqqqqqq.ggg␣NOLIST␣NOMAP␣PUNCH=PUN: cr
specifies that the content of the PROMs is to be punched on the paper tape punch with no listing of the PROM map or PROM content.

Each option is preceded by a required blank and options may be given in any order. The full option name or the abbreviated option name can be used. If filename1 has no generic specified, it defaults to .OBJ. If filename2 (PUNCH) or filename3 (LIST) is input without a generic, AMPROM assumes no generic, and uses exactly what was input.

INTERACTIVE AMPROM INPUT

Once AMPROM has begun execution, the user acts interactively with the console. Messages from the console will be received and responses can be input followed by a carriage return. The terminal prints the requested output and sends a message requesting additional input. When execution is complete, control is returned to AMDOS29.

Sample console messages are given below. For this example, underlined numbers are used to illustrate the user's input. Table 5-2 is a list of the acceptable substitutes that may be used for the underlined values. After the user has input an AMPROM execution command, the terminal responds by displaying these messages:

```
DON'T CARES? 1 cr
ENTER PROM WIDTHS 4 * 8, 4 cr
ENTER PROM DEPTHS 128 cr
```

If a map listing at the output device is requested the PROM map is output here. Then the console displays the message,

```
WHICH PROMS DO YOU WISH TO PRINT? 5-7 cr
```

If printing of the PROM content was specified, the PROM content is printed. A tape of the contents will be punched unless NOPUNCH was specified. The punch device should be turned on before keying in the PROMs to be printed and punched.

When execution is complete, control is returned to AMDOS29.

BNPF PAPER TAPE OPTION

When BNPF is specified as an option, the output is in the BNPF format. B is punched as the first character, then a P (for a one) or an N (for a zero) is punched as the last character for this row of PROM data. This continues until all rows (the depth) of the PROM are punched.

Before the first BNPF for each PROM is punched, the program punches identification on the tape that consists of 32 rubouts, 4 ASCII characters that are the PROM number and 32 NULs to be used as the leader when loading the PROM burner tape reader. After the PROM data is punched, 40 NULs are punched to facilitate tape handling.

If PROM #5 is 4 bits wide by 128 bits deep, and begins at origin zero, the paper tape will appear as shown in table 5-3.

HEXADECIMAL PAPER TAPE OPTION

When punching is desired, and HEX is specified or assumed by default, the PROM contents are punched in the DATA I/O hexadecimal format.

The same initial data (32 rubouts, PROM number and 32 NULs) is punched as is punched for the BNPF format, followed by the PROM content in hexadecimal.

TABLE 5-2. AMPROM INPUT SUBSTITUTES

CONSOLE PROMPT	SUBSTITUTES	MEANING
DON'CARES?	0 or 1	The value specified here is assigned to all don't care bits in the PROM(s). Any value except 0 or 1 is an error and the prompt is repeated.
ENTER PROM WIDTHS	n	n is a decimal integer and each PROM is n bits wide. If the microword size is 60 and n is given as 8, 8 PROMs will be generated. The first seven will contain actual microword information but the 8th PROM will contain microword information in its leftmost 4 bits and don't cares in the 4 right-hand bits (i.e., if the microword width is not an even multiple of n, it is padded on the right with don't cares).
	i*b	i is a decimal integer indicating a number of PROMs. b is a decimal integer indicating the number of bits wide for each of these PROMS. Thus, 3 * 4 means there are 3 PROMS each 4 bits wide.
	Combinations of n and i*b	For the PROM map (figure 5-2), the user would write 4, 8, 4, 16, 2*4, 8. Any combination of n and i*b is permissible if separated by commas and if the total number of bits is less than or equal to the microword width.
ENTER PROM DEPTHS	r	r is a decimal integer and each PROM is r instructions deep (long). If the binary object code is not an even multiple of r, AMPROM fills the final PROM locations with don't cares.
	t*d	t is a decimal integer indicating a number of PROMs. d is a decimal integer indicating how many words deep each of these PROMs is to be. Thus 2*512 indicates there are 2 PROMs each 512 bits deep.
	Combinations of r and t*d	For the PROM map in figure 5-2, the user would write 256, 512, 2*128. Any combination of r and t*d is permissible if separated by commas.
WHICH PROMS DO YOU WISH TO PRINT***	Y	Y is a decimal integer which is a PROM number. 5 means list the contents of PROM #5.

