ALPHA MICROSYSTEM AM-100

ASSEMBLY LANGUAGE PROGRAMMING SYSTEM

USER'S REFERENCE MANUAL

ALPHA MICROSYSTEMS™
17875-N Sky Park North
Irvine, CA 92714

ALPHA MICROSYSTEM AM-100

ASSEMBLY LANGUAGE PROGRAMMING SYSTEM

USER'S REFERENCE MANUAL

ALPHA MICROSYSTEMS
17875 Sky Park North
Irvine, CA 92714

I would like to express my thanks to the following people for their assistance in the development of this manual:

John M. Keefe Jr.
Orange County Computer Center

Mike Roach
Educational Support Systems

Guruprem Singh Khalsa
Byteshop of Pasadena

and especially to Carolyn
for patience and understanding and dinners and ...

Dick Wilcox

ALPHA MICROSYSTEMS
17875 Sky Park North
Irvine, CA 92714

INDEX

SECTION 1 - PROGRAMMING AND OPERATION OF THE MACRO ASSEMBLY SYSTEM

SECTION 2 - OPERATION AND USAGE OF AM-100 MONITOR CALLS

SECTION 1

PROGRAMMING AND OPERATION OF THE MACRO ASSEMBLY SYSTEM

INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

The AM-100 microcomputer supports a flexible and efficient assembly language development system under the AMOS supervisor which includes the assembler, linkage editor, symbol file generator, and symbolic debug program. The assembler is a multi-pass macro assembler with conditional assembly directives, library copy function, and external segment links. The linkage editor is used to link multi-segment programs together and create a runnable program file. The operating system supports segment overlays thereby allowing a large program to be logically divided into smaller segments and executed sequentially. The debug program accepts a specially created symbol file as input and allows the program to be traced and debugged in symbolic instructions using all the labels as they were entered in the source program. All components of the assembly language development system run under control of the standard AMOS supervisor.

There currently exist over 70 supervisor calls in macro form that are used by the assembly language programmer to communicate with the AMOS monitor and make use of the routines it has to offer. These macro calls are predefined in a file called SYS.MAC located in area [7,7] on the AMOS system disk. The programmer uses a single COPY statement to include this complete library of predefined functions in his assembly language program and then refers to the supervisor calls by their macro names which makes for an easy-to-use communication link to the system resources. SYS.MAC also includes equate statements for many of the predefined system variables including the job table entries for the user's impure job variables.

To be compatible with the AMOS system architecture all programs must be written in totally relocatable code which means that the program may be loaded anywhere in RAM and executed without modifying any addresses within the program itself. There are machine instructions which assist in writing totally relocatable code, and by following a few simple restrictions, the task of writing assembly language programs for the AM-100 becomes almost foolproof.

Optionally, the user may write programs which are reentrant and then incorporate these programs or subroutines into the system monitor to be shared by all users without requiring a separate copy for each one. The rules for reentrant programming will not be delved into here since this is an advanced programming technique and requires specific rules which are not machine dependent. There are numerous books on the subject and all general practices will apply to the programming of the AM-100 system. There are a number of features in the instruction set which do lend themselves quite nicely to reentrant code and the proficient systems programmer will quickly recognize them.

This manual is designed to cover the procedures for writing assembly language programs, using the supervisor calls, and the operation of the component programs described above to implement and test the finished program. The user should be familiar with machine language programming techniques in general and in particular with the AM-100 machine instructions which are described in the WD16 MICROCOMPUTER PROGRAMMER'S REFERENCE MANUAL.

FILES USED IN THE ASSEMBLY LANGUAGE SYSTEM

This section will describe the files that are used during the normal course of building and testing an assembly language program. The files will be referred to by their extensions, i.e., a MAC file is any file with an extension of "MAC". All files described here will not necessarily be used by all programmers during any one session but you will eventually run across all of them at one time or another so you might as well know briefly what they are used for and how they are created.

MAC SOURCE FILES

MAC files are the original ASCII source files created by the programmer using the EDIT program. They are input to the assembler program (MACRO) which makes one or more passes over them depending on the assembly options selected. Any changes to be made to a program are made to the MAC file by using the EDIT program and then reassembling and relinking it. Library files which are included with the COPY assembly pseudo-opcode are also ASCII source files with an extension of MAC.

OBJ INTERMEDIATE OBJECT FILES

OBJ files are the direct output of the assembler (phase 2) and contain the assembled binary code, symbol references, internal symbol definitions, and unresolved external symbol references. OBJ files are not directly usable for anything by themselves but must first be processed by one of three other programs to get a finished file that has a direct use by itself. The linkage editor program (LINK) reads one or more OBJ files and creates a fully resolved and runnable binary program file in memory image format. The symbol file program (SYMBOL) reads the same OBJ files and creates a file which contains all user defined symbols and their resolved addresses. This file is used by the symbolic debug program (DDT). The assembler itself also rereads the OBJ file during phase 3 together with the MAC source file to create the ASCII list file.

PRG BINARY PROGRAM FILES

PRG files are created by the linkage editor program (LINK) and are the end result of all this nonsense we are attempting to describe. The PRG file is a binary memory image of the assembled program which is loaded into user RAM when the program is requested for execution. It must have been written using the rules for totally relocatable code so that it may be dumped into any memory location and executed without modification. One or more OBJ files may have been input to the linker for the creation of the single PRG file. Once the PRG program file has been checked out it may be placed into the common system area on DSK0:[1,4] where it will become available to all users if such is its fate.

OVR BINARY OVERLAY FILES

If the program contains overlay segments which do not all reside in memory at the same time, the linkage editor will generate one PRG main segment file and one or more OVR overlay segment files. Each overlay file will be generated as a result of an OVRLAY assembler pseudo-opcode. The PRG program segment will be responsible for the calling and executing of each of the other OVR segments during the running of the program. Overlay segments may be selectively bypassed as in the assembler itself which contains four overlays. Overlay files have the same memory image format as the PRG program files except that they are resolved at an effective address other than zero so that they will not completely overlay the controlling segment. This addressing is the direct responsibility of the programmer and will be described later in the section dealing with the OVRLAY opcode.

LST PROGRAM LISTING FILES

An optional output of the assembler is a complete resolved listing of the source program with the associated binary code that was generated. This list file is created during phase 3 of the assembly process and may be generated directly from the MAC and OBJ files by bypassing phases 1 and 2 with the /O assembly switch. The LST file is formated ASCII and may be typed with the TYPE command or examined in part by the EDIT program. It may also be printed by the line printer spooler if you have that job included in your configuration.

SYM RESOLVED SYMBOL FILES

The SYM file is a direct output of the symbol file generation program (SYMBOL) which takes one or more object (OBJ) files and creates a symbol table with all user defined symbols and their resolved machine addresses. The SYM file is used as input to the debug program (DDT) which may then operate with references to the user symbols in the program instead of absolute machine addresses. In a system where the program is always offset by some amount in memory this is almost essential to keep from going bananas while tracing execution flow of a program under test. The SYM file is in a special packed binary form and, as such, is not much good for anything except input to DDT.

IPF' INTER-PHASE WORK FILE

The IPF file is a temporary work file built during the assembly process by phase 1 of the assembler to carry information on to phase 2. The IPF file is packed binary junk and the only reason it is mentioned here is that if the system crashes during an assembly you may find one left on your disk. Erase it since it is useless and just takes up space. No problem if you don't find it since the next assembly of the same program will erase any IPF file it finds during phase 1 before attempting to create a new one.

MACRO SOURCE PROGRAM FORMAT

A macro source program is a single file with the extension MAC composed of a sequence of ASCII source statement lines. Each line must be complete in itself since there is no provision for multiple-line statements. Each statement may be one of the following depending on its function:

1. Valid machine instruction
2. Data generation statement
3. Symbolic equate statement
4. Assembly control statement
5. Conditional assembly directive
6. Macro definition
7. Macro call
8. Comment or blank line

Each line should be limited in size to under 100 characters and should be terminated by a carriage-return and line-feed pair which the editor provides when the return key is entered. Unless otherwise specified, all of the above lines may contain an optional comment field following the actual statement which starts with a semicolon (;) and extends to the end of the line. Spaces and tabs (CONTROL-I) are treated as equal and used to delimit fields within statements. Tabs are nifty to keep statement fields aligned and make for pretty listings. Lightning may strike down those who do not use tabs.

The term "user symbol" will be referred to several times during the course of this manual so it will be defined at this point. A user symbol is any name which is unique within the program under the various rules to be presented. A user symbol may be from 1-6 characters in length and made up from the character set defined by the alphabetics A-Z, the numerics 0-9, and the two special symbols "$" and "." with the first character being non-numeric. Symbols are packed RAD50 and stored as two words in the symbol table during the assembly process along with their current assigned value and attribute flags.

MACHINE INSTRUCTIONS

One machine statement is allowed per line and is assembled into a single machine hardware instruction which generates one, two, or three words of binary code depending on the instruction and addressing modes used. The general format of a machine instruction statement is:

{label:} {opcode} {operands} {;comments}

The label field is optional and is used to give a symbolic name to the current instruction being assembled. It must be terminated by a colon. The label may be any valid user symbol that has not been previously defined. The value of the label may have an attribute of either absolute or relocatable depending on the current assembly status. Relocatable symbols will be resolved during link-edit time by adding the label value to the current program relocation bias (calculated by LINK). More than one label may appear on the same statement line separated by colons in which case each label is given the same value as the current location. Any symbol used in a label field may not be redefined later

in the program. A label may appear as the only item on a line in which case it is assigned the address of the next byte of generated code.

The opcode field is required and is one of the machine instruction opcodes in mnemonic form such as MOV, CLRB, TST, ADD, etc. Refer to the WD16 MICROCOMPUTER USERS MANUAL for a complete description of all the machine instructions available in the AM-100 system. The opcode field is terminated by a space, tab, semicolon or carriage-return. If a label field was used the space or tab between the colon and the opcode is optional but recommended.

The operands field is required on those instructions which have either one or two operands. The operands field is separated from the opcode field by one or more spaces or tabs. If the instruction being used required two operands they are separated from each other by a comma. Leading spaces are always ignored in the operands field while the operands themselves are terminated by a space, tab, comma, semicolon or carriage-return.

The comments field is optional and if used is defined by a leading semicolon. The comments field then extends through the remainder of the line up to the carriage-return. Any valid ASCII characters are legal in the comments field.

DATA GENERATION STATEMENTS

Data generation statements resemble machine instructions in format and generate binary data within the program flow. The data generated is normally not interpreted during program run as executable instructions but rather as constant data such as ASCII messages to be typed or numeric values to be used by those instruction being executed. The general format of the data generation statement is:

{label:} {operator} {operands} {;comments}

The label field is optional and if used follows the same format and rules as the machine instruction label field. The operator field contains the specific data generation mnemonic for the type of data desired. These codes will be defined in another section. The operands field contains the actual data to be generated by the statement and its format depends on the type of operator in use. Some operators such as WORD and BYTE allow multiple operands within the same statement so that the amount of binary data generated by the one statement is variable. If a label is used, its value is always that of the address into which the first byte of data will be assembled. As with machine instructions, the comments field is optional.

There is a special default type of data generation statement which the user should be aware of. If no operator is present the statement is assumed to be a WORD statement and the operands field is interpreted as such. The assembler works in the following manner when analyzing statements:

1. Leading symbols which are terminated by colons are processed as labels and stored in the assembler symbol table.
2. The next symbol is first scanned for a match in the macro table which consists of all macros defined previously in the program.
3. If the operator symbol is not a macro name it is then matched

against the table of machine instruction opcodes, data generation operators, and assembly control pseudo-opcodes.

4. If none of the above result in a defined operator, the default WORD processor is entered and the symbol is assumed to be the beginning of the associated operands field for the WORD statement.

SYMBOLIC EQUATE STATEMENTS

A user symbol may be assigned a value by entering it on a statement line followed by an equal-sign (=) and the expression to which it is to be equated. The general format of the equate statement is:

{user symbol} = {expression} {;comments}

The equal-sign may have leading or trailing spaces and tabs if desired for formatting purposes. The expression may be any valid numeric expression but since all equate statements must be fully resolved during phase 1, any user symbols used in the expression must be defined at the time that the equate statement is encountered. Equate statements may not contain references to external symbols. The comments field is optional as in the machine instruction statement.

User symbols that are assigned values in the program may be reassigned a different value later in the program by using another equate statement to redefine the desired symbol. Labels may not be redefined by equate statements, however. If the relocation attribute of the evaluated expression is zero the value assigned to the symbol is absolute. If the relocation attribute is non-zero then the value assigned is relocatable. If the expression contains a register symbol then the equated symbol is also given a register attribute. In other words, the value assigned to the user symbol pretty much follows the attributes of the expression to which it is equated.

ASSEMBLY CONTROL STATEMENTS

Assembly control statements cover a wide range of functions that generally set up or alter the parameters which control the assembly process. They do not themselves generate any binary code but are used for such purposes as listing format control, numeric radix assignment, and program generation or addressing information. The general format for assembly control statements is:

{pseudo-opcode} {arguments} {;comments}

The pseudo-opcode is the mnemonic which defines the function to be performed. These are listed in a later section along with an explanation of what each one does. Some of them require arguments that are needed to set up parameters. These arguments are separated from the pseudo-opcode by one or more spaces or tabs. As in other statement formats, the comments field is optional. Unless otherwise specified in the detailed explanation, labels are normally not permitted in assembly control statements.

CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly is defined as statements within defined bounds which are selectively assembled or bypassed depending on the value of some variable at the time the assembly is performed. The bounds are made by conditional assembly directives which specify the variable or variables to be tested and the condition to be met in order for the assembly to occur. Conditional assembly directives are most commonly used in conjunction with macro definitions to direct the tailoring of each macro call as it is encountered. Conditional assembly directives will be fully explained in a later section dedicated entirely to them.

MACRO DEFINITIONS AND MACRO CALLS

Macros are defined as one or more valid statements which may be called for by using a single symbol (the macro name) within the program anytime after the macro has been defined. Macros are always defined by the user within his program or within a library which is called into his program by the COPY statement. The library file called SYS.MAC is a collection of over 70 such macro definitions which define all the supervisor calls available to the user program for communicating with the monitor routines. This library file is supplied on the AMOS system disk in area [7,7].

Macro calls are those statements which name the defined macro as the operator of the statement and give the specific arguments to be used by the macro (if any are required). A macro call within the program causes the defined macro to be included in its tailored form at the point of the call. Macro calls normally cause one or more machine instructions to be assembled and the respective binary code to be generated.

Macro definitions and macro calls are defined more fully in a later section.

COMMENT LINES AND BLANK LINES

Statements which begin with a semicolon (after any leading spaces and tabs) are considered comment lines and do not result in the generation of any binary code or in the alteration of any assembly control parameters. They are useful only for documenting the source programs and making them look real pretty. Blank lines are also considered comment lines and are for appearances only in the source file. Lightning may also strike down those who do not use sufficient comment lines in their programs.

TERMS AND EXPRESSIONS

This section describes the various terms and components used in MACRO source statements including the defined character set for the construction of symbols and expressions.

CHARACTER SET

The entire ASCII character set is legal in MACRO source programs except for the control characters. Lower case characters will be translated to upper case before being used in syntax checking of each source line. The characters which are valid in user defined symbols are limited to A-Z, 0-9, "$" and "." due to the fact that symbols are packed RAD50 before being stored in the symbol table. The folowing list will give the special characters that are recognized by the assembler when scanning source lines.

:	Label terminator
;	Comment field indicator
=	Equate statement operator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminating register indicator
,	Operand field or macro argument separator
.	Value of the assembly current location counter when used as a term
<	Initial argument or expression indicator
>	Terminating argument or expression indicator
+	Arithmetic addition operator or autoincrement mode indicator
-	Arithmetic subtraction operator or autodecrement mode indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical inclusive OR operator
'	Single ASCII character term indicator
"	Double ASCII character term indicator
[	Initial RAD50 triplet term indicator
]	Terminating RAD50 triplet term indicator
^	Universal unary indicator

The use of the above legal characters out of context for their designed purposes will cause the generation a syntax error (code Q).