*** The same PROMS are printed and/or punched. Thus, all values for printing apply for punching also.

TABLE 5-2. AMPROM INPUT SUBSTITUTES (continued)

CONSOLE PROMPT	SUBSTITUTES	MEANING
WHICH PROMS DO YOU WISH TO PRINT*** (continued)	Y_1-Y_n	Y_1 is a decimal integer specifying the number of the first PROM to be listed. Y_n is a decimal integer specifying the last PROM to be listed. Thus, 2-5 specifies listing of PROMs 2, 3, 4, and 5.
	Combinations of r r and t*d	3, 5-7, 9 means print (and punch) PROMs 3, 5, 6 and 9. All combinations of Y and Y_1-Y_n are acceptable if separated by commas.
	Cs	C means column and s is a decimal integer which specifies the PROM column desired. C4 means print all PROMs in column 4.
	Cs_1-s_n	Print columns s_1 through s_n. C1-6 indicates print PROM columns 1 through 6.
	Combinations of Cs, s_1-s_n	C5, 7-9, 11 means print columns 5, 7, 8, 9, 11. C3-6, 10 means print columns 3, 4, 6, 10 (i.e., C is only given once, then the s and/or s_1-s_n separated by commas).
	Rs	R means row and s is a decimal integer which specifies the row desired. R1 means print all PROMs in row 1.
	Rs_1-s_n	List the contents of PROM rows 1_1, through s_n. R2-6 means print all PROMs in rows 2 through row 6.
	Combinations of Rs, s_1-s_n	The same as columns. The R is given once, followed by the row numbers separated by commas. R1, 4-6, 11-13 prints rows 1, 4, 5, 6, 11, 12, 13.
	N	The letter N is typed if the user wishes to indicate none of the PROM contents are to be listed.
	A	The letter A when typed means all PROMs are to be printed.

*** The same PROMS are printed and/or punched. Thus, all values for printing apply for punching also.

For PROMs 4 or less bits wide, one hexadecimal character and a space is punched. For PROMs greater than 4 bits wide, two hexadecimal characters and a space are punched. Thus, two characters, space, two characters, space would be punched for either 2 rosw of an 8-bit PROM, or for 1 row of a 16-bit wide PROM.

Thus if PROM #7 (16 bits x 128 words) is punched, the output will appear as shown in table 5-4.

TABLE 5-3. BNPF PAPER TAPE CONTENTS

TAPE CONTENTS	CONTENT EXPLANATION
$Rubout_1$ ⋮ $Rubout_{32}$	32 Rubouts
Characters 0005	PROM number
NUL_1 ⋮ NUL_{32}	32 NULs
Character B	
Character N or P Character N or P	BPNF format for one row of this 4-bit wide
Character N or P	PROM
Character F	
Space	See Note
Character B	
Character N or P ⋮ NUL_1	Repeated 127 times
⋮ NUL_{40}	40 training NULs

NOTE: Carriage return/line feed for possible listings is inserted after 8 words for PROMs4 or less bits wide, after 43 words for widths of 16 or less bits, and after one word for width greater than 16.

TABLE 5-4. HEXADECIMAL PAPER TAPE CONTENTS

TAPE CONTENTS	CONTENT EXPLANATION
$Rubout_1$ ⋮ $Rubout_{32}$	32 Rubouts
Characters 0007	PROM number
NUL_1 ⋮ NUL_{32}	32 NULs
SOH Character Character Space Character Character Space Character Character	Contents of PROM Row 1 (4 HEX digits) Repeated 127 Times *See Note
⋮ EXT	End of Text
NUL_1 ⋮ NUL_{40}	40 NULs

NOTE: Carriage return/line feed for possible listings is inserted after 16 groups of hexadecimal characters.