TERMS

A term is the basic unit of data in an arithmetic expression and may be one of the following:

1. A number as composed of legal digits within the current radix of the system or as temporarily defined by the inclusion of a leading temporary radix change operator

2. A user symbol as previously defined which is given an assigned value either by its use as a label or a direct equate statement

3. An ASCII conversion defined by the single or double quote indicators

4. A RAD50 triplet enclosed in square brackets

5. The period symbol (.) which represents the current value of the assembly current location counter

6. An expression or term enclosed within angle brackets. Angle brackets are used to alter the normal hierarchy of expression evaluation which is normally done in a left-to-right manner. Any quantity enclosed within angle brackets will be evaluated before the remainder of the expression in which it is found. The action of angle brackets within a MACRO source expression is the same as that of parentheses within a normal arithmetic expression such as is used in the BASIC language. Angle brackets may also be used to apply a unary operator to an entire expression such as -<16/A>.

EXPRESSIONS

An expression is a combination of terms and operators which will evaluate down to an unsigned 16-bit value in the decimal range of 0-65535. Negative values in the range of -32768 through -1 will be stored properly after evaluation but will be treated the same as their unsigned conterparts in the range of 32768 through 65535. The evaluation of any expression also includes the evaluation of the mode of that expression (absolute, relocatable, and external) and the register designation of the expression.

Operators are defined as unary or binary. Unary operators precede a single term and alter the evaluation of that term alone. Multiple unary operators may be applied in sequence to the same term and are evaluated in reverse order. Binary operators combine two terms to give a resultant effective single term value. Multiple binary operators are illegal.

Expressions are evaluated left to right under the hierarchy of the operators which are in use within that expression. Angle brackets may be used to alter the normal process of evaluation. Unary operators always take precedence over binary operators and are applied to the associated terms first.

The legal operators are:

+	Unary plus sign (default of any term not preceded by another unary)
-	Unary minus sign which negates the associated term value
^C	Unary one's complement operator (XOR's the term with all ones)
^D	Temporary radix change to decimal for the associated term
^B	Temporary radix change to binary for the associated term
^O	Temporary radix change to octal for the associated term
^H	Temporary radix change to hexadecimal for the associated term
+	Binary addition operator
-	Binary subtraction operator
*	Binary multiplication operator

/ Binary division operator
& Binary logical AND operator
! Binary logical inclusive OR operator

Expressions are evaluated as being either absolute, relocatable, or external. This distinction becomes particularly important since we are writing totally relocatable code for the AM-100 system. The following rules apply in the evaluation of the relocation attribute of an expression:

1. An expression is absolute if its value is fixed and contains no relocatable terms. Also, a relocatable term minus another relocatable term results in an absolute value. Labels allocated within an absolute section (ASECT) will be assigned absolute values and attributes.

2. An expression is relocatable if its value is fixed relative to the current program base which is relocatable at load time. The value may have an offset added to it by LINK if it is not within the first segment of a program file. Labels allocated within a relocatable section (RSECT) will be assigned relocatable values and attributes.

3. An expression is defined as external when one or more of its terms is an external symbol reference. This expression will not be fully resolved until the program file is generated by the linkage editor (LINK) when the external terms are defined. The final resolution of an external expression may be relocatable or absolute, depending on the attributes of the terms involved (both internal and external). The linkage editor also contains all the mechanics for evaluating the attributes of resolved expressions.

NUMBERS

Any source item which starts with a digit (0-9) is considered to be a number and this number will be evaluated under the currently prevailing radix unless preceded by a temporary radix operator or followed immediately by a decimal point. The prevailing radix always starts as octal (base 8) at the beginning of any assembly but may be changed by the RADIX assembly control statement. Any number that terminates with a decimal point will be evaluated as decimal (base 10) regardless of the prevailing radix. Fractional numbers are not allowed in MACRO source statements since all numbers must evaluate to a 16-bit binary integer value.

The prevailing radix controls the default evaluation of numbers and may be set by the RADIX statement to any value from 2 (binary) through 36 (which is really weird)! Numbers in a base above 10 (decimal) use the alphabetic characters A-Z to represent the digit values of 10 through 35. The most common system above base 10 is hexadecimal where the letters A-F represent the digit values 10-15. All numbers must begin with a digit 0-9 so that the hexadecimal value of F56 must be entered as 0F56 to distinguish it from a user symbol.

Negative numbers are preceded by a minus sign and will be evaluated and stored by the assembler in two's complement form. Positive numbers may optionally be preceded by a plus sign but this is not required.

REGISTER SYMBOLS

The WD16 microcomputer (the heart of the AM-100 system) contains eight 16-bit registers which are symbolically named and used as follows:

R0 - register 0, general purpose
R1 - register 1, general purpose
R2 - register 2, general purpose
R3 - register 3, general purpose
R4 - register 4, general purpose
R5 - register 5, general purpose
SP - register 6, stack pointer
PC - register 7, program counter

These eight symbols are already defined to the assembler and must be used when the address mode explicitly requires a register to be referenced. The above register symbols have a register attribute associated with them and the user may equate his own symbols to these registers if he so desires. The register attribute will be carried forth to his newly defined symbol. For example, the equate statement IOPTR=R4 will equate the user symbol IOPTR to the value of 4 and also give it a register attribute so that it may be used in place of R4 for address modes.

ASSEMBLY LOCATION COUNTER

During the assembly process, sequential memory locations are assigned to all machine instructions and data constants as they are encountered in the source program. At any given statement, the next byte to be assigned will be internally stored in the assembly location counter. This address may be used in expressions by referencing the period (.) as a symbolic term. For example, the instruction "JMP .+6" will cause a jump to the address which is 6 bytes in front of the first byte of the instruction itself.

The assembly location counter has an attribute associated with it which is either absolute or relocatable. Initially it is set up in the relocatable mode and cleared to zero value for the allocation of relocatable binary code as machine instructions and data constants are assembled. If an ASECT statement is encountered the attribute of the assembly locataion counter is changed to absolute which means the address associated with it will not be adjusted by the LINK program. If an RSECT statement is encountered the attribute will be set back to relocatable again which means that the address associated with it will be adjusted by the LINK program to compensate for the program segment offset. The assembler also maintains two separate address counters for switching between ASECT and RSECT sections.

Initially the value of the assembly location counter is set to zero and is incremented as each statement which produces binary code is assembled during phase 1. The setting of the assembly location counter may be changed at any time explicitly by a direct equate statement using the period symbol instead of a user symbol. For example, the statement ".=500" will force the assembly location counter to take on a value of 500 and begin all assembly allocation from that point on.

ASSEMBLY CONTROL PSEUDO-OPCODES

Assembly control statements perform a wide variety of functions which do not in themselves generate any binary code but instead, set up or alter certain parameters which control the assembly process. Each statement consists of a defined assembly control pseudo-opcode followed by optional arguments as required by the specific format. These pseudo-opcodes will be described here along with the required arguments for each.

COPY filename

The COPY statement allows another file to be included in the assembled program at the point where the COPY statement is located. The entire copied file is assembled but conditional assembly statements may be used to omit certain portions if desired. The most common use of this statement is for the inclusion of the standard library file SYS.MAC which defines all system call macros and system parameters to the program.

The filename specifies the name of the file to be copied into the source program during assembly. Note that the actual source program is not actually modified but rather the assembler merely gets the input from the copied file and then returns to the original source file. A copy file may not include another COPY statement within itself although the original file may include as many individual COPY statements as desired. The filename may actually be a complete file specifier which references another disk area. If the extension is not specified, the assumed default is MAC. If the area is not specified the assembler first searches the system library area in DSK0:[7,7] and if not found there, searches the current user disk area. An error message results if the file cannot be found and the assembly is aborted. The user may put other library files into the system library area DSK0:[7,7] which will then become available to all users through the COPY statement.

The source statements in the copied file are not normally output during the listing phase of the assembly since most users do not want the system library (SYS.MAC) and other common library routines to be repeated in all program listings. This may be overidden by using a "/L" switch following the filename in the statement which will cause the copied file to be included in the assembly listing.

PAGE

The PAGE statement causes a skip to the next page during the listing phase before continuing with the listed output. No action takes place other than this during assembly.

ASECT - RSECT

The ASECT statement causes the assembler to generate code for the absolute section of the program. This code will not be modified during LINK editing and the values assigned to labels will not have the relocatable attribute flag set.

The RSECT statement causes the assembler to generate code for the relocatable section of the program. This is the normal section for the AM-100 system which always relocates the program in user memory. This code will be modified during LINK editing and the values assigned to labels will have the relocatable attribute flag set. Two separate assembly location counters are maintained during program assembly.

SYM - NOSYM

The SYM statement causes all following user symbols to be output to the object file along with their assigned values. The NOSYM inhibits this output for all following user symbols. These symbols are later used by the SYMBOL program to generate a reference file for the dynamic debug program DDT. No change in the actual program is ever noticed by the use of SYM and NOSYM.

EVEN

The EVEN statement forces the next binary code to be generated on a word boundary (next even byte) by incrementing the assembly location counter if it is odd (no change if it is even). This is necessary since all instructions must lie on a word boundary for proper execution by the AM-100 system.

RADIX n

The RADIX statement forces a new default radix to be set up in the assembler. The default radix of the system determines how all numbers that are not preceded by a temporary radix operator (^B,^D,^H,^O) will be interpreted. The radix change argument "n" must be a decimal number in the range of 2-36. Radix values above 10 will use the letters A-Z to represent the digit values of 10-35 inclusively. The default radix of all assemblies is base 8 (octal) in the absence of any explicit RADIX statement.

END

The END statement terminates the source file and is included only to give a defined end on the listing. In the absence of an END statement the assembly will terminate with the logical end of input file. Note that if an END statement is encountered anywhere in the source input (including inside a copied file) the assembly will terminate whether the logical end of the input file has been reached or not.

DATA GENERATION PSEUDO-OPCODES

The MACRO assembler has several pseudo-opcodes which generate specific data constants within the program area for use as text messages, constant values, tables, etc. This section will list these pseudo-opcodes and give details on the data formats which are generated by them. All statements may have labels in which case the label is assigned the address that will receive the first byte of the generated data. All data statements begin allocating their specific data formats at the address specified by the assembly current location counter and generate multiple bytes in sequence incrementing the current location counter as necessary. Those statements which generate byte data (BYTE, ASCII, BLKB) may begin and end on any byte address, odd or even. Those statements which generate word data (WORD, RAD50, BLKW) must begin on a word boundary (even byte) or else a boundary error (B) will result. The EVEN statement may be used at any point where the status of the current location counter is in doubt to insure an even boundary.

BYTE

The BYTE statement generates one or more bytes of data. The arguments for generating the data consist of expressions separated by commas. Any legal expression is valid but only the lower byte will be stored after evaluation. Some examples are:

```
ZER:    BYTE    0               ;Generates one byte of data containing zero
        BYTE    1,2,3           ;Generates three bytes of data containing 1,2,3
MULTI:  BYTE    A-B,TAG*4,SAM   ;Generates three bytes of data
        BYTE    'A,'Q           ;Generates two bytes of ASCII data
```

WORD

The WORD statement generates one or more words (16 bits each) of data. The arguments for generating the data consist of expressions separated by commas. Any legal expression is valid which evaluates into a 16-bit value. WORD statements may also be generated by default if the first symbol on a line (after any labels) is not defined as an opcode, pseudo-opcode or macro name. Some examples are:

```
ZER:    WORD    0               ;Generates one word (two bytes) of data zero
        WORD    1,2,3           ;Generates three words of data containing 1,2 ,3
        WORD    A-B,"QT,SAM-.   ;Generates three words of data
        SAM                     ;Generates by default the value of SAM
```

ASCII

The ASCII statement generates one or more bytes of ASCII data. The argument for generating the data is a string of legal ASCII characters bounded on both ends by the same character which must not be included in the data string itself. Only one such string may be generated by each ASCII statement. Some examples are:

```
MSG:    ASCII   /THIS IS A MESSAGE/  ;Generates a string of 17 data bytes
        ASCII   /Q/                  ;Generates a single data byte of "Q"
MSG2:   ASCII   $ I/O TERM $         ;Generates a string of 10 data bytes
```

RAD50

The RAD50 statement generates one or more words (16 bits each) of data. The argument is a string of valid RAD50 packable characters bounded on both ends by the same character which must not be included in the data string. The legal characters for RAD50 packing are A-Z, 0-9, dollar-sign ($), period (.) and space. One packed word will be generated for each three characters in the string or fraction thereof with trailing spaces being assumed to fill out the last triplet. Some examples are:

```
DDB:    RAD50   /DSK/          ;Generates one word of packed data
        RAD50   /SAM QQ/       ;Generates two words of packed data
        RAD50   /ABCD/         ;Generates two words (same as RAD50 /ABCD  /)
```

BLKB - BLKW

These statements do not actually generate data but are included in this section because they result in the allocation of memory in a defined manner. The BLKB allocates an area of bytes and the BLKW allocates an area of words. In all other respects they operate the same. The argument for each is a single expression which evaluates down to a value between 0 and 65535. This value is then added to the assembly current location counter (twice if BLKW) which effectively reserves that block of memory and continues allocating memory at the new address. Normally this results in a contiguous area of all zeros since the linker clears all blank areas when it generates the program file. This action does not always happen, however, because the location counter may be stepped back into the reserved area in which case the new data will overlay the reserved block of memory. This is an important concept in dealing with the absolute section since no data is actually generated by these statements, only memory addresses are reserved. Some examples are:

```
DATA:   BLKB    44              ;Reserves 44 bytes of memory
        BLKB    A*B             ;Reserves A*B bytes of memory
        BLKW    200             ;Reserves 200 words (400 bytes) of memory
```

SEGMENTING ASSEMBLY LANGUAGE PROGRAMS

The MACRO assembler together with the LINK editor and monitor overlay calls support a powerful method of segmenting and overlaying programs for both convenience during system development and memory conservation during execution. This section will describe the methods available for the various options and also the assembler pseudo-opcodes which help support the system. The pseudo-opcodes to be described are INTERN, EXTERN and OVRLAY.

There are several reasons for segmenting a program and also different methods for doing so, depending on the end result desired. A very large source program takes longer to edit (even a small change) and gives a greater opportunity for total loss if some disaster strikes the file links. A large program also takes longer to assemble and more memory in which to do so. Segmented programs may be organized in such a manner as to allow portions of the program to be resident in memory and other portions to be called in from disk only as required. Segmented programs may also contain duplicate symbols if the program segments are assembled separately and linked together by LINK. Also, program segments which are assembled separately may also be listed separately resulting in less listing time (and less paper used) for each change that is made.

The simplest method for creating a program in segments gains one of the above advantages. This method makes use of the COPY statement and allows a large program to be edited as multiple segments which are then copied into the main source program by using one COPY statement for each segment. As changes are made to the source program you need only edit the segment which requires the changes. The assembly is done, however, on the complete source program since all copied files are included in the source input. Only one object file results and only one single list file can be created. The /L option on the COPY statement may be used to control those segments that are desired to be included on the listing itself.

A more complex but flexible method is to break up the program into logical segments which may be assembled separately and then linked together at a later time by the LINK program. Several object (OBJ) files result as output of the different segment assemblies which are then input to the LINK program which creates a fully resolved and runnable program (PRG) file. The advantages of the COPY method are realized as well as the added advantage of having to assemble only those segments which require changes. The LINK process runs much faster and requires less user memory than the assembly process. One of the requirements of a program which is segmented in this manner is that all references to routines and data constants which reside in another segment must be done through two special assembler pseudo-opcodes, INTERN and EXTERN. Since a reference to a routine in another segment is not defined during the assembly of the calling segment, the symbol (name of the routine) is said to be "external". It is declared external by the EXTERN statement which tells the assembler that it is defined and will be resolved by the linkage editor at a later time. The segment in which the routine exists then delares that symbol as "internal" via the INTERN statement which tells the assembler to output the symbol with a special code which defines it to the linkage editor for final resolution.

The method of segmenting a program and then creating a single runnable program

with LINK may be extended one step further using a feature in the monitor which allows program segments to be called in from the disk and overlay an existing portion of the main program. A segment which is to be used as an overlay defines itself as such by using the OVRLAY statement and giving the address at which the overlay is to be loaded. The main program then uses a special form of the FETCH supervisor call to load the overlay segment and then executes it by jumping to a known segment start address. This implementation of overlaying segments is used in the MACRO assembler itself and conserves user memory during execution of large system programs. The LINK program creates one program (PRG) file for the main segment and one overlay (OVR) file for each overlay segment in use.