EXAMPLE OF AMPROM

Figure 5-4 is an example of AMPROM for the Am2900 Learning and Evaluation Kit.

```
CONSOLE INPUT
DON'T CARES?0
ENTER PROM WIDTH?8
ENTER PROM DEPTH?16
WHICH PROMS DO YOU WISH TO PRINT?3-4

AMPROM OUTPUT

AMD AMPROM UTILITY
AM2900 KIT EXERCISE 10B
PROM MAP
     PC    C1    C2    C3    C4
R1  0000    1     2     3     4
PROM CONTENTS
PC   ADD P 3      P 4
0000 000 00110000 00001111
0001 001 00110000 00011001
0002 002 00110000 00100000
0003 003 00110000 01000100
0004 004 01000000 00110000
0005 005 01000000 00000001
0006 006 00110000 00000000
0007 007 01000001 00010001
0008 008 00110000 00010000
0009 009 01000010 00100001
000A 00A 00110000 00100000
000B 00B 00010000 01000000
000C 00C 00000000 00000000
000D 00D 00000000 00000000
000E 00E 10000000 00110000
000F 00F 00110000 00110000

PUNCH OUTPUT
   3
BNNPPNNNNF BNNPPNNNNF BNNPPNNNNF BNNPPNNNNF
BNPNNNNNNF BNPNNNNNNF BNNPPNNNNF BNPNNNNNPF
BNNPPNNNNF BNPNNNNPNF BNNPPNNNNF BNNNPNNNNF
BNNNNNNNNF BNNNNNNNNF BPNNNNNNNF BNNPPNNNNF
   4
BNNNNPPPPF BNNNPPNNPF BNNPNNNNNF BNPNNNPNNF
BNNPPNNNNF BNNNNNNNPF BNNNNNNNNF BNNNPNNNPF
BNNNPNNNNF BNNPNNNNPF BNNPNNNNNF BNPNNNNNNF
BNNNNNNNNF BNNNNNNNNF BNNPPNNNNF BNNPPNNNNF
```

Figure 5-4. AMPROM Output for Am2900 Learning and Evaluation Kit.

AMMAP DESCRIPTION

The AMMAP mapping RAM/PROM data assembler enables AmSYS29 to generate non-microinstruction PROM data for the mapping RAM in the microprogram sequencer card.

AMDASM outputs a symbol table file of microprogram entry point symbols as an option with the generic file name MAP. The AMMAP assembler uses this file, in conjunction with an assembly source file provided by the user, as a symbol table to generate an object file. The object file uses the generic file name OBM, and is compatible with the AMDASM object file format. Therefore, it can be loaded/verified by the LBPM and VBPM programs.

AMMAP is a one-pass assembler that allows the user to specify the width of the mapping PROM, the assembler's location counter value, and the microprogram entry point addresses to be assembled into any PROM location.

AMMAP MAJOR FUNCTIONS

The principal function of the AMMAP assembler is to generate mapping PROM data through a symbolic source program. When AMMAP is called for execution, the user must specify the map file to be used for symbol table input. AMMAP builds a symbol file from this file and begins assembly of PROM data.

The individual functions of AMMAP are as follows:

a. Entry point symbol table management - AMMAP will manage and utilize the entry point symbol table built from the user specified MAP file.

b. Location counter control - AMMAP starts assembly at PROM location 0 unless specified otherwise via user directives that set the location counter value. In addition, it keeps track of locations and assigns locations for each entry point address assembled.

c. Data assembly - translates symbolic entry point addresses into internal binary equivalents and assembles them into PROM location.

d. Assembly directive processing - processes all assembly directives: PROM width specification, number base specification for setting location counter, assembly listing, and object output control, and directive.

e. Assembler output generation - generates an assembly listing, object data output file, and error diagnostics.

f. User command language interface - processes user-specified assembler execution parameters and other user interfaces.

AMMAP PERFORMANCE CHARACTERISTICS

AMMAP runs under the 64K memory configuration for AmSYS29. It allows at least 40K for entry point symbol table space and can handle more than 3000 entry point symbols.

USER INTERFACE

PROGRAM SOURCE STATEMENT CONCEPTS

The format of an assembly statement in AMMAP is:

location: entry0, entry1,. . . .,entryn

where:

location is a binary, octal, decimal, or hex constant. The number base is selectable via the BASE directive and default base is hexadecimal. location and colon following it are optional. If not present, AMMAP assigns the next available location. Assembly origin is 0, unless specified otherwise.

entry(n) is an entry point symbol that is defined during AMDASM assembly phase and entered into the symbol table written out as the MAP file. It may also be an absolute address in which case it must be a constant which follows AMDASM syntax rules.

COMMENT STATEMENTS

A comment may be introduced into any source line by preceding the comment with a semi-colon (;). AMMAP will treat all source text on a line after a semi-colon as a comment up to the carriage return.

ASSEMBLER DIRECTIVES

There are four assembler directives used by AMMAP: WIDTH, TITLE, BASE and END.

WIDTH (PROM WIDTH) DIRECTIVE

The format for the WIDTH directive is:

WIDTH n

where: n is a decimal constant (which specified the width of mapping PROM or RAM 1_n_128).

TITLE DIRECTIVE

The format for the TITLE directive is:

```
TITLE  text
```

where: text is a title string of up to 60 characters.

The title will appear in the page header of assembly listings and the title record for object file.