INTERN

The INTERN statement is used to define one or more user symbols as internal to the segment so that they will be defined to the linkage editor program for final resolution. Each INTERN statement may be followed by one or more internal user symbols separated by commas. As many INTERN statements as required may be used in the program. There is also no limit to the number of symbols that may be referenced by each INTERN statement except for the physical line length.

Each symbol that is referenced in an INTERN statement must be defined within the segment either as a label on a routine or constant or as a value by an equate statement. The symbol will then be available to the LINK program for resolving references to it which come from EXTERN statements in other segments. Any symbol defined as external in a segment that has not been defined as internal in another segment will result in an undefined error during linkage editing. A symbol may never be defined by more than one INTERN statement during any one LINK run, i.e. the same symbol cannot appear as internal in two different segments that will eventually be linked into the same program.

EXTERN

The EXTERN statement is used to define one or more user symbols as external to the segment so that they may be resolved by the linkage editor program. Each EXTERN statement may be followed by one or more user symbols separated by commas. As many EXTERN statements as required may be used in the program. There is also no limit to the number of symbols that may be defined by each EXTERN statement except for the physical line length.

Each symbol that is defined by an EXTERN statement may be referenced within the segment just as if it had been defined within the segment as a label or an equate statement item. There is no limitation placed on its use as a term within any operand expression since the LINK program has complete expression resolution mechanics built in. There are two restrictions to its use within the segment. An externally defined symbol may not be used within the address operand of any branch instructions (BR, BEQ, BGT etc.) due to the fact that there is no way to insure that the resulting placement will fall within the 127-word relative requirement. They may, however, be used within the address operand of the jump (JMP) instruction. The second restriction is that an equate statement may not contain any externally defined symbols in its operand expression since all equates must be fully resolvable as they are encountered.

The LINK program builds a symbol table from all the symbols referenced in all INTERN statements in all program segments. It then goes back and resolves all expressions containing symbols defined by EXTERN statements by looking them up in the table of INTERN symbols. Any symbol defined in an EXTERN statement but not matched by some INTERN symbol will give an error message during linkage editing.

OVRLAY

The OVRLAY statement identifies a program segment as being an overlay file instead of a continuation of the main program file. It also defines the address of the base of the overlay relative to the base of the main program so that the loading of the overlay segment is done at the proper spot in the program memory area. The OVRLAY statement takes a single argument which is a user symbol that must be defined in some other segment in an INTERN statement. The OVRLAY address will be resolved by LINK when the files are processed. The OVRLAY statement must be the first non-comment statement in the program if it is used. Information on the code used to load the overlay segments into memory will be found in the description of the FETCH supervisor call. Further information on processing of the OVRLAY statement may be found in the section describing the LINK program processing.

CONVENIENCE PSEUDO-OPCODES

There exist a few pseudo-opcodes in the assembler that I refer to as convenience opcodes for lack of a better term. They are those few goodies that tend to make life easier for those of us that are used to working on larger machines and have tended to become set in our ways. These opcodes do not really do anything that cannot already be accomplished by the existing source language in some other format but they are easier to understand and make the listing more readable when used in the form that they have been implemented here. Some of them are implemented directly in the assembler program itself while others exist as predefined macro calls in the system library SYS.MAC which is normally called by all programs.

EXTENDED CONDITIONAL JUMPS

One very frustrating thing about editing some new changes into a program is when you find that an existing BNE (or other conditional branch) no longer reaches due to the new code extending the address out of the 127-word limit for branches. The most common solution to this problem is to replace the offending branch with a branch of the opposite condition followed by a jump to the desired address. In other words, our BNE TAG could be replaced by BEQ .+6 followed by JMP TAG which effectively does the same thing. The only problem here is that this makes the listing somewhat less than clear when trying to decipher the flow of the program. I have therefore implemented into the assembler a set of conditional jump opcodes which effectively generate this two-instruction code sequence for the proper opposite conditional but which still look very readable in the source listing. These opcodes have been listed here along with the actual WD16 instructions generated:

JEQ	TAG	generates	BNE	.+6	followed by	JMP	TAG
JNE	TAG	"	BEQ	.+6	"	JMP	TAG
JPL	TAG	"	BMI	.+6	"	JMP	TAG
JMI	TAG	"	BPL	.+6	"	JMP	TAG
JLO	TAG	"	BHIS	.+6	"	JMP	TAG
JHI	TAG	"	BLOS	.+6	"	JMP	TAG
JLOS	TAG	"	BHI	.+6	"	JMP	TAG
JHIS	TAG	"	BLO	.+6	"	JMP	TAG
JLT	TAG	"	BGE	.+6	"	JMP	TAG
JGT	TAG	"	BLE	.+6	"	JMP	TAG
JLE	TAG	"	BGT	.+6	"	JMP	TAG
JGE	TAG	"	BLT	.+6	"	JMP	TAG
JCC	TAG	"	BCS	.+6	"	JMP	TAG
JCS	TAG	"	BCC	.+6	"	JMP	TAG
JVC	TAG	"	BVS	.+6	"	JMP	TAG
JVS	TAG	"	BVC	.+6	"	JMP	TAG

Remember that although these opcodes are easier (require less planning) than the simple branches they do actually generate three words of binary code instead of only one so use them only when necessary if space is at a premium.

PUSH - POP

The hardware stack in the WD16 is normally referenced by its index register (SP) and transferring words of data to and from the stack are done by MOV instructions. Many machines have dedicated instructions to push and pop data to and from the stack. In order to make the flow of system programs a little clearer for those of us used to pushing and popping, two macros have been implemented in SYS.MAC which recognize the PUSH and POP instructions. Each takes a normal source address argument but each also has a special default format which is used when no specific argument address is desired. These instructions generate the following code:

```
PUSH  SRC    generates    MOV  SRC,-(SP)     ;Pushes SRC onto stack
PUSH              "       CLR  -(SP)         ;Pushes a zero onto stack
POP   DST         "       MOV  (SP)+,DST     ;Pops stack into DST
POP               "       TST  (SP)+         ;Removes top stack word
```

CALL - RTN

The normal subroutine calling sequence of the WD16 is the JSR instruction which links its arguments through any of the eight registers. The assembler recognizes the more popular mnemonic opcode CALL for which it generates a JSR instruction. In addition, if no register is specified in the CALL or RTN instructions the assembler assumes the most commonly used register PC for its argument linkage. In other words:

```
CALL  TAG    generates    CALL  PC,TAG
RTN               "       RTN   PC
```

OFFSET

There are many times during the programming of totally relocatable code where an address must be expressed and stored as a relative offset from the location of the constant itself. In other words, the storage of the address TAG must be in the form of TAG-. which is actually the offset from the current position of the constant itself to the address defined as TAG. The value of this constant offset will not change no matter what its position in memory happens to turn out to be. A good example of the use of relative address offsets is in the tables associated with the instructions TJMP and TCALL which must be relative offsets and not direct addresses. The OFFSET pseudo-opcode has been implemented to make the listings a little more obvious as to intent. The OFFSET opcode takes a single address argument and generates the relative offset to that address from the current position of the constant.

USER DEFINED MACROS

It is often convenient for the user to create his own opcode definitions which, when used in the source program, result in the creation of a predefined sequence of one or more source code statements. These user-created opcodes are called "macros" in assembly language programming and the AM-100 assembler supports a flexible macro subsystem which accomplishes this objective. There are two phases which the user goes through when using macro calls. First, the macro opcode is defined once in the program as a series of source code statements along with possible dummy arguments. This is done only once and remains defined throughout the remainder of the assembly process. Second, the user then invokes the macro by a single source statement giving the macro name along with optional real arguments which replace the defined dummy arguments in the macro source code which is generated. Calling the macro in this manner causes the macro statement to be replaced by the defined sequence of source code statements which have been custom tailored by the optional real arguments in the calling statement. This calling sequence may be done as many times as needed in the source program with as many different real arguments as desired.

MACRO DEFINITION

Defining a macro generates no actual binary code in the program but merely places the macro definition in a special table in the assembler memory work area. Calling the macro which then generates the sequence of source statements is the process that actually generates the binary code. If the macro is never called in the program or if the macro does not contain any code-generating source statements, no binary code is produced. The use of conditional assembly directives within a macro definition may result in no code-generating statements for this particular call to the macro. The fact that no code is actually generated if the macro is never called is an important concept since it then allows libraries to be created which may contain many macro definitions that are standard for a particular user system. Those macros that are never called in any specific program do not generate any code and therefore take up no additional memory. The system library SYS.MAC contains over 70 such macro defintions which define the supervisor calls to the monitor.

There are two formats available for use in defining macros. The normal format allows one or more source lines to be generated as a result of the macro call. The single-line format restricts the macro definition to one line of generated source code but takes up less room on the source listing (and looks niftier too).

The general format for multiple-line macros is:

```
DEFINE  name    {dummy argument list}
        source line 1
        source line 2
            ...
            ...
        source line n
        ENDM
```

The general format for a single-line macro is:

```
DEFINE  name  {dummy argument list} = source line
```

In both forms the macro name is any legal user symbol which effectively becomes the opcode by which the macro is called. This symbol may duplicate a label in the program or may even redefine an AM-100 pseudo-opcode or a WD16 machine opcode (you can redefine the MOV opcode to do an ADD if you really want to confuse some people). A macro name may only be defined once and an attempt to redefine it later in the program will give unspecified results in the current version of the assembler.

The dummy argument list is optional in both forms and if used consists of one or more user symbols separated by commas. These symbols are unique only wihtin the actual definition of the current macro and may be duplicated in other macro argument lists or may even be other opcodes and defined symbols. These dummy argument symbols will never appear as such in the generated sequence of source statements when the macro is called but will be replaced by the equivalent real arguments supplied in the calling statement. The dummy argument symbols may appear anywhere in the definition source lines, even as labels. Each time a dummy argument is encountered when generating the source lines during a macro call it will be replaced by the real argument supplied in its place by the calling statement.

The multiple-line macro definition source statements begin with the line immediately following the DEFINE statement and continue through to but not including the ENDM termination line which must be there. When the macro is called in the program text, all macro source lines will be generated and assembled just as if they had been entered directly into the source program explicitly. In the single-line form the source line begins with the character following the equal sign and continues through (and including) the carriage-return and line-feed pair which terminates the DEFINE statement line.

Macro definitions must not be nested within other macro definitions. Macro processing is done on a special prepass scheme which prohibits the processing of any DEFINE statements within another DEFINE statement.

A comment may follow the dummy argument list in the multiple-line form but should not be used with the single-line form. Comments should be avoided in the actual generated source lines in the macro definition simply because the entire source text is stored in work memory as ASCII characters which includes all comments if used. This may tend to use up work memory to the extent that you may not have enough to finish the assembly. KABOOM!

A label must not be used on the DEFINE statement line since it has no meaning. Labels may be used on the calling statements. A label must not be used on the ENDM line or the entire ENDM line will not be detected.

The following examples may help to clarify things a bit.

A macro called ADDIT which generates four instructions:

```
DEFINE  ADDIT
        MOV     R1,R3
        ADD     R3,SUM
        ASL     R3
        ADD     R3,SUM
        ENDM
```

A macro called XCHNG which exchanges two memory words:

```
DEFINE  XCHNG   MEMA,MEMB
        MOV     MEMA,R1
        MOV     MEMB,MEMA
        MOV     R1,MEMB
        ENDM
```

A macro called FOOF which subtracts a memory word from the top stack word:

```
DEFINE  FOOF    TAG
        SUB     TAG,@SP
        ENDM
```

The same FOOF macro in the single-line format since only one line is used:

```
DEFINE  FOOF  TAG = SUB  TAG,@SP
```

For some more complex examples of macro definitions, print out or inspect the system library SYS.MAC which defines all of the supervisor calls used by the AM-100 timesharing system.

MACRO CALLS

The actual generation of the defined source code comes when the user calls the macro by its name within the text of his source program. The macro must have been defined prior to its first reference. Macros are only processed for definition during phase 1 of the assembly process. Macro calls have the same format regardless of whether the macro definition is multiple or single line format:

```
{label:}  name  {real arguments}  {;commments}
```

The label is optional and if used will be assigned the address contained by the assembly current location counter. This will normally be the address of the first byte of code which is generated by the macro source lines assuming that the macro does actually generate code. If the macro does not generate code then the label will still be defined but it will represent the address of the next byte of code that is generated after the macro call.

Name represents the name given to the macro definition which becomes the effective opcode by which the macro is called.

The real arguments are used when the defined macro had a dummy argument list and actually replace the dummy arguments in the source code text of the macro definition. The real arguments replace the dummy arguments in the exact same order that the dummy argument list was in on a one-for-one basis. The first real argument in the call takes the place of each occurance of the first dummy argument in the definition, and so on for all the arguments. If there are not enough real arguments given in the call to fill all required dummy arguments the unfilled dummy arguments take on a null value and are effectively replaced with nothing. If there are more arguments in the call then required to fill the dummy arguments in the definition, the excess arguments are ignored.

Refer to the following example:

```
DEFINE  TBLADD  ARG1,ARG2,ARG3
        MOV     ARG1,R1
        ADD     ARG2,R1
        MOV     R1,ARG1(ARG3)
        ENDM
```

This macro is called TBLADD and requires three real arguments.
Assume the following call in the user program:

```
SAM:    TBLADD  SUMS,ENTRY,R5
```

The following source statements would be generated:

```
SAM:    MOV     SUMS,R1
        ADD     ENTRY,R1
        MOV     R1,SUMS(R5)
```

It is evident from the usage that ARG3 must be a register.
Assume that only two arguments were given in the call:

```
SAM:    TBLADD  SUMS,ENTRY
```

The following source statements would be generated:

```
SAM:    MOV     SUMS,R1
        ADD     ENTRY,R1
        MOV     R1,SUMS()
```

Notice that the third instruction would contain an error due to the missing register term which resulted from the missing third argument. Sometimes a missing argument may be used to advantage by altering the generation of the source statements with the conditional assembly statements. These statements (described in the next section) can detect the fact that the argument is missing and be used to omit portions of code selectively.

MACRO CALL ARGUMENTS

Normally the real arguments are separated by commas and the assembler expects this format. Also, leading and trailing blanks are ignored when processing each real argument in the macro call statement. Often it is desired to include a comma or blank as part of the real argument without having it act as a delimiter or be bypassed. Any argument that is enclosed in angle brackets will be passed onto the source code generation in total including any blanks and commas.

The macro call

```
        XPURT   ONE,TWO,THREE
```

has three real arguments while the call

```
        XPURT   <ONE,TWO,THREE>
```

has only one argument which includes the two commas and the call

```
        XPURT   <ONE,TWO>,THREE
```

has two real arguments of which the first includes one comma.

The system macro TYPE is another good example:

```
DEFINE  TYPE    MSG
        TTYI
        ASCII   /MSG/
        BYTE    0
        EVEN
        ENDM
```

This macro is one of the AM-100 supervisor calls and is designed to type out the ASCII message which appears as the argument to the TYPE macro call. The BYTE 0 statement insures a null terminator and the EVEN statement insures that the next instruction is again synchronized on a word boundary.

The call

```
        TYPE            HELLO
```

will type out the message "HELLO" because all the leading blanks are automatically ignored before the argument is processed.
The call

```
        TYPE    <       HELLO >
```

will type out the message "   HELLO " because the blanks are included in the argument as a result of the angle brackets.
Similarly, the call

```
        TYPE    HELLO, I AM A COMPUTER
```

will type out the message "HELLO" because the comma will terminate the

argument and the rest of it will be ignored.
The call

```
        TYPE    <HELLO, I AM A COMPUTER>
```

will type out the message "HELLO, I AM A COMPUTER" because the comma is included in the argument as a result of the angle brackets.
Get it? Got it? Good!

NESTED MACRO CALLS

Macro calls may be nested to a depth of 16 levels. A nested macro is defined as a macro call within the source statements generated by another macro call. Arguments may be passed to nested macros by naming the dummy arguments the same throughout the levels. Arguments that contain blanks or commas may be passed through successive levels by enclosing them in one set of angle brackets for each level of nesting since one set of angle brackets will be removed from an argument with each nesting level. For example, to pass the argument A,B through three levels of nested macro calls you would enter the argument as <<<A,B>>> in the first level macro call.

ARGUMENT CONCATENATION

Since dummy arguments must be valid user symbols the apostrophe (') will be a legal delimiter for any dummy argument within a macro definition source line. When an apostrophe immediately precedes and/or follows a dummy argument in the source text, the apostrophe is removed and the substitution of the real argument occurs at that point. This is useful for building symbols with arguments that are to be a part of that symbol.

Given the following macro definition and eventual calls

```
DEFINE  BUILD   AA,BB
TAG'AA: MOV     R1,Q'BB'7
        ENDM
         ..
         ..
        BUILD   RA,STS
        BUILD   T,P
```

the effective code generated by the two calls would be

```
TAGRA:  MOV     R1,QSTS7
TAGT:   MOV     R1,QP7
```

NCHR AND NTYPE

These two macro directives return a value which specify the number of characters in an argument (NCHR) or the addressing mode type of an argument (NTYPE). The NCHR and NTYPE statements function similarly to the equate statement (=) in that they assign a value to a user symbol which may be reassigned as many times as

desired during the course of the assembly. These statements are normally used to control the development of macro source code based on the size and type of arguments passed to the macro and therefore are defined in this section dealing with macros. In actuality they may be used anywhere in the source program with any valid source code as an argument but are fairly meaningless unless used within a macro.

The NCHR statement assigns a value to a user symbol which is equivalent to the number of characters in the argument string. It has the format:

```
NCHR    symbol,string
```

The NTYPE statement assigns a value to a user symbol which is equivalent to the 6-bit addressing mode of the argument. It has the format:

```
NTYPE   symbol,argument
```

The following is a list of the addressing modes and the values that they will deliver via the NTYPE statement. The upper case "R" represents any of the eight registers (R0-R5, SP, PC) which have a corresponding result value of 0-7 added to the resulting mode they are used in.

```
R       direct register delivers 0R
@R      indirect register delivers 1R
(R)+    autoincrement delivers 2R
@(R)+   indirect autoincrement delivers 3R
-(R)    autodecrement delivers 4R
@-(R)   indirect autoincrement delivers 5R
X(R)    indexed delivers 6R
@X(R)   indirect indexed delivers 7R
#X      immediate delivers 27
TAG     relative delivers 67
@TAG    indirect relative delivers 77
```

Once the symbol has been assigned a value by either the NCHR or NTYPE directives it may be used by itself or in expressions to control the development of the macro source code through the conditional assembly statements.

CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives allow the programmer to selectively include or bypass certain lines or segments of source code based on variable parameters which are tested during assembly. This allows several different versions of the same program to be generated from one source file. They also find their widest use within macro definitions where they are used to tailor the macro based on the real arguments used in the macro call.

Like the macro definitions, conditional directives take on two general forms. The normal form allows one or more lines of source code to be selected or bypassed based on the current status of a variable. The single-line form performs the same function but is a shorter version and only allows the control of a single line of source code.

The general form of a normal conditional block is:

```
IF      condition,argument
source line 1
source line 2
    ...
    ...
source line n
ENDC
```

The general form of a single-line conditional is:

```
IF      condition,argument, source line
```

Both forms employ the IF pseudo-opcode to identify the conditional directive and both forms require a condition code which specifies the type of test to be performed and an argument upon which to perform that test. The condition code is a symbol which identifies the test which is performed at the time the conditional is encountered during phase 1 of the assembly process. The argument may be a symbol, expression or macro argument depending on the type of test being performed.

Note that the indicating item that distinguishes the two forms is the comma that follows the argument in the single-line form. If the comma exists the remainder of the line up to and including the carriage-return and line-feed will be the source line that will either be assembled or bypassed depending on the result of the conditional test. If the comma does not exist the conditional assembly will be done on the source line which follows the conditional directive (IF) line up to but not including the ENDC terminating line.

CONDITION CODES

The following is a list of the condition codes that are legal and the type of condition that the associated argument is tested for. Unless otherwise specified, the argument is evaluated as an expression and the 16-bit result of that evaluation is the quantity that is tested to meet the condition. The conditional source lines are to be assembled if the argument meets the condition

listed next to the code below.

EQ	The argument is equal to zero
NE	The argument is not equal to zero
LT	The argument is less than zero
GT	The argument is greater than zero
LE	The argument is less than or equal to zero
GE	The argument is greater than or equal to zero
DF	The argument is completely defined at this point
NDF	The argument contains one or more undefined symbols at this point
B	The argument (a string of ASCII characters) is blank or null
NB	The argument (a string of ASCII characters) is not blank or null

SUBCONDITIONALS

There are three subconditional directives which allow the alteration of the normal conditional processing within a conditional block. These subconditionals (IFF, IFT and IFTF) require no other parameters and must be used within the source code that is between the IF and ENDC statements. The following functions may be performed through the proper use of subconditionals:

1. Assembly of an alternate block of code when the main conditional code is being bypassed due to a failed conditonal test.

2. Assembly of a noncontiguous body of code within the conditional block depending on the result of the main conditional test.

3. Unconditional assembly of a block of code within a conditional block regardless of the result of the conditional test.

The three subconditionals and their functions are:

IFF	The source lines following the IFF statement up to the next subconditional or end of main conditional are assembled if the main conditional test result was false.
IFT	The source lines following the IFT statement up to the next subconditional or end of main conditional are assembled if the main conditional test result was true.
IFTF	The source lines following the IFTF statement up to the next subconditional or end of main conditional are assembled regardless of the main conditional test result.

NESTING OF CONDITIONALS

Conditionals and subconditionals may be nested to a maximum depth of 16 levels. Any conditionals within a higher level conditional will be bypassed (the test will not be performed) if the result of the higher level conditional test was false. Subconditionals within outer level conditional blocks will be tested while those within inner level untested blocks will be ignored. Consider the following example (mundane as it may be):

```
TEST1:  IF      EQ,3-3          ;True so assemble following code
        WORD    33              ;Assembled since EQ,3-3 was true
        IF      NE,4-4            ;False so bypass following code
        WORD    44                ;Not assembled since NE,4-4 was false
        IFF                          ;Tested - true since NE,4-4 was false
        WORD    441                  ;Assembled since IFF was true
        IFT                          ;Tested - false since NE,4-4 was not true
        WORD    442                  ;Not assembled since IFT was false
        IFTF                         ;Tested - true regardless of NE,4-4
        WORD    443                  ;Assembled since IFTF was true
        ENDC                      ;End of NE,4-4 conditional block
        ENDC                    ;End of EQ,3-3 conditional block
TEST2:  IF      EQ,5-6          ;False so bypass following code
        WORD    56              ;Not assembled since EQ,5-6 was false
        IF      EQ,6-6            ;Not tested since EQ,5-6 was false
        WORD    61                ;Not assembled since EQ,6-6 was untested
        IFF                          ;Not tested since EQ,6-6 was untested
        WORD    661                  ;Not assembled since IFF was untested
        IFT                          ;Not tested since EQ,6-6 was untested
        WORD    662                  ;Not assembled since IFT was untested
        IFTF                         ;Not tested since EQ,6-6 was untested
        WORD    663                  ;Not assembled since IFTF was untested
        ENDC                      ;End of EQ,6-6 conditional block
        ENDC                    ;End of EQ,5-6 conditional block
```

The system macro for the PUSH convenience opcode is a good example of how conditionals may be used to control the code generated by a macro:

```
        DEFINE  PUSH    SRC
                IF      B,SRC, CLR -(SP)
                IF      NB,SRC, MOV SRC,-(SP)
                ENDM
```

If the macro is called without an argument (SRC is blank) then the first conditional is true and the code CLR -(SP) is generated to push a zero word onto the stack. The second conditional is therefore false and generates no code. If the macro is called with an argument (SRC is not blank) then the reverse happens and the code MOV SRC,-(SP) is generated with SRC being replaced by the real argument in the calling statement. This causes the SRC word to be pushed onto the stack.

The same PUSH macro could have been alternately coded using subconditionals:

```
DEFINE  PUSH    SRC
        IF      B,SRC
        CLR     -(SP)
        IFF
        MOV     SRC,-(SP)
        ENDC
        ENDM
```

For some more examples of conditionals used within macros, print out or inspect the system library SYS.MAC which defines all of the supervisor calls used by the AM-100 timesharing system. This file can be found on the system disk in area [7,7].

WRITING TOTALLY RELOCATABLE CODE

The AM-100 system not only supports but requires that all programs written for operation under control of the AMOS monitor be written in totally relocatable code. This means that a program may be loaded physically into memory at any location and it will run without modification. No addresses within the program ever need to be modified since all references to memory are made in relation to the current value of the program counter register (PC). The program may even be dynamically moved about in memory without modification as long as it is not currently active while it is being moved. The code is actually independent of its position in memory and therefore has often been referred to by other manufacturers as position independent code.

Writing relocatable code for the AM-100 system has been simplified by the incorporation of several instructions which make references to the current position of the program automatic. The load effective address (LEA) instruction may be used to calculate the current value of any relocatable address and load that current value into any register. The table referencing instructions (TJMP and TCALL) both use relative offsets to perform their functions as opposed to absolute or calculated addresses.

Due to the normally relocatable nature of the AM-100 instruction set and addressing modes, writing totally relocatable code merely involves obeying a few specific restrictions in the course of programming. The most important of these is to never refer to any absolute address in main memory unless you are sure of its location and contents. Two of the addressing modes will always generate absolute memory references and must be avoided when writing relocatable code. Note the following examples:

```
CLR     @#TAG
CLR     TAG(R4)
```

In the first example the absolute address of TAG is stored in immediate mode and then used to indirectly address that absolute memory location. This addressing mode is not relocatable unless the reference to TAG is a reference to a known absolute memory location. In the second example the very most common method of indexing can be shown to be non-relocatable. Normal indexing address schemes take the base of some area (in this case it is TAG) and add an offset from some calculation which is stored in an index register (in this case R4) to develop the target memory address. The value of TAG is stored in the instruction as an absolute value and no offset is ever added to compensate for relocation of the program. This mode would not be relocatable unless, as in the first example, the reference to TAG is to a known absolute memory location.

The two above addressing modes are the most commonly made errors that violate the rules for relocatable code. A more subtle mistake is made when a register is set up as an index to a table within the user program to be referenced later through the register. Take these examples:

```
MOV     #TABLE,R0
LEA     R0,TABLE
```

The first example stores the address of TABLE as an absolute value due to the

immediate mode addressing. Since the assembly of the program is done starting at location zero, the value of TABLE during assembly is really the offset from TABLE to the base of the program. When the program actually runs it will not be located at zero (the operating system resides in the first 12K or so) and the actual address of TABLE will not be the same as at assembly time. The second example is the proper instruction to be used when setting up a register to a memory reference. The instruction is coded at assembly time as an offset from the instruction itself to the location marked as TABLE and when the LEA instruction is executed, the actual value of TABLE in its current location is calculated and loaded into the register. This method awards two gold stars for intelligence.

Addressing modes which involve only register references are totally relocatable and also rate a gold star. These modes are listed here:

```
Rx      direct register
@Rx     indirect register
(Rx)+   autoincrement
@(Rx)+  indirect autoincrement
-(Rx)   autodecrement
@-(Rx)  indirect autodecrement
```

The two relative addressing modes are also relocatable:

```
TAG     relative
@TAG    indirect relative
```

Index modes can be relocatable or non-relocatable depending on their usage and set up procedure. Generally speaking, if the register is absolute and the index offset is a relative tag in the program the indexing is not relocatable and will deliver wrong results. If the register is first loaded with the effective value of the relative address within the program and the index offset is the absolute component then the scheme is relocatable and will give the desired results. Take the following two examples of clearing the third word (sixth byte) in TABLE:

This is the wrong way:

```
MOVI    6,R3            ;R3 GETS ABSOLUTE COMPONENT OFFSET
CLR     TABLE(R3)       ;ABSOLUTE LOCATION TABLE(R3) IS CLEARED
```

This is the right way (two gold stars):

```
LEA     R3,TABLE        ;R3 GETS CURRENT ADDRESS OF TABLE IN PROGRAM
CLR     6(R3)           ;RELOCATABLE LOCATION AT TABLE+6 IS CLEARED
```

In summary, the best way to learn how to evaluate the reloacatblity of a particular programming technique is to become thoroughly familiar with the addressing modes used by the WD16 microcomputer and the type of code that they generate. This information can be found in chapter two of the WD16 MICROCOMPUTER PROGRAMMER'S REFERENCE MANUAL. Soon you will be whizzing along writing totally relocatable code in your sleep, on your way to work, in the shower, etc.

After your first futile ordeal with writing the source code you may get brave and actually attempt to assemble it. Some instructions on the operation of the assembler may be in order to accomplish this in a reasonably sane manner. The assembler actually runs in five distinct phases which are selectively called depending on functions desired. A brief summary of their respective functions follows.

PHASE 0 - interprets the command line and sets up parameters in the common area for use by successive phases.

PHASE 1 - reads the source file (MAC) and performs pass 1 of a standard two pass assembly process by expanding macros, building the user symbol table, and generating the interphase work file (IPF).

PHASE 2 - reads the interphase file (IPF) and performs pass 2 of a standard two pass assembly process by resolving symbols and generating the object code file (OBJ). The interphase file is deleted.

PHASE 3 - reads the source file (MAC) and the object file (OBJ) and creates a list file (LST) or outputs to the terminal.

PHASE 4 - actually not part of the assembler but an automatic call to the LINK program to read the object file (OBJ) and create a runnable program file (PRG). Only occurs if there were no internal or external references in the program.

The general format for the assembler command line is

MACRO filename {/switches}

Filename is the name of the source program to be assembled or listed and may optionally be a complete file specification as in other system programs. The /switches option is comprised of a slash followed by one or more alphabetic characters which alter the normal assembly process. If no switches are entered an assembly is performed on the specified source file and an object file is created but no list file (phase 3 is bypassed). If the program is single segment (no INTERN or EXTERN statements) then phase 4 is entered which creates a program file.

The action of the assembler option switches are listed below:

/B generates a bottom header line on every page using the rest of the command line as title information and if used, must be the last switch on the line.

/C includes conditionals on listing which are normally suppressed

/E lists only those lines which contained an error

/H lists binary code in hexadecimal instead of octal

/L generates a list file by calling phase 3 during assembly

/O uses current object file by omitting phases 1 and 2

/T prints the listing on the user terminal instead of disk file

/X lists all macro expansions which are normally suppressed

Any of the above switches may be combined as desired in a single command line. The most common method of assembling new programs is as follows:

1. Assemble the program with the command: MACRO filename
   This will count any errors that occurred during phases 1 and 2

2. If no errors occurred create a list file with: MACRO filename /LO
   or optionally list it on the terminal with: MACRO filename /TO

3. If there were errors list them alone with: MACRO filename/TOE

4. Correct the errors and go back to step 1

5. If the program has only one segment then phase 4 has already created a program file otherwise refer to the LINK operations section of this manual to create one

ASSEMBLY LISTING FORMAT

Phase 3 of the assembly process creates a list file on disk or optionally outputs to the user terminal. The listing is formatted and contains the resultant binary code that is generated. Each page contains a title and page number giving the name of the program that has been assembled and the account number that the file was assembled in. Unless otherwise controlled by PAGE statements, each page contains 54 lines of source data. Each page is terminated by a form-feed character to perform the page function on most line printers.