BASE (LOCATION COUNTER BASE) DIRECTIVE

The format for the BASE directive is:

```
BASE type
```

where: type may be one of the following: 2, 8, 10, or 16 to designate that binary octal, decimal, or hex numbers will be used for specifying PROM location.

If a number base is not specified in the program, the default used is 16 (hexadecimal).

END (END OF PROGRAM) DIRECTIVE

The format for the END directive is:

```
END
```

The END directive must be used to terminate the AMMAP assembly source input file. Use of TAB characters are also allowed in AMDASM.

COMMAND LANGUAGE

The AMMAP assembler may be executed with the following AMDOS29 transient command:

```
AMMAP   filename1  MAP  = filename2  options  cr
```

where:

filename1 is the primary filename of the AMMAP source input file which must have the generic file name .OPC.

filename2 is the primary filename of the .MAP output file from AMDASM to be used as the entry point symbol table.

options are user selectable options described in table 5-5.

TABLE 5-5. AMMAP OPTIONS

OPTION	ABBREVIATED OPTION	DEFAULT	MEANING
LIST⍽filename or LIST=filename	L		Specifies the listing is to be output to a file with the name (filename). If not given the listing is placed on a file with the default name pppppp.P4L.
NOLIST	NL	pppppp.P4L	Suppresses the creation of a listing. If not specified defaults to L=pppppp.P4L.
OBJECT⍽filename or OBJECT=filename	O		Specifies that the microcode (object code) is to be output on a file with the name (filename). If not given, the microcode is placed on a file with the default name qqqqqq.OBM.
NOOBJECT	NO	qqqqqq.OBM	Suppresses placement of the microcode onto a file. If block format printing is requested, the object code printing is also suppressed. If not specified defaults to OBJECT and the microcode goes to file qqqqqq.OBJ.
WIDTH⍽n or WIDTH=n	W	n=80	Specifies width of n (a decimal number) characters for listing devices. Default is 80.
LINES⍽n or LINES=n	LN	n=66	Specifies length of n, (a decimal number) lines per page. If no specified, default is 66 lines (11 inches).
HEX	H		Specifies listing of location counter in hexadecimal format.
OCTAL	O	HEX	Specifies listing of location counter in octal format. If no specified defaults to HEX.
SYMBOL	S	SYMBOL	Specifies listing of constant names and labels and their associated values.
NOSYMBOL	NS		Suppresses listing of symbol table. If not specified, defaults to SYMBOL.

AMMAP ERROR MESSAGES

Table 5-6 is a list of AMMAP error messages and their definitions.

TABLE 5-6. AMMAP ERROR MESSAGES

ERROR	MEANING
ERROR 1	Illegal character
ERROR 2	Undefined symbol
ERROR 3	Illegal location counter value
ERROR 4	Missing colon after location Counter value
ERROR 5	Missing delimiter after PROM Data specification
ERROR 6	Missing end statement
FATAL ERRORS:	
ERROR 100	Command option syntax error
ERROR 101	Illegal mapping PROM width specification

PFORMAT DESCRIPTION

The PROM Programmer subsystem provides the software routines that reformat the microinstruction fields and output the microcode to the PROM Programmer. PFORMAT.COM converts an AMPROM output file (filename.OUT) to a DATA I/O format file (filename.DIO). PPROG.COM interfaces .DIO format files to the PROM Programmer via a set of subsystem commands; PLPROG.COM interfaces to a Prolog PROM programmer.

The PFORMAT command converts an AMPROM output file to a PROM Programmer input file. Each PROM defined on the AMPROM output file is defined by PROM number, on the PROM Programmer .DIO input file. The format of the PFORMAT command is:

PFORMAT filename1 (.filetype)filename2(.filetype)

filename1 is the name of the AMPROM output file; its filetype is optional and will default to .OUT if omitted.

filename2 identifies the DATA I/O format file; it is optional. When filename2 is not specified, it will default to filename1. The filetype for filename2 is also optional; it will default to .DIO if omitted.

A space is required to delimit PFORMAT from filename1 and delimit filename1 from filename2.

For information on using PPROG and PLPROG, refer to the AMDOS 29 User's Manual.

CHAPTER 6

ERROR MESSAGES AND INTERPRETATIONS

AMDASM ERRORS

Each source file input statement is processed until a single error is detected. One missing comma between fields, for example, would result in incorrect processing of the remainder of the statement.

Thus, the assembler stops when an error is encountered, records the error, the statement that caused it, and processes subsequent source input statements.