Each data line on the listing contains four sections:

1. Error codes are listed in columns 1-5 if any occurred on this line

2. The current address of the generated data is listed in columns 8-13 if any code was generated or the value of the assignment if this is an equate statement

3. The generated binary data (maximum of the first three words) is listed in columns 16-37 in octal (or hex if /H switch used)

4. The source line is listed in columns 40-132

The error codes and their meanings are listed below:

A	branch address was out of the 127-word range
B	boundary error - a word operand was on an odd byte address
C	conditional statement syntax error
D	duplicate user symbol
I	illegal character in source line
M	missing term or operator in operand or expression
N	numeric error which indicates a digit out of the current radix range
P	an expression that had to be resolvable on the first pass was not
Q	questionable syntax - this is a general catch-all error code
R	register error - a register expression was not in the range of 0-7
T	source line or operand terminated improperly
U	undefined user symbol during pass 2
V	value of an absolute parameter was out of its defined range
X	assembler system error - please notify Alpha Microsystems

Phase 3 generates the list file by reading the source file and the object file and synchronizing the source lines with the resolved object data to come up with the listing line data. If these two files get out of sync there is no way that the listing may proceed and the message [SYNC ERROR] will appear on the user terminal. The list file will then be closed at the point of the sync error but the line that caused the error will not have been included. A sync error of this sort means one of two things: either you have an out-of-date object file that you are using with the /O switch or you have found one of several known but undiagnosed assembler bugs. These bugs usually occur when you get fancy with nested macros and conditionals and have a valid error buried down deep within. The assembler just seems to get tied up in knots for no apparent reason. Temporary action on your part would consist of trying to locate the reason for the error in your source file and trying again. These bugs will be fixed as fast as they can be pinpointed.

LINKAGE EDITOR OPERATION

The output of the assembler itself is not in a form that is directly usable by the operating system for running the program. It is an object file (OBJ) which is not fully resolved and contains symbol definitions and imbedded cross-segment commands. This object file is resolved by the linkage editor (LINK) which reads one or more of these object files and creates one runnable program file (PRG) which the operating system may load into memory and run. Furthermore, if the program contains overlay segments, LINK will resolve them and create one overlay file (OVR) for each one. These overlay files are loaded into memory upon command during the running of the program and allow memory conservation for large programs such as the assembler itself.

Referring to the previous section on the assembler operation you will note that if the program had only one segment the linkage editor was called automatically to create a program file. In this case no further action is necessary and you are ready to run the program. If, however, the program is comprised of more than one segment you must run the LINK program and specify the name and order of the segments involved.

The general format of the LINK command is:

```
LINK file1,file2,file3,......file-n
```

LINK will read each of the files specified and create the necessary program and optional overlay files. Errors will be listed on the terminal if any are encountered during processing. The most common error is the undefined global symbol error which means you have an EXTERN symbol in one segment which is not defined in another segment by an INTERN statement. No program file will be generated (and no gold star awarded) if one or more of the segments is not found in its assembled object form (OBJ).

If the program contains more files than will fit on the command line you may continue the files on the next line by terminating the last filename with a comma. LINK will continue to accept files as long as the last file on the line terminates with a comma.

SECTION 2

OPERATION AND USAGE OF AM-100 MONITOR CALLS

COMMUNICATING WITH THE AM-100 MONITOR

The AM-100 monitor contains over 70 routines which are available for use by assembly language programs running in user or monitor memory. These routines are called by the supervisor calls SVCA and SVCB which have been coded into macro form to make them easy to incorporate into user programs. The macros are included as a part of the system library file SYS.MAC in area 7,7 of the system disk. These calls have been grouped into logical sections according to the function they perform and will be described in this and the following sections.

The general format for all monitor calls is:

{label:} opcode {arguments} {;comments}

As can be seen from the format, the only required item in all calls is the opcode itself which is the name of the monitor call. A label may be used if desired in which case it is assigned the address of the SVCA or SVCB instructions which start all monitor call sequences. The total number of words generated by any monitor call depends upon the call itself. Some call generate up to four words of code to perform the function. Those calls which incorporate an ASCII message (such as the TYPE call) will generate a string of bytes which will vary in length depending on the message involved. Comments may also be placed at the end of the line as desired just as in the machine instructions and will be preceded by the semicolon which is the identifying character for all comments.

Some calls will require one or more arguments which specify parameters for the execution of the monitor call function. These arguments most normally are source and/or destination address items for the data being manipulated by the monitor call. Some calls allow the user to specify the location of data parameters while other calls operate with predefined registers that must be set up by the user beforehand. As each call is defined in the sections that follow the arguments that are required will be detailed. The arguments will normally be defined as being expression values, standard addresses, or ASCII strings. Expression values may be any valid source expression which evaluates down to a value which is within the predefined range of the argument definition. ASCII strings are just that; a string of characters which are probably used as a message to be displayed. Standard addresses are so important and complex enough that we will devote the next whole section to the explanation of them.

STANDARD ADDRESS ARGUMENTS

Standard addresses form the heart of many of the more complex monitor calls and therefore should be thoroughly understood by the user in order to gain maximum flexibility from the system. A standard address argument is coded exactly the same as a standard source or destination operand for a machine instruction such as ADD or MOV. There are some restrictions that should be noted, however, due to the method used in processing the standard address. Standard addresses are only used with those monitor calls that are coded as SVCB instructions. The SVCB pushes all user registers onto the stack and it is from these stored values on the stack that the monitor call processor gains access to the address calculations using those registers. Standard addresses may take the form of any of the legal WD16 addressing modes however all autoincrement and autodecrement processing is done on a word basis even though the monitor call may be requesting only one byte of data. In addition, the value used for SP register references is a dummy value which is not reloaded into SP when the monitor call exits so the autoincrementing and autodecrementing will be ignored if used with the stack pointer register.

The monitor call processing software within the monitor actually duplicates the hardware and calculates the target address from the stored register value on the stack and the data from the extra word if the address mode uses one. This target address then becomes the address of the data to be manipulated by the specific monitor call routine itself. Sometimes this data is only one byte while other times it may be several words or more. The target address calculated by the processing of the standard address argument always points to the first byte of the data if more than one byte is required by the monitor call. A special case occurs when the standard address argument specifies the direct register address mode. In the WD16 hardware instructions there is never more than one full word of data involved for the standard source and destination address modes so direct register will work on whether the low byte or the full word in the target register. In the processing of monitor call standard addresses, however, this is not always the case since we pointed out that some calls require several words of data to be manipulated. When direct register mode is used the target address is actually the address of the stored register on the stack which was a direct result of the SVCB hardware instruction processing. If more than one word is used by the call it will merely sequence right on through the stored words on the stack. In simple terms this means that if a monitor call wants three words of data for an argument and you specify the register R2 as the standard address argument, the three words that will be used will actually be the words in R2, R3 and R4 in sequence. Pretty nifty, huh? A word of caution: if you specify a register for a call that wants more words than you have registers for (most I/O calls want a 20-word DDB argument) the monitor call will walk right on through your stack and possibly blow the whole silly system up in your face!

At this point we should pause to allow the smoke to clear. Perhaps a beer might help you digest the above nonsense before we go on to more frivolous things. Just try to think of the standard address arguments as good old source or destination addresses like in the machine instructions. When you foul them up you will definitely find out about it quick.

ALPHABETICAL LISTING OF AM-100 MONITOR CALLS

The following is a quick reference to all AM-100 monitor calls

ALF	tests the character indexed by R2 for alphabetic
ASSIGN	assigns a non-sharable device to a job
BYP	bypasses all spaces and tabs in the string indexed by R2
CHGMEM	changes the size of a user memory module
CLOSE	closes a logical dataset
CRLF	prints a carriage-return line-feed pair on the user terminal
DCVT	converts a binary value to decimal and prints it on the user terminal
DEASGN	deassigns a non-sharable device from a job
DELETE	deletes a file from a file-structured device
DELMEM	deletes a user memory module from his partition
DSKALC	allocates next available record on disk and returns block number
DSKBMR	reads disk bitmap and sets reentrant lock for user modification
DSKBMW	rewrites disk bitmap after user modification
DSKCTG	allocates a contiguous file for random processing
DSKDEA	deallocates a record on disk and makes it available for use again
DSKDRL	sets reentrant directory lock for a specific user's directory
DSKDRU	clears reentrant directory lock for a specific user's directory
EXIT	exits from user program and returns to monitor command mode
FETCH	fetches a module from disk into user memory unless already in memory
FILNAM	processes a filename specification indexed by R2 into RAD50 format
FSPEC	processes a complete file specification indexed by R2 and sets up DDB
GETMEM	allocates a user memory module in his partition
GTDEC	converts a decimal number indexed by R2 into binary and returns it in R1
GTOCT	converts an octal number indexed by R2 into binary and returns it in R1
GTPPN	converts a PPN format indexed by R2 into binary and returns it in R1
INIT	initializes a dataset driver block (DDB) for I/O processing
INPUT	performs a logical record input I/O function on an open dataset
IODQ	dequeues a device driver transfer function after completion
JOBGET	retrieves a job control block item for the current job
JOBIDX	set an index to a job control block item for the current job
JOBSET	sets data into a job control block item for the current job
KBD	accepts input from user terminal keyboard (character or line mode)
LIN	tests the character indexed by R2 for valid end-of-line character
LOCK	locks the processor against interrupts (perfoms IDS instruction)
LOOKUP	looks for a specific file on disk and returns information about it
NUM	tests the character indexed by R2 for numeric
OCVT	converts a binary value to octal and prints it on the user terminal
OPEN	general form of the I/O logical dataset open calls
OPENI	opens a logical dataset for input
OPENO	opens a logical dataset for output
OPENR	opens a logical dataset for random access
OUTPUT	performs a logical record output I/O function on an open dataset
PACK	packs an ASCII triplet into its RAD50 code
PFILE	prints a complete file specification on user terminal from a DDB
PRNAM	prints a filename specification on user terminal from its packed format
PRPPN	prints a PPN specification on user terminal from its packed format
PTYIN	forces one character into another job's terminal input buffer
PTYOUT	retrieves one character from another job's terminal output buffer
READ	performs a physical record read I/O function on a dataset

RENAME	renames a file on a file-structured device
SCHED	forces a job scheduling sequence
SLEEP	puts the user job to sleep for a specified number of line clock ticks
SRCH	searches for a named memory module and returns its address
TAB	sends a tab character to the user terminal
TIDX	sets R0 to index the base of the user terminal line table in the JCB
TIN	reads one character from the user terminal input buffer
TOUT	sends one character to the user terminal output buffer
TRM	tests the character indexed by R2 for a valid termination character
TTY	outputs one character to the user terminal
TTYI	outputs an inline message to the user terminal
TTYIN	retrieves one character from any job's terminal input buffer
TTYL	outputs a message to the user terminal
TTYOUT	forces one character into any job's output buffer
TYPE	types an ASCII message on the user terminal
TYPECR	types an ASCII message on the user terminal with appended CRLF pair
TYPESP	types an ASCII message on the user terminal with one appended space
UNLOCK	unlocks the processor for interrupts (performs IEN instruction)
UNPACK	unpacks a RAD50 code word into its equivalend ASCII triplet
USRBAS	returns the address of the current user's memory partition base
USREND	returns the address of the current user's memory partition end
USRFRE	returns the address of the current user's free memory area
WAIT	puts user job into a wait state until the completion of I/O
WRITE	performs a physical record write I/O function on a dataset

JOB SCHEDULING AND CONTROL SYSTEM

The AM-100 timesharing monitor performs the task of allocating jobs and scheduling CPU time and resources for their operation. In order to properly write assembly language programs which make use of some of the more complex features of the system, a basic understanding of how jobs are scheduled and controlled is necessary. The total theory of how jobs are handled is too great a task to attempt to cover in one section of this manual but it is hoped that enough information can be given to provide the basic fundamentals on job control by user programs.

Each job that is running in the system basically has two dedicated components which are not shared by any other job in the system; a monitor job control block and a user memory partition. In the monitor memory area itself there exists a job control table which contains one area for each job that has been allocated to the system. One job is allocated for each JOB command in the system initialization command file which gives the job name and the terminal to which it is connected. The area that is allocated for each job in the job control table is used to contain specific information about that job. This area is called the job control block and will be referred to from now on as the JCB.

The format of the JCB is defined in the system library file SYS.MAC as a series of equate statements. Each equate statement has the name JOBxxx where xxx is a three-character code for the specific item being defined. The value of this symbol is actually the offset in bytes from the base of the JCB to the item itself. The user may, during the course of his program, desire to read the current data in his own JCB or in some instances modify it. References to the JCB items should be made in one of two ways:

1. Use the system monitor calls JOBGET, JOBSET and JOBIDX which is the preferred method.

2. Locate the JCB for your job by moving @#JOBCUR into a register and then referencing all JCB items via JOBxxx(Rx).

There are three words in the system communication area which define the entire job control system during timesharing operation. These three words are not part of the JCB areas but rather are non-sharable parameters set up during system initialization and not part of any one job. I point this fact out because the names of these three words are JOBTBL, JOBCUR and JOBESZ which appear to be part of a user JCB but really are not. JOBTBL contains the base of the JCB table where all JCB's are stacked sequentially. This address is set up at system initialization time and is never changed. JOBCUR always contains the address of the JCB which has current control of the CPU and is updated to point to the new JCB each time the job scheduler switches to a different job. Therefore, @#JOBCUR will always point to your JCB if you reference it because the reference will only be executed while you have current control of the CPU. JOBESZ contains the size of the JCB in bytes and is used by the system and by user programs for scanning through the JCB table. Since the size of the JCB may expand as new features are added to the system, JCB table scans must be made by setting an index to the base of the table (MOV @#JOBCUR,Rx) and then adding the size to the index to get to the next entry (ADD @#JOBESZ,Rx). When scanning the JCB table, the first word of each JCB is guaranteed to be non-zero and the table

is terminated by a null (zero) word. Again, these three words are a part of the master system communication area and not in the job table itself.

The following is a brief example of how to scan the JCB table and process each JCB entry (such as for a system status report):

```
        MOV     @#JOBTBL,R0     ;SET JCB TABLE INDEX R0 TO TABLE BASE
;LOOP HERE TO PROCESS EACH JOB TABLE ENTRY (JCB)
LOOP:   ...     ...             ;PROCESS THE JCB ENTRY WHICH IS INDEXED BY R0
        ...     ...             ;REFERENCES TO JCB ITEMS ARE VIA JOBxxx(R0)
        ...     ...
        ADD     @#JOBESZ,R0     ;ADVANCE R0 TO NEXT JCB ENTRY
        TST     @R0             ;IS THIS END OF JCB TABLE?  (NULL WORD)
        BNE     LOOP            ;  NOPE - GO PROCESS VALID JCB ENTRY
;AT THIS POINT WE HAVE FINISHED THE JOB TABLE SCAN
        ...     ...
        ...     ...
```

MONITOR CALLS

There are three monitor calls which should be used to gain access to your own JCB when necessary. Two calls are used to transfer a single word of data to and from a specific word in the JCB and one call is used to set an index to a specific spot in the JCB area so that multiple words may be transferred or so that faster access may be obtained when needed.

```
        JOBGET  tag,item        ;Transfers one word from JCB item to tag
        JOBSET  tag,item        ;Transfers one word from tag to JCB item
        JOBIDX  tag,item        ;Sets absolute address of JCB item into tag
```

All calls share the same basic format where tag is a standard argument used for the transfer of one word of data in the JOBGET and JOBSET calls or to receive the index address in the JOBIDX call. The item argument is one of the JCB item tags (JOBSTS, JOBPRI, JOBNAM, etc) which identifies the desired item to be used in the transfer or to have the index set to. These items are equated to their relative offset value in SYS.MAC and will be explained in the remainder of this section along with the usage of each and its importance to the user, if any.

JOBSTS

The first word in each JCB is the job status flag word. Each bit in this word indicates a particular state in which the job may reside. Some legal states are defined by more than one bit being on at a time. The system and some of the system programs set and reset these bits as the current state of the job changes but the user is cautioned against altering this word unless extreme caution (and intelligence) is used. A brief list of the bits and the mneumonics assigned to them follows along with a basic description of the function of the bit when it is set.

```
        J.ALC=1         ;JOB ENTRY IS ALLOCATED (GUARANTEES JOBSTS NON-ZERO)
        J.TIW=2         ;JOB IS IN TERMINAL INPUT WAIT STATE
        J.TOW=4         ;JOB IS IN TERMINAL OUTPUT WAIT STATE
```

```
J.SLP=10        ;JOB IS IN SLEEP STATE (HOHUM!)
J.IOW=20        ;JOB IS IN I/O WAIT STATE
J.EXW=40        ;JOB IS IN EXTERNAL EVENT WAIT STATE
J.CCC=100       ;A CONTROL-C ABORT IS WAITING TO BE PROCESSED
J.DLK=200       ;DISK LOCK FLAG TO STALL OFF J.CCC UNTIL DISK DONE
J.RUN=400       ;JOB IS RUNNING
J.MON=1000      ;JOB IS IN MONITOR COMMAND MODE (NO PROGRAM ACTIVE)
J.FGD=2000      ;JOB IS IN FOREGROUND WAIT STATE
J.LOD=4000      ;PROGRAM IS BEING LOADED FOR EXECUTION
J.SUS=10000     ;JOB IS IN SUSPEND STATE
J.LOK=20000     ;JOB HAS CPU LOCKED (BY USER PROGRAM COMMAND)
```

If any of the following flags are on the job will not be scheduled for CPU run time until the flag has been cleared: J.TIW, J.TOW, J.SLP, J.IOW, J.EXW, J.FGD, or J.SUS

JOBPRI

One word which is used for software priority control of jobs. The high byte contains the priority assigned to the job (by SETPRI command) and the low byte is used by the job scheduler to count down the clock ticks when the job is running. The user may modify the high byte to change the job priority if desired as the SETPRI program does.