Console error messages without an error number are AMDOS29 error messages.

AMDASM and AMPROM error message format is as follows:

```
***ERROR n {y}
```

where:

n = error number

y = illegal character or symbol.

Fatal error messages appear on the console output device as well as on the assembly file.

Seemingly inappropriate error messages occur because the assembler is unable to determine the programmer's intent. This is often the result of a missing comma, semicolon, blank, or colon. Table 6-1 is a list of AMDASM errors.

ERROR 1: ILLEGAL CHARACTER

The character which cannot be interpreted is printed and the line in which it occurs is also printed. This message may be generated by:

- Striking the wrong console key.
- A missing comma or semicolon (B#101Q#7 is not interpretable).
- A wrong number base used (N#3 or Q#8 cannot be interpreted).

TABLE 6-1. AMDASM ERRORS

ERROR 1	ILLEGAL CHARACTER
ERROR 2	UNDEFINED SYMBOL
ERROR 3	UNDEFINED FORMAT
ERROR 4	DUPLICATE FORMAT
ERROR 5	DUPLICATE LABEL
ERROR 6	DUPLICATE SUBDEFINE
ERROR 7	FORMAT FIELD OVERFLOW
ERROR 8	SUBDEFINE FIELD OVERFLOW
ERROR 9	UNDEFINED DIRECTIVE
ERROR 10	ILLEGAL MICROWORD LENGTH
ERROR 11	ILLEGAL FIELD LENGTH
ERROR 12	DON'T CARE FIELD TOO LONG
ERROR 13	ARITHMETIC OPERATION ON FIXED FIELD
ERROR 14	ATTRIBUTE ERROR
ERROR 15	(Not Used)
ERROR 16	MISSING END STATEMENT
ERROR 17	ILLEGAL SYMBOL
ERROR 18	OVERLAY ERROR
ERROR 19	NO DEFAULT VALUE
ERROR 20	FIELD LENGTH CONFLICT
ERROR 21	$ SPECIFIED FOR NON-ADDRESS FIELD
ERROR 22	(Not Used)
ERROR 23	MISSING DESIGNATORS
ERROR 24	SPACE DIRECTIVE ERROR
ERROR 25	ORG SET TO LESS THAN CURRENT PC
ERROR 26	NO FORMAT NAME AFTER &
ERROR 27	(Not Used)
ERROR 28	ADDRESS NOT IN CURRENT PAGE
ERROR 29	LENGTH REQUIRED FOR $ MODIFIER
ERROR 30	ILLEGAL FIELD LENGTH IN FF STMT
ERROR 31	(Not used)
ERROR 32	NO EXPLICIT LENGTH BEFORE (

ERROR 2: UNDEFINED SYMBOL

This message will most often occur when:

- Something is misspelled.

```
HERE:  EQU␣100
GO TO:DEF␣HEER (the assembler cannot find HEER)
```

- The # is missing after a B, Q, D, or H.
- The space is missing after definition words DEF, EQU, SUB, WORD, TITLE, RES, ORG, ALIGN, FF, SPACE.
- A symbol is referenced before it is defined by a SUB or and EQU.
- A VFS for a hexadecimal field begins with the letters A through F and the H# designator does not precede the letter.

ERROR 3: UNDEFINED FORMAT

The format name given is misspelled or was not defined in the definition phase or the required blank was not supplied after the format name.

ERROR 4: DUPLICATE FORMAT

The name given before a format (DEF) has already been used as a name. If names contain more than 8 characters, the first 8 must be unique. Check for misspelled names.

ERROR 5: DUPLICATE LABEL

This label has been used more than once as a constant name or a label. If the label is more than 8 characters, the first 8 must be unique.

ERROR 6: DUPLICATE SUBDEFINE

The name given preceding a subformat (SUB) has already been used as a name. If names contain more than 8 characters, the first 8 must be unique. Check for misspelled names.

ERROR 7: FORMAT FIELD OVERFLOW

The user is permitted a maximum of 128 fields per format name (DEF). This number has been exceeded. The format must be revised and fields must be combined.

ERROR 8: SUBDEFINE FIELD OVERFLOW

The user is permitted a maximum of 128 fields per subformat name (SUB). This number has been exceeded. Revise the sub format and combine fields or use two subformats for this bit pattern.

ERROR 9: UNDEFINED DIRECTIVE

No name: was found and the characters given are not TITLE, WORD, LIST, NOLIST, END, ORG, RES, SPACE, or ALIGN.