JOBSLP

One word which is loaded with the number of clock ticks to sleep when the job is put to sleep. The monitor SLEEP call loads the specified count into JOBSLP and then sets the J.SLP flag in the JOBSTS word which puts the job to beddy-by for that number of ticks. Caution - if J.SLP is set but JOBSLP is set to zero the job will sleep forever until J.SLP is reset by some other event or job.

JOBSPX

One word which is used to store the current value of the SP register when the job forfeits the CPU to another job during timesharing. The user must not alter this word under any circumstances.

JOBNAM

Two words which contain the six-character job name packed RAD50. This name is set up by the JOB command in the system initialization file. If a user program alters this word it is effectively altering the name of the job.

JOBBAS

One word which contains the base address of the user memory partition if one has been allocated for this job. This address is altered only by the MEMORY program which allocates and deallocates user memory partitions. The user is advised

against altering this address unless a thorough understanding of the memory allocation process is first attained.

JOBSIZ

One word which contains the size of the user memory partition in bytes if one hase been allocated for this job. This size word together with the above JOBBAS address word define the current user memory partition. JOBSIZ is altered only by the MEMORY program and the monitor command processor.

JOBUSR

One word which contains the current user PPN (account number) if the user is logged in. Zero indicates that no user is currently logged into this job. JOBUSR is modified by the LOG and LOGOFF programs and is tested by various protection schemes in the system to allow user access to files, etc.

JOBICP

One word which controls the handling of control-c interrupts while the job is running a user program. If JOBICP is a zero the job will be aborted if a control-c is typed. This is normal mode for most programs and JOBICP is set to zero by the command processor just before a new program is initiated. The user program may intercept the abort procedure by setting an address into JOBICP. If the address is even the monitor will set the word specified by that address to all ones if a control-c is typed and the program will continue. It is then up to the program itself to continually test this word (not the JOBICP word but the one whose address is in JOBICP) and take appropriate action if it is found to be set to ones. If the address set into JOBICP is odd the monitor will strip off bit 0 to make it even and then execute a CALL instruction to the routine specified by that address. The routine must save any registers before using them and must return to the monitor via the RTN instruction once it has done its diabolical task. This method is complex and not recommended. AlphaBasic traps control-c interrupts by the first method and checks the flag each time a new source statement is about to be executed. This is why BASIC is never aborted by a control-c.

JOBPRG

Two words which contain the six-character program name which is currently running or was the last job run if in monitor command mode. JOBPRG is loaded with the program name (packed RAD50) by the command processor when the program is loaded or located for execution. Currently the only significance of this program name is in the displays created by the SYSTAT program (user terminal status display) and the DYSTAT program (video monitor).

JOBCMZ

One word which contains the size of the current command file area in the user memory partition if a command file is being processed. If this word is zero no command file is currently in effect. This word is set to the initial size of a command file when that file is loaded into the top of the user partition and is decreased as each line is extracted from the area and sent to the monitor command processor. When it gets to zero the command file is finished and the system returns to normal command mode input from the user terminal. The user should not alter this word.

JOBCMS

One word which contains flags used by the command file processor when a command file is being processed. These flags should never be altered by the user so they will not be detailed here. JOBCMS works in conjunction with JOBCMZ to effect the command file processing scheme.

JOBERC

One word which controls the processing of WD16 hardware bus errors as described in the WD16 PROGRAMMERS REFERENCE MANUAL. If JOBERC is zero a bus error will cause a message to be printed on the user terminal and the job will be aborted. If JOBERC is non-zero a jump will be made to the address specified in JOBERC which better contain a valid routine for gracefully shutting down the program. Note that the bus error is fatal for this user only and does not normally kill the whole timesharing system.

JOBTYP

One word which specifies the type of job which is assigned to this jobstream. Currently used only by the system, the user must not modify this word.

JOBBPT

One word which specifies the address to jump to if a breakpoint is encountered during the execution of a user program. JOBBPT is used by the DDT debug program for breakpoint handling and not normally used by user programs.

JOBUTR

One word which is normally zero but when loaded with an address, gives control to that address each time a character is entered on the user terminal. Details on the use of this scheme are somewhat complex and will be explained in a later section on terminal service programming.

JOBATT

One word which contains the address of the JCB that this job is attached to. If JOBATT is zero, this job is not attached to another job. A job which is attached to another job gets its terminal output rerouted to that job instead of having it output to its own terminal.

JOBDEV

One word which contains the RAD50 device code for the default device to be used if the file specification being processed by the FSPEC call does not specify an explicit device. Normally this default device is DSK.

JOBDRV

One word which contains the drive number in binary for the default drive number to be used if the file specification being processed by the FSPEC call does not specify an explicit drive number. Only used if the device code matches the code in JOBDEV or if the device code is left to deafult also. JOBDEV and JOBDRV normally contain the device and drive number set by the LOG program when a user logs in. They specify the disk device and drive which will be used for most processing of this user.

JOBCLK

One word which specifies the address of a user line clock routine but which is not currently in use by the AM-100 system. Don't mess with this one! (And don't ask why not!)

JOBTRM

An area which contains the terminal line parameters for this job. This area is currently 22 words in size and used extensively by the terminal service routines within the monitor for user terminal processing and echo control. The user is advised not to alter data within the JOBTRM area.

JOBKBF

A buffer (currently 84 characters in size) which is used to accept each terminal input line before delivering it to the monitor command processor or to the user program whenever a KBD call is executed. The KBD call accepts a full edited line of ASCII data into this buffer and then sets index R2 to the base of the buffer before returning to the user program.

JOBRBK

A 14-word area which is the run control block for the jobstream. It is used for the loading of programs and overlays during job execution and is set up by the user program with the parameters needed to fetch the next program or overlay segment prior to the execution of a FETCH call. Refer to the description of the FETCH monitor call for more details on the use of this item.

JOBFPE

One word which contains the address to jump to if a floating point error is executed such as a divide by zero. Used by AlphaBasic to control such nonsense. A user program which executes floating point instructions should enter his error trap address into JOBFPE and not into the vector at memory location 76 since this would destroy the sharable resource of that vector.

JOBDYS

One word which contains the address to the byte in the VDM screen memory area for the job execution arrow. Set by the DYSTAT program and referenced by the monitor job scheduler. The user should not alter this address.

JOBSTK

A 100-word area which acts as the stack for this job. SP is set to the top of this area when a new program is initiated. The user may reset his own stack pointer by moving the address of a larger area within his own partition if the program needs more stack area. Be sure to allow at least an extra 20 words or so for possible real-time interrupt handling which needs a valid stack area for register saves. The job scheduler also saves all user registers and some other nonsense on the user stack during job context switching.

MEMORY CONTROL SYSTEM CALLS

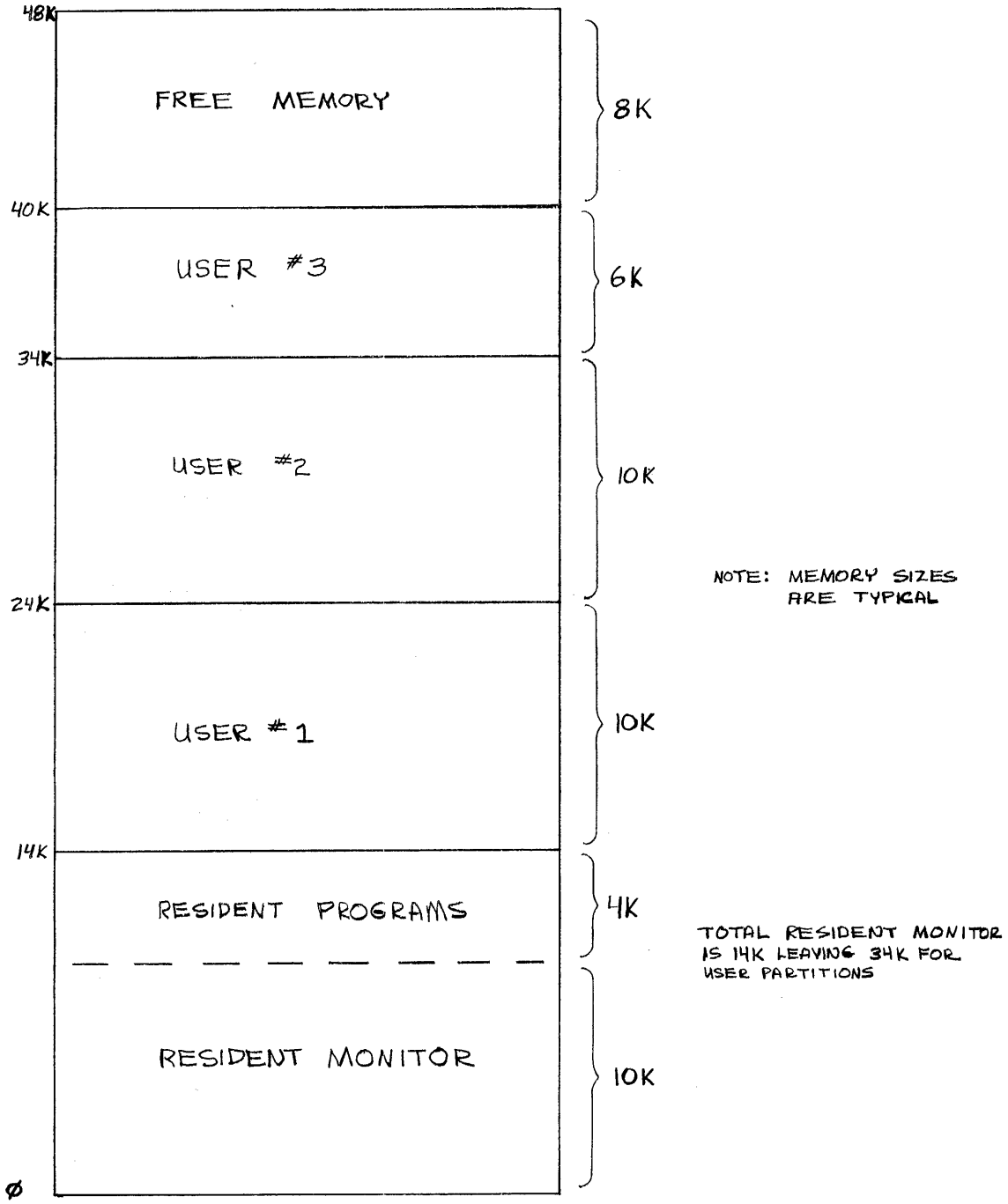
The AM-100 system contains a fairly sophisticated memory control system even though there is no memory protection or mapping hardware associated with it. In order to make maximum use of the memory resources available and minimize system crashes due to memory violations the assembly language programmer should understand how the monitor allocates memory and the rules under which memory should be accessed. This section will describe the memory allocation scheme and the monitor calls that assist the user in using memory in the proper way.

The memory that is available to the AM-100 processor is up to 65K bytes (32K words) with the top 256 bytes being unavailable because it is mapped to the I/O ports. The AMOS monitor resides in low memory beginning at location zero and extending upward as far as the monitor requires (typically around 14K bytes). The remaining memory above the monitor up to the end of the total amount of memory in your system is available for assignment as user memory partitions for each of the jobs. All of the user memory may be allocated to one job or it may be split up into several partitions of varying sizes with one partition allocated to each job. The amount of memory that a user program has to play with is therefore defined as the single contiguous memory partition which has been assigned to his job by the operator MEMORY command. This memory partition block is then allocated into smaller defined blocks called "modules" which are used by the system and the user to contain programs and data areas. Monitor calls exist which allow the user program to locate the absolute boundaries of his own memory partition and also to allocate, change, and delete memory segments in the form of defined modules. These modules have the capability of being named just like files (filename.extension) so they may be located by that name. Any program loaded for execution will be in the form of a module. During execution some programs create other modules for device buffers, data tables, etc.

MEMORY PARTITION FORMAT

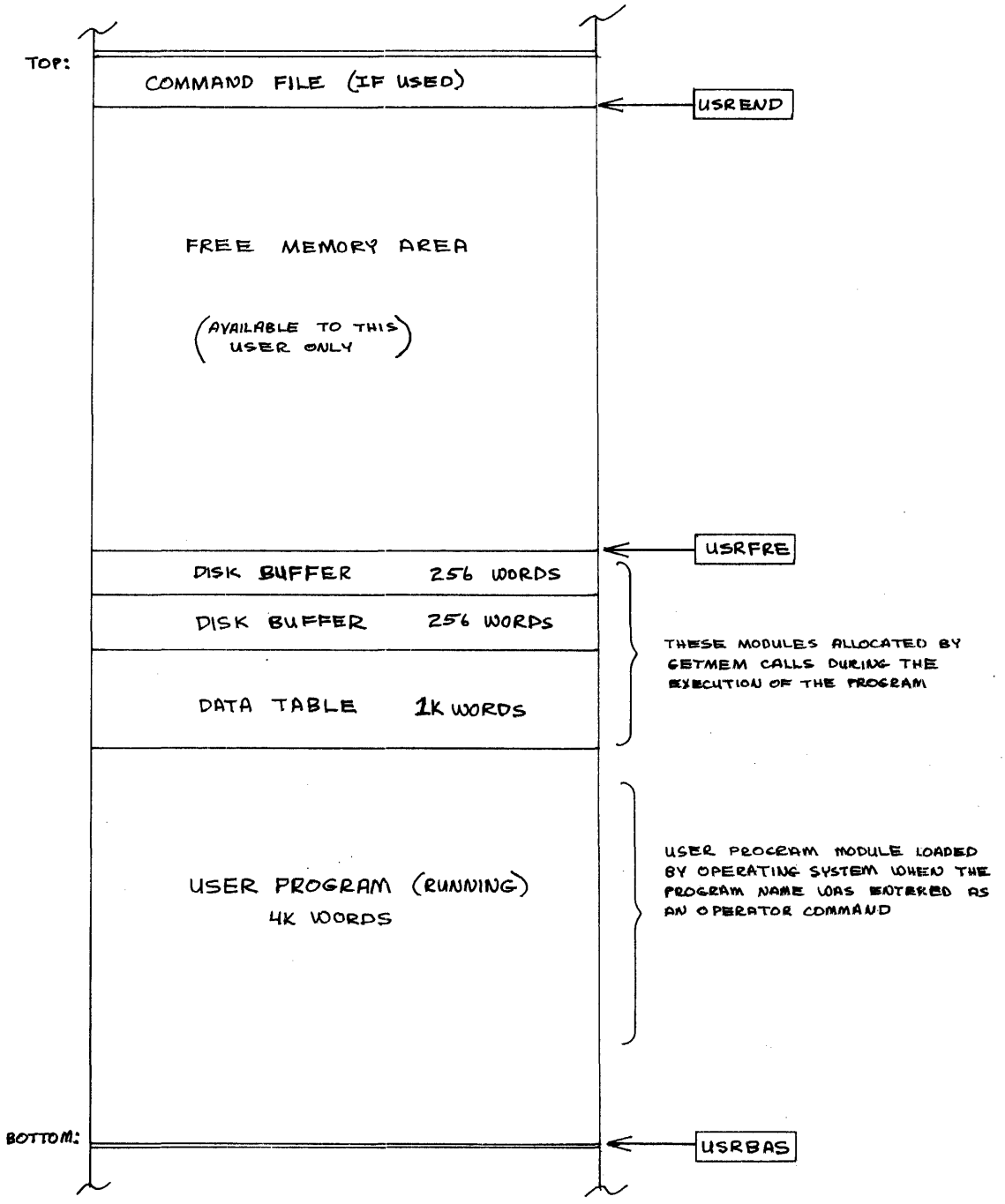
The memory partition assigned to a job may be located anywhere in memory depending on the memory that was available when the job assigned it using the MEMORY operator command program. The user program may not count on any specific location for this partition. Within the partition, memory modules are allocated upward beginning at the base of the defined partition and building modules on top of each other as long as space permits. Modules may not be built that will extend past the top boundary of the user partition. As modules are deleted from memory all modules above them are automatically shifted downward to fill up the space that the deleted module left. Also, when any module is changed in size the modules above it are shifted in position accordingly. This method insures that all available memory is always at the top of your partition in one contiguous block. This method of grabbing the first hunk of free memory to load a program into is the main reason that all programs must be written in totally relocatable code.

Figure M-1 shows a typical memory layout for three users operating in a 48K system. The free memory at the 40K boundary could be used by a fourth job or by one of the current jobs for expansion.



MEMORY MAP FOR A TYPICAL 48K SYSTEM (3 USERS)

FIG M-1



MEMORY MAP FOR A TYPICAL USER JOB PARTITION (11 K WORDS)

FIG M-2

There are three monitor calls that will return information about your memory partition as it happens to be allocated. These three calls all take a single standard argument into which will be delivered the absolute address of the base, end, or free base of the user memory partition. The three calls and the addresses that they return are listed below:

```
USRBAS  arg   - absolute base of user memory partition (last word)
USREND  arg   - absolute end of user memory partition (last word)
USRFRE  arg   - current base of remaining free memory (last module+2)
```

Since modules must always occupy an even number of bytes the above calls will always return an even address. If no modules are allocated in the current partition the USRFRE address will equal the USRBAS address. Otherwise, the USRFRE address will be the word following the last currently allocated module in the memory partition. The remaining free memory that the user may make use of may be calculated by subtracting the USRFRE address from the USREND address.

Figure M-2 shows a typical user job partition during the execution of a program which was loaded automatically by the operating system. The program itself was the first module to be allocated in the user partition and then was executed after being loaded. It will remain in memory until it has completed its task and exits to monitor at which time it will be deleted by the operating system monitor. During execution the program allocated a 1K data table module which possibly may be used for storage of symbols or some similar function. Two I/O files were then opened on disk which caused the operating system file service routine to allocate the two disk buffer modules. The remaining memory in the partition has not yet been allocated in our example.

Note that the USREND call does not actually return the absolute end of the partition but rather the end of the available free memory at the time of the call. If a command file is in progress it occupies the upper part of the partition which we do not wish to alter during the execution of a program. In fact, the program should not have to take into consideration whether or not it was called by direct command or from a command file. Use of the USREND call insures that the user program may use all of free memory without having to compensate for the remaining part of any command module.

Although the standard use of memory by the operating system is through the use of the memory management system calls (to be described next) the user may find that it is easier to make use of free memory without regard to module boundaries, especially for use in variable length tables or hashing techniques. For this reason, the free memory space is always defined as the area between the addresses returned by the USRFRE and USREND calls. Note that the initialization of files normally results in the allocation of a buffer module and the operating system allocates this buffer at the current setting of the USRFRE address and then updates that USRFRE address. Therefore, the user must be sure all I/O buffers and any work modules are allocated before freely using the memory above the USRFRE address. The INIT and FETCH calls both cause the indirect allocation of a memory module in addition to the direct allocation or alteration of modules by the GETMEM, CHGMEM and DELMEM calls.

MEMORY MODULE FORMAT

Memory modules are the basic unit of formal data structure within the user memory partition. They are always allocated on word boundaries and must contain an even number of bytes to maintain this format. The monitor calls will automatically pad an odd sized module with a null byte to even it up. All modules contain five housekeeping words followed by any number of data words from zero to the maximum size left in the user memory partition. The five housekeeping words are always allocated and so a single-word module really takes up six words of memory.

The module format is as follows:

Word 1 - total size of module in bytes including the housekeeping words
Word 2 - module flag word
Word 3 - module filename packed RAD50
Word 4 - module filename packed RAD50
Word 5 - module extension packed RAD50
Words 6 thru n - module data area

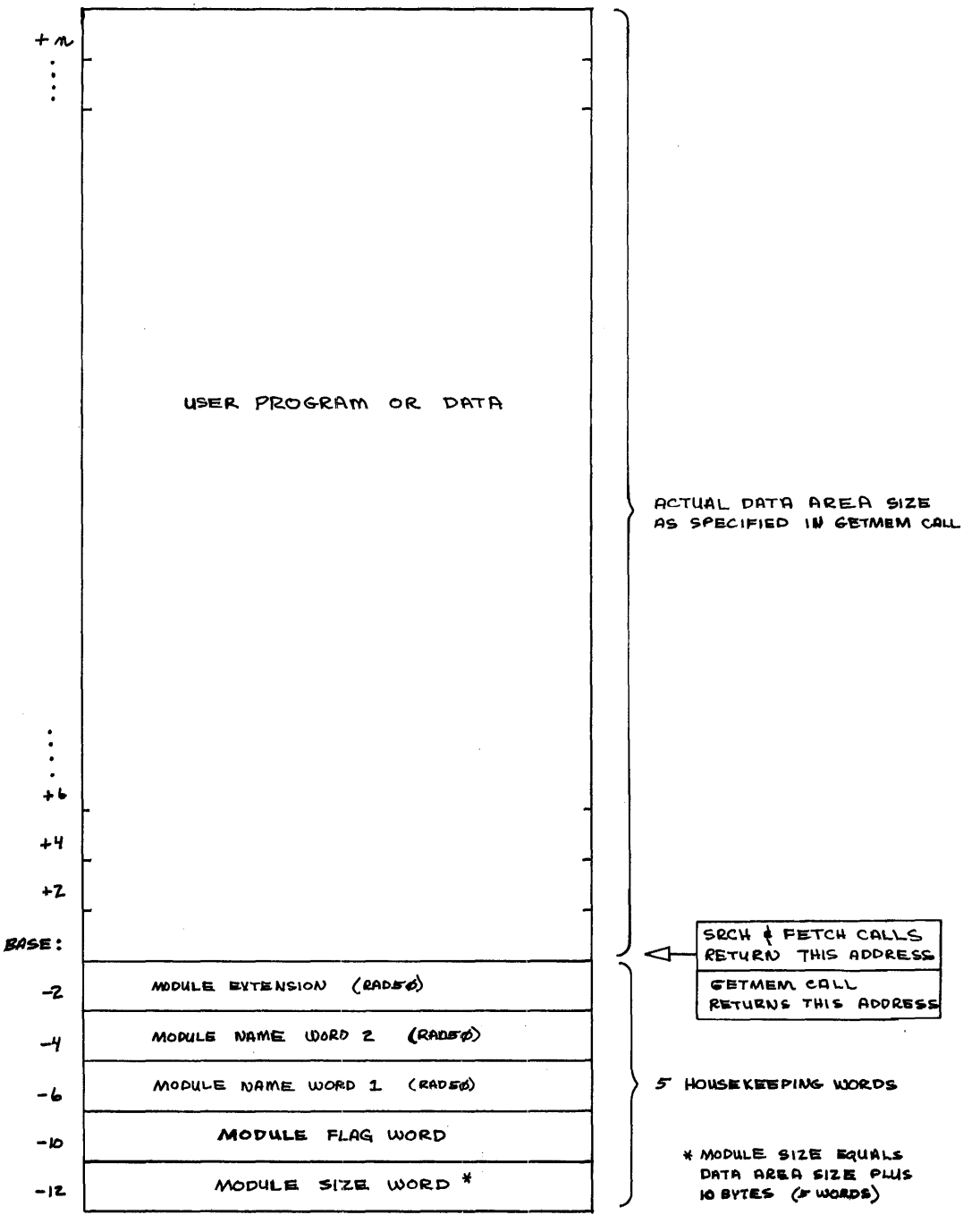
Figure M-3 gives a pictorial view of the above standard module format. The data area is usually the only area with which the user is concerned and so all references are made from the base of this area. The SRCH and FETCH calls (to be described in a later section) will return this absolute address when locating or loading the requested module instead of the address of the base of the housekeeping words. References to the housekeeping words should therefore be made via negative offsets relative to the data base address.

When scanning for a specific module or locating the end of the current module string, the user may set his index using the USRBAS call which will return the address of the size word of the first allocated module. He may then merely check the housekeeping words for the correct module name or other determining parameters and if the module is to be bypassed, add the size word to the index. This bumps the index to the next module allocated. The last module always has a zero word following it and the user must be careful not to destroy this zero word if he is manipulating free memory directly wihtout allocating it using the memory calls.

The module filename and extension follow the same format as the file names on disk if the module in memory is named. The name is optional and need be used only if the module is to be located by name at a later time.

Modules may be either temporary or permanent depending on the method used to load them into memory. A module is made permanent by setting the file bit on in the housekeeping flag word when the module is allocated. Temporary modules are automatically deleted by the monitor when the program finishes and executes the EXIT call. Permanent modules are not automatically deleted but may be deleted by either the operator DELETE command or the monitor DELMEM call. Forcing a zero into the size word of the module is another way of deleting it but this is not the recommended way since this effectively also deletes all modules above it (the zero is the module area termination word).

There are three monitor calls which are used to create, alter and delete these crazy memory modules. All three calls take a single standard argument which



STANDARD MEMORY MODULE FORMAT

FIG M-3

must be the address of a two-word block called a memory control block (MCB). The first word of this MCB will contain the absolute memory address of the data area in the allocated module (past the housekeeping words). The second word will contain the size of the data area in bytes (ten bytes less than the total module size since the housekeeping words are not included). The MCB therefore is the user's block which defines a contiguous area in memory by its base address and size in bytes. The user need not be concerned with the housekeeping words unless he needs to access them directly for some exotic reason.

The following three calls are used to manipulate memory modules:

```
        GETMEM  MCB     - allocates a new memory module at current USRFRE
        CHGMEM  MCB     - changes the size of the module defined by MCB
        DELMEM  MCB     - deletes the memory module defined by MCB
```

The following example shows the allocation of a 100 byte module

```
        MOV     #100.,MCB+2     ;SET MODULE SIZE AS 100 (DECIMAL) BYTES
        GETMEM  MCB             ;ALLOCATE MODULE (MCB GETS ITS ADDRESS)
        ...
        ...
        ...
MCB:    WORD    0               ;RECEIVES ADDRESS OF MODULE DATA AREA
        WORD    0               ;SIZE OF MODULE DATA AREA IN BYTES
```

Then we may increase the size of the same module by

```
        ADD     #20.,MCB+2      ;INCREASE SIZE WORD BY 20 BYTES
        CHGMEM  MCB             ;CHANGE ITS SIZE
```

The above will cause the monitor to adjust the module housekeeping size word to reflect the new size. The address of the module will not change but note that the USRFRE address will advance by 20 bytes and that any modules that were allocated after the one at MCB will be shifted up in memory but their corresponding addresses in their MCB will not be adjusted by the monitor. I/O buffers allocated after the MCB module will therefore be erroneously addressed after the change so the CHGMEM call must be used with care.

To delete the above module we use the code

```
        DELMEM  MCB             ;DELETE THE MODULE
```

Recall that all temporary modules will automatically be deleted by the monitor when the program exits. The user may force the module to be permanently left in memory by giving it a name and setting the file bit (defined in SYS.MAC as "FIL") in the flag word. The following example will illustrate the allocation of a 200 word module which is made permanent with the name "TABLE1.DAT":

```
        MOV     #200.,TBL1+2    ;SET SIZE AS 200 BYTES
        GETMEM  TBL1            ;ALLOCATE THE MODULE
        MOV     TBL1,R0         ;SET R0 TO INDEX THE DATA AREA BASE
        MOV     #[DAT],-(R0)    ;SET THE MODULE NAME AND EXTENSION (RAD50)
        MOV     #[LE1],-(R0)    ;  INTO THE HOUSEKEEPING WORDS
        MOV     #[TAB],-(R0)    ;  IN REVERSE ORDER FOR EFFICIENT USE OF R0
```

```
        BIS     #FIL,-(R0)      ;SET PERMANENT FILE BIT ON IN FLAG WORD
        ...
        ...
        ...
TBL1:   WORD    0               ;RECEIVES ADDRESS OF MODULE
        WORD    0               ;SIZE OF MODULE IN BYTES
```

Permanent memory modules may be saved onto disk using the operator SAVE command or they may be deleted from memory when done by the operator DEL command. Refer to the AMOS OPERATOR'S MANUAL for details on these commands.

LOADING AND LOCATING MEMORY MODULES

Memory modules may optionally have a filename and extension associated with them by which they may be located. Normally, when an operator command is entered from the terminal, the first place searched for locating the requested program is in the resident system memory area and then the user's own memory partition. If the program is resident in either of these places it need not be loaded in from disk and execution begins immediately using the resident program directly where located. The user may make use of two monitor calls (FETCH and SRCH) for loading and locating modules in memory by name. In actuality, the SRCH call is a specialized version of the FETCH call and is included only for convenience and compatibility with older programs that are still hanging around in the system. Basically, the SRCH call will only locate a module if it is in memory while the FETCH call will automatically load a module into memory from the disk if it is not found to be in memory already.

Both calls have the same basic format:

```
SRCH    nameblock,index,control-flags
FETCH   nameblock,index,control-flags
```

Nameblock is a standard argument used in both cases to specify the name of the module to be located or loaded. The format of the actual nameblock referenced is different in each case, however. In the case of the SRCH call, nameblock refers to a 3-word block of memory (or 3 contiguous registers) which contain the filename and extension of the desired module in RAD50 packed form. For the FETCH call, nameblock refers to a full file Dataset Driver Block (DDB) which allows the user to specify a full disk file specification to load the module from in case it is not located in memory. The DDB has not yet been introduced and will be defined and explained in a later section dealing with file I/O calls. In brief, the DDB is a 24-word (octal) area in memory which contains all the information and work areas to define and manipulate a specific disk file in any area on any defined disk device. The DDB is normally set up by processing an ASCII file specification with the FSPEC call (more on this later).

The second argument is the index which is to receive the absolute memory address of the located (or loaded) memory module data area. Refer to figure M-3 in the preceding section for the layout of the memory module and the place that this index is set to. The index argument is also a standard argument although the normal mode is to receive the module address in a general register (R0-R5). If the index argument is not specified in the call, the default used is register R0 which is compatible with older versions of this system.

The third argument is the optional control flags which may be used to control the operation of the SRCH and FETCH calls. This argument is any valid expression which evaluates down to a value in the range of 0-17 (octal). Only the low order four bits are significant and they have been given the following mneumonic definitions in the system library SYS.MAC:

```
F.FCH=1         ;Fetch module from disk if not in memory
F.USR=2         ;Search user memory only
F.ABS=4         ;Load absolute segment from disk
F.FIL=10        ;Set module permanent file flag after load from disk
```

F.FCH is the flag that actually differentiates the SRCH call from the fetch call since they both technically are the same SVCB supervisor call. The SRCH call forces this bit off while the FETCH call forces this bit on. When set, the F.FCH bit causes the nameblock to be interpreted as a full file DDB and the module to be loaded from disk if not located in memory first. Since the use of this bit is controlled by specifying either SRCH or FETCH as the calling opcode, the user is not supposed to include this bit in the control-flags argument of his call.

F.USR is the flag used to specify bypassing the searching of the resident system memory area for the module and proceed directly to searching the user area only. This allows specific versions of modules to be loaded and used even though they may possibly be duplicated in the system memory area. This flag is not normally used by programs other than system software.

F.ABS when set forces a direct search to the disk for the requested module, bypassing all memory searches that would normally occur. The module is then loaded into memory at the absolute address specified by the index argument in the calling sequence. No housekeeping words are allocated and the first word of the module gets loaded into the first word specified by the index argument. Note that this form is the only time that the index argument is used to pass an address to the FETCH processor instead of being used to receive the address of the located module. The F.ABS form of the FETCH call is used to load program segment overlays but I am certain that some other exotic uses will be thought of by you all out there.