Check for a missing colon after a name, or misspelling, or blanks in TITLE, WORD, etc.

ERROR 10: ILLEGAL MICROWORD LENGTH

Each time DEF or FF is encountered, the assembler checks to see if the sum of the bits for all fields for this format name exactly equals the microword length.

Thus, the user is assured that each DEF or FF contains an exact number of bits. If the number of bits in this format does not exactly equal the number of bits given with WORD, the interpretation of the faulty DEF or FF is bypassed and the assembler attempts interpretation of the next source statement.

ERROR 11: ILLEGAL FIELD LENGTH

No field, except a don't care field, may be more than 1 bits in length. The value calculated for this field cannot be represented in 16 bits.

ERROR 12: DON'T CARE FIELD TOO LONG

The explicit length given for a don't care field exceeds the microword length specified by WORD. Improper digits may have been assumed for the explicit length due to a missing comma or designator.

ERROR 13: ARITHMETIC OPERATION ON FIXED FIELD.

If a field is defined as a variable field in the definition file, an expression cannot be used as a VFS in the assembly file unless the field contained the % attribute in its definition.

ERROR 14: ATTRIBUTE ERROR

Both the negative (-) and inversion (*) signs have been assigned to a single variable or constant. This is not permitted. 4V-* or 4B#1011*- are meaningless.

ERROR 15: (Not used)

ERROR 16: MISSING END STATEMENT

The definition or assembly file is missing the END statement.

ERROR 17: ILLEGAL SYMBOL

A character other than A through Z, digits 0 through 9, or period was used in a name, or a comma may be missing between fields.

ERROR 18: OVERLAY ERROR

This message is given when two formats are overlaid and both of them contain constants for the same bit position. If the assembler is run using each of the formats in the overlay statement as a separate format, and the output is printed in block form, the erroneous bits are easily detected.

For example, if the definition file statements are:

```
A: DEF¥4X,B#1011
B: DEF¥B#01111,3X
```

and the source file statement is

```
A & B
```

the overlay error message occurs.

Rerun the assembly file with source statements given as

```
A
B
```

and block output requested which generates

```
XXXX |1| 011
0111 |1| XXX
```

It can easily be seen that bits |1| are causing the overlay error. The improper DEF can then be corrected and the overlay A & B can be used in the assembly file statement.

ERROR 19: FIELD LENGTH CONFLICT

The calculated or implicit field length for the constant or expression given after the designator does not have the same number of bits as the explicit field length. Check for a missing % or :, or a comma missing after the previous field.

This message may be output when commas are left out. For example,

```
8H#A39Q#274
```

is missing the comma between 3 and 9. Thus the program assumes A39 is to be substituted into the 8-bit hexadecimal field.

Similarly,

```
8H#A3,9Q27,4
```

will generate this error message since the comma between the 7 and 4 is misplaced.

ERROR 21: $SPECIFIED FOR NON-ADDRESS FIELD

In order to use the value of the program counter (indicated with a $) as a VFS, that field must contain the % attribute.

ERROR 22: (Not used)

ERROR 23: MISSING DESIGNATOR

A field has been encountered which contains only decimal numbers. This is not permitted for a field in a DEF, SUYB, or FF. Decimal numbers must be input as, n D# digits, where n is the explicit length of the field and digits are the decimal integers which generate the desired bit pattern or field value.

ERROR 24: SPACE DIRECTIVE ERROR

The value input following SPACE is interpreted as less than zero or greater than the number of lines given per page.

ERROR 25: ORG SET TO LESS THAN CURRENT PC

When ORG is encountered, the value given is compared with the current program (location) counter. If ORG is less than the program counter, the value given with ORG is ignored.

ERROR 26: NO FORMAT NAME AFTER &

When a line ends with an & and no continuation (/) is given at the beginning of the next line, this error is generated. A format name is missing after the &, or a / is missing on the continuation line.

ERROR 27: (Not used)

ERROR 28: ADDRESS NOT IN CURRENT PAGE

When the user gives a label or a label$ as a VFS or has defined the variable field with the $ attribute, this message will be generated if the left bits to be truncated do not match the corresponding bits of the current program counter.

ERROR 29: LENGTH REQUIRED FOR $ MODIFIER

Paged addressing (use of the $ as a modifier) requires the field length before the symbol in FF statements. Thus, 6SYMBOL$ is correct but SYMBOL$ is incorrect.

ERROR 30: ILLEGAL FIELD LENGTH IN FF STMT.

A field is greater than 16 bits in a FF statement. Only don't care fields may be larger than 16 bits.

ERROR 31: (Not Used)

ERROR 32: NO EXPLICIT LENGTH BEFORE (

An expression in a FF statement must be enclosed in (). The explicit field length must precede the (.

AMDASM ERRORS WHICH HALT EXECUTION

Error messages with n $\geq$ 100 cause execution to stop. They are listed in table 6-2 and described below.

TABLE 6-2. AMDASM ERRORS WHICH HALT EXECUTION

ERROR 100	COMMAND OPTION SYNTAX ERROR
ERROR 101	DEF TABLE OVERFLOW
ERROR 102	SUB TABLE OVERFLOW
ERROR 103	EQU TABLE OVERFLOW
ERROR 104	INCORRECT OR MISSING WORD SIZE
ERROR 105	UNEXPECTED END OF FILE
ERROR 106	FIELD TABLE OVERFLOW
Errors 101, 102, 103 and 106 occur when the amount of memory available has been exceeded.	

ERROR 100: COMMAND OPTION SYNTAX ERROR

The input command contains an error. Check for correct spelling of filenames and options, spaces between options, and correct drive specification with filenames.

ERROR 104: INCORRECT OR MISSING WORD SIZE

Either the WORD n command is not given as the first command (or the first command after TITLE) or the value given for n is <1 or >128.

ERROR 105: UNEXPECTED END OF FILE

The user has given an incorrect file name or the source file is not correct. AMDASM has encountered an end of file when it was still expecting data.

AMSCRM ERRORS

Table 6-3 is a list of the error messages output by AMSCRM:

TABLE 6-3. AMSCRM ERRORS

ERROR 1	COMMAND OPTION ERROR
ERROR 2	INPUT OUTPUT FILE NOT SPECIFIED
ERROR 3	FIELD LENGTH EXCEEDS MAXIMUM
ERROR 4	FIELD EXCEEDS MICROWORD SIZE
ERROR 5	TRANSFORMATION PARAMETER ERROR
ERROR 6	TRANSFORMED FIELDS OVERLAP

ERROR 1: COMMAND OPTION ERROR

There is an error in the execution command. Check for delimiters, correct option spelling, etc.

ERROR 2: INPUT/OUTPUT FILE NOT SPECIFIED

The input or output file was not specified in the execution command, or an incorrect filename was given.

ERROR 3: FIELD LENGTH EXCEEDS MAXIMUM

The maximum width of any field to be moved (W_n) is 16.

ERROR 4: FIELD EXCEEDS MICROWORD SIZE

The bit number given or the number of bits to be moved is incorrect. For example, if the microword is 32 bits wide, and the parameters

10,5,28

are given, the program attempts to move 5 bits to positions 28, 29, 30, 31, 32. This is impossible since the bit positions for a 32 bit microword only range from 0-31.

ERROR 5: TRANSFORMATION PARAMETER ERROR

An incorrect character or value has been given in the user's input S_n, D_n, W_n or a comma is missing between S_1 D_1 W_n or comma is missing between S, D or W.

ERROR 6: TRANSFORMED FIELDS OVERLAP

If the user attempts to move bits into positions where AMSCRM has already moved bits, this error occurs. For example, the parameters

```
6,9,3
15,11,3
```

would generate this error since they attempt to move two different bits into bit position 11.

AMPROM ERRORS

Table 6-4 is a list of AMPROM errors.

TABLE 6-4. AMPROM ERRORS

ERROR	1	DON'T CARE DEFINITION ERROR
ERROR	2	WIDTH INPUT SYNTAX ERROR
ERROR	3	WIDTH EXCEEDS MICROWORD SIZE
ERROR	4	TOO MANY PROM COLUMNS
ERROR	5	DEPTH INPUT SYNTAX ERROR
ERROR	6	WARNING DEPTH EXCEEDS MAXIMUM PC
ERROR	7	TOO MANY PROM ROWS
ERROR	8	ILLEGAL VALUE FOR ROWS OR COLUMNS
ERROR	9	ILLEGAL PROM NO., ROW, OR COLUMN DESIGNATION
ERROR	10	UNEXPECTED END OF FILE ON INPUT FILE
ERROR	100	COMMAND OPTION SYNTAX ERROR

ERROR 1: DON'T CARE DEFINITION ERROR

A value other than zero or one was input as the value for don't care bits. The user has input an incorrect character.