F.FIL is used to force the permanent file flag bit on in the module flag word after the module has been loaded from disk. The FETCH call always places the filename and extension into the housekeeping words 3-5 so even if the module is only temporary, it may still be located by name as long as the program which loaded it is still active. This is useful for dynamic loading of subprograms and/or data modules. Setting the F.FIL flag on in the control-flags argument means that the module will not be deleted from memory by the operating system when the calling program finally exits. The operator LOAD command uses this method to load a program into memory and leave it there to be called by name.

When the SRCH or FETCH call returns, the user must test the status of the Z-bit to see if the module was located or loaded successfully. If the Z-bit is set (tested by BEQ) the operation was successful. If the Z-bit is not set (tested by BNE) the module was either not located or would not fit into the remaining free memory within the user's partition. TOO BAD!

FILE SERVICE SYSTEM DESCRIPTION

The AMOS monitor has a simple yet powerful device independent files service system which relieves the programmer of the task of I/O coding for each device he wishes to have his program interface with. In addition to this device independence, the monitor contains all routines to manage the disk file system on a logical call basis. The programmer need not be concerned with the exact physical placement of files on the disk except in rare instances where the system software is being developed or tested. The monitor also contains an efficient means for the development of new device drivers to be incorporated into the system when unsupported devices must be interfaced. This section will give a general overview of the file service system and describe the Dataset Driver Block (abbreviated as DDB) which is the descriptor link for all I/O and file calls to the monitor.

All I/O operations and file operations are accomplished by monitor calls with reference to a DDB which defines the device or file being operated upon. Whether the operation is to a unit record device such as a printer or to a specific file within a file structured device such as disk will depend upon the parameters passed to the monitor through the referenced DDB. There is no limit to the number of devices or files that may be active at any given time but there must be one separate DDB for each device or file in use concurrently. There are no internal channel numbers or device numbers to limit the number of concurrently active devices or files. The general sequence of events for the complete processing of a device or file operation can be summed up as follows:

1. The DDB is set up with the defining parameters such as device name, drive number, filename and extension, project-programmer number, etc. This data normally comes from the processing of an ASCII file specification such as DSK1:FILTST.MAC[101,1] by an FSPEC call.

2. The I/O buffers are allocated either directly by the user program or by an INIT call referencing the DDB in use.

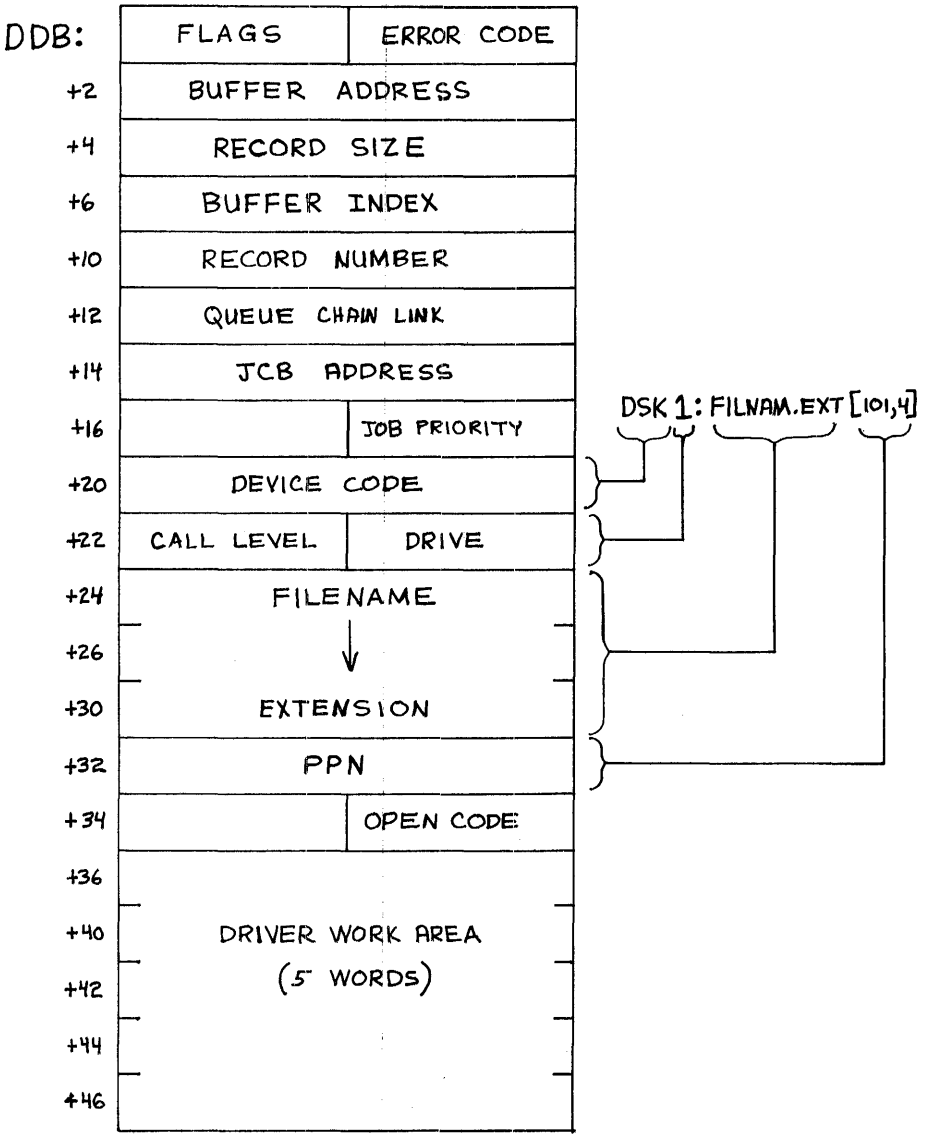
3. The logical opening processes for the device or file are performed which normally consists of an OPEN call referencing the DDB.

4. Data transfers to or from the device are performed by either READ and WRITE calls for physical transfers or INPUT and OUTPUT calls for logical transfers.

5. The logical closing processes for the device or file are performed which normally consists of a CLOSE call referencing the DDB.

6. The programmer breathes a sigh of relief if all went as planned.

The monitor contains complete error processing routines which allow the programmer to specify (by flags in word 1 of the DDB) whether or not any uncorrectable errors are to result in an automatic error message to the operator on his terminal, an aborting of the program and return to monitor, or both. The programmer may also elect to process the errors himself by checking the error code returned in word 1 of the DDB.



DATASET DRIVER BLOCK

FIG F-1

DDB FORMAT

Figure F-1 shows the format of the DDB which must be allocated within the user program area and set up by the user before any I/O operations can take place. The DDB is 24 octal words in size and is usually allocated by a BLKW 24 statement. The DDB can be assigned any tag which will then become the reference tag for all subsequent operations to that dataset. Some of the items in the DDB must be set up by the user before certain operations may be called for while other items are set up and used by the monitor file service routines. The item descriptions that follow will explain the use of each of these.

Error code - this byte is set to a non-zero code at the completion of an I/O operation that was unsuccessful for various reasons. A zero indicates the operation was successful. The user need test this byte only if the error control flag in the flags byte (DDB+1) specifies returning to the user on an error condition or if the operation allowed a non-fatal error condition to occur. The error codes are listed at the end of this section.

Flags - this byte is used to control the flow of the I/O operation and the handling of error codes by the file service routines. The following functions are controlled by the eight flag bits:

0 - set by user to force a return on error condition (abort if clear)
1 - set by user to bypass printing of error messages on error conditions
2 - real-time transfer flag (currently not implemented)
3 - spare
4 - transfer initiated (for internal file service use only)
5 - read if 0 or write if 1 (for internal file service use only)
6 - device inited - set by INIT call or by user if explicit buffer in use
7 - dataset busy (transfer initiated or queued)

Buffer address - this is the 16-bit absolute address of the base of the buffer to be used for all dataset transfers (read and write). It is set by the INIT call which allocates a buffer or it must be set by the user program is he is allocating his own buffer and not using the INIT call. This address is used in conjunction with the flag bit 6 above which indicates that a buffer has been allocated either by the INIT call or by the user. No transfers can take place without a buffer.

Record size - this is the size in bytes for the physical transfer to use. The READ call will transfer this number of bytes from the device to the user buffer beginning with the address in DDB+2. The WRITE call will transfer this number of bytes from the user buffer to the user device. This size is set to the standard buffer size by the INIT call or by the user if he is doing his own buffering. It may be modified by the user for transferring records of variable sizes as long as it does not exceed the buffer size of the capacity of the device or driver in use. Various logical file service routines set this size word during processing such as the OPEN call for the disk which must perform directory operations on a 512-byte buffer at all times.

Buffer index - this is a byte counter which is used by logical routines (INPUT and OUTPUT calls) for keeping track of bytes transferred into and out of the user buffer. Various calls will reset this value and the user will then make use of it and increment it as bytes are transferred into and out of the buffer.

Details will be given in later sections where the calls themselves are described. This buffer index word is normally not a true buffer pointer but rather an offset from the buffer base (per DDB+2) to the current byte being manipulated.

Record number - set by the user to read or write a specific random record from a random access device such as disk. The first record on the device is considered record zero and the record numbers increment sequentially from there. This record number is actually used only by the physical driver routines for READ and WRITE calls but other logical calls set this word to perform transfers to specific disk areas such as directory operations on disk. Most devices are not random access and therefore this record number will be ignored by the respective drivers.

Queue chain link - this word is for internal use only and is the link used by the I/O queueing routines for interrupt driven transfers. The user should not alter this word.

JCB address - file service routines store the address of the controlling job's JCB so that interrupt driven drivers can locate the corresponding job for activation on transfer complete status. This word is also for internal use only.

Job priority - the current software job priority is set here by file service routines to specify the priority of the transfer in queued operations. This byte is for internal use only. The top byte of this word (DDB+17) is currently not used.

Device code - the three character device code (packed RAD50) must be set here by an FSPEC call or directly by the user before any IO operations may be performed.

Drive - used only be drivers for devices with multiple drives, this byte must be set to specify the drive to be used for the transfer. A -1 byte (octal 377) may be used to indicate the current default drive number which will depend on the particular driver in use as to action. If the device is DSK the default drive used will be the drive that the current user is logged in under.

Call level - for internal use only, this byte is used to keep track of the level of nesting of the file service calls for proper error recovery handling. This byte must be zero before the first file call is executed.

Filename and extension - three words which contain the RAD50 packed filename and extension for file structured devices. These words are ignored by drivers for devices which are not file structured but they may cause funny error messages if they are not set to zero values.

PPN - this is the octal project-programmer bytes for the area to be used to locate the file. Used only on file structured devices which are multi-user based such as disk. A zero causes the default value to be the current PPN which the job is logged in under. This word should be zero if not used to prevent funny error messages.

Open code - this byte is set by the OPEN call to indicate the mode of the open statement for future processing operations. It is normally ignored by drivers

for devices which are not file structured. It is for internal use only and should not be modified by the user. The corresponding top byte of the word (DDB+35) is currently not used. The following open codes are in use:

0 - file is not open
1 - file is open for sequential input (OPENI call)
2 - file is open for sequential output (OPENO call)
4 - file is open for random input/output (OPENR call)

Driver work area - the remaining five words are for internal use by the device drivers for links, record counts, etc and should not be modified by the user during processing. Not all drivers make use of the work area but it must be there if device independence is to be preserved.

DEVICE TRANSFER BUFFERS

Each dataset must have an associated transfer buffer for input and output operations to take place through. This buffer must be allocated either directly or through use of the INIT call which allocates the buffer as a memory module by using a GETMEM call. The INIT call will allocate a standard size buffer for the device being used (the size of the buffer is defined within the driver itself). If you do not wish to use the INIT call you may allocate any size buffer you wish (must be large enough for any logical calls to be performed) and then set its address in DDB+2. Refer to the section detailing the I/O calls themselves for more details on the use of these buffers.

ERROR HANDLING

When an error occurs during any file service call the file service routines will normally perform typical error correction procedures. If the error is fatal (uncorrectable) two operations may or may not take place depending on the setting of bits 0 and 1 in the flags byte at DDB+1. First, bit 1 is tested and if it is not set, the monitor outputs a standard error message to the user terminal giving the type of call that failed, the file specification for the device that the error occurred on, and the reason for the error. The appropriate error code is also placed in the error byte at DDB+0 for later testing by the user. Second, bit 0 of the flags byte is tested and if it is not set, the user program is aborted by the file service system and a return to monitor mode is made. The user will normally set these bits on before any I/O calls are made if it is desired to process the errors within the user program itself.

The following list gives the error code (in octal) returned in the DDB error byte by the file service system along with the reason for the error:

01 - file specification error (FSPEC)
02 - insufficient free memory for buffer allocation (INIT)
03 - file not found (OPENI, OPENR, DELETE, RENAME)
04 - file already exists (OPENO)
05 - device not ready (all calls)
06 - device full (OUTPUT)
07 - device error (all calls)

10 - device in use (ASSIGN)
11 - illegal user code (all file calls)
12 - protection violation (OPENO, OPENR, DELETE, RENAME)
13 - write protected (all output calls)
14 - file type mismatch (OPENI, OPENO, OPENR)
15 - device does not exist (all calls)
16 - illegal block number (READ, WRITE)
17 - buffer not inited (all calls except INIT)
20 - file not open (READ, WRITE, INPUT, OUTPUT, CLOSE)
21 - file already open (all OPEN calls)
22 - bitmap kaput (all disk bitmap calls)

FILE SERVICE MONITOR CALLS

This section will describe the file service calls which are available to the user program for both logical and physical I/O operations. All calls have the same general format which uses a single argument representing the dataset driver block (DDB) to be used for the operation. See the preceding chapter for a complete description of the DDB format. In brief, the calls to be described in this section are:

FSPEC	process a device specification
INIT	initialize a dataset driver block buffer
LOOKUP	lookup a file to see if it exists
OPENI	open a file for sequential input
OPENO	open a file for sequential output
OPENR	open a file for random input/output
CLOSE	close a file to further processing
READ	read a physical record
WRITE	write a physical record
INPUT	read a logical record
OUTPUT	write a logical record
WAIT	wait for an I/O operation to finish
DELETE	delete a file
RENAME	rename a file
ASSIGN	assign a device to a job
DEASGN	deassign a device from a job

FSPEC

The FSPEC call is used to process an ASCII file specification from a command line (or any other ASCII buffer) and set up the parameters in the DDB according to the results of the processing. The ASCII file specification must be indexed by R2 and must be in the standard format of dev:filnam.ext[ppn] with a valid termination character if a short default specification is used.

The FSPEC call is slightly different from the rest of the I/O calls in that it allows a second argument to be used if desired. This argument must be the default extension for the filename parameter to be used in the event that the file specification does not contain an explicit extension (identified by a period after the filename). If the second argument does not exist the FSPEC processor will not process the input file specification past the colon which terminates the device/drive parameters.

The device code (3-characters) will be packed RAD50 and stored in DDB+20 if it exists as marked by the terminating colon. The drive number will be stored in the byte at DDB+22 if it exists. If the device code does not exist, the current default device (stored in the job's JCB item JOBDEV) will be stored in DDB+20. If the drive number is not in the input spec an octal 377 will be stored in DDB+22 to flag the default drive number to the device driver.

The filename and extension are then processed unless no second argument was used in the call in which case the FSPEC processor returns to the user at this point. The filename and extension are packed RAD50 and stored in the three words at

DDB+24 through DDB+30. If no filename is entered in the input specification the word at DDB+24 will be cleared to zero to flag the absence of the filename parameter. If a filename is entered but no extension is entered then the default extension specified in the second argument of the FSPEC call is stored as the extension in DDB+30.

If a project-programmer number is in the file specification (marked by a left square bracket "[") it will be processed and stored in DDB+32. If no ppn is entered, DDB+32 will be cleared to zero to flag its absence.

At the conclusion of the processing of the input file specification, the index R2 will be pointing to the termination character (the first character following the file specification string). If an error in the input string is detected the FILE SPECIFICATION ERROR message will be printed (unless suppressed by bit 1 in DDB+1) and the program will be aborted (unless suppressed by bit 0 in DDB+1). The error code 01 will be set in DDB+0 error code byte.

No other modifications take place to the DDB area except that the error byte at DDB+0 is cleared at the start of the FSPEC