ERROR 2: WIDTH INPUT SYNTAX ERROR

The PROM width specified using n and/or I*b has been stated incorrectly. Check for missing commas or asterisks.

ERROR 3: WIDTH EXCEEDS MICROWORD SIZE

The width given for all of the PROMs total to so many bits that at least one additional PROM width s being specified. For example, if the microword width is 60 and PROM width is specified as 9*8, an error will be generated as there are 12 (72-60) extra bits specified which is greater than the 8-bit width of each PROM. Program execution stops. However, 8*8 will not generate an error since the extra 4 bits (64-60) will fit within one 8-bit wide PROM.

ERROR 4: TOO MANY PROM COLUMNS

The user is limited to 32 columns in his PROM map. When a number of columns greater than 32 is specified this error occurs.

ERROR 5: DEPTH INPUT SYNTAX ERROR

The data (r and/or t*d) specifying the PROM depths has been input incorrectly. Check for missing commas or asterisks.

ERROR 6: WARNING DEPTH EXCEED MAXIMUM PC

The depth specified by the user will require at least one additional PROM filled with don't cares.

Thus, if the object code depth is 120 words and the user specifies 3*64 to t*d, the extra 72 words are flagged as an error. However, if the user specified 2*62 (or 128) the extra 8 words would simply be filled with don't cares and the program continues executing.

ERROR 7: TOO MANY PROM ROWS

A PROM map may contain a maximum of 64 rows. This provides for 64K of storage if the user has chosen 1K PROMs. A PROM map with more than 64 rows is not permitted.

ERROR 8: ILLEGAL VALUE FOR ROWS OR COLUMNS

The user has input something other than a decimal integer Y or Rs or Cs or the letters N or A.

The user may have forgotten the - between Y_1 and Y_n or Cs_1 and Cs_n, etc.

ERROR 9: ILLEGAL PROM NO., ROW, OR COLUMN DESTINATION

The user has requested a PROM number or a PROM row or column using a decimal value greater than any of the PROM numbers, PROM row numbers, or PROM column numbers.

ERROR 10: UNEXPECTED END OF FILE ON INPUT FILE

This error only occurs when input to AMPROM is from a file (i.e., the user is not putting the data interactively). A line giving the don't care value, the PROM width or the PROM depth, or the printing information has been omitted.

ERROR 100: COMMAND OPTION SYNTAX ERROR

This error occurs due to illegal command options or illegal syntax. Execution halts and the correct command must be entered.

Check for misspelling, missing blanks or =, or incorrect drive specifications.

NOTE

Errors 1, 2 and 5 are indicated on the console and the previous data request is repeated. In order to end this loop, the user must input correct data or, if he inputs a control-C, the loop ends and the system is rebooted.

AMDOS 29 ERROR MESSAGES

If a system error occurs that is related to AMDASM, AMSCRM or AMPROM, AMDOS outputs an error message on the console. Table 6-5 is a list of AMDOS29 error messages.

TABLE 6-5. AMDOS29 ERRORS

(filename) FILE NOT FOUND
FILE EXTENSION ERROR
END OF DISK DATA ERROR
NO DIRECTORY SPACE
VERIFY
WRITE PROTECTED
FILE ERROR
CLOSE ERROR
FILE EXTENSION ERROR

(name) FILE NOT FOUND

The (name) input by the user cannot be located on the designated drive. Check for misspelling of the filename or the wrong drive designator.

This is a system error indicating an attempt to write outside the current file extent.

END OF DISK DATA ERROR

No more disk space for file data. Delete files from current disk or assign files to another disk.

NO DIRECTORY SPACE

The diskette directory is full. The user must indicate output is to go to another drive or he must make room on this diskette by deleting some files.

NOTE: If the user has inserted a disk which is write protected, he will receive a variety of error messages including:

VERIFY ERROR
WRITE PROTECTED
FILE ERROR
CLOSE ERROR
etc.

INDEX

INDEX (continued)

COMMENT SHEET

Address comments to:

Advanced Micro Computers
Publications Department
3340 Scott Boulevard
Santa Clara, CA 95051

TITLE: **AmSYS29/10 Microprogram Support Software User's Manual**
PUBLICATION NO: **059910515-001** **Revision A**

COMMENTS: (Describe errors, suggested additions or deletions, and include page numbers, etc.)

From: Name: _____ Position: _____

Company: _____

Address: _____

Advanced
Micro
Computers

A subsidiary of
Advanced Micro Devices
3340 Scott Boulevard
Santa Clara,
California 95051
(408) 988-7777
TELEX: 171 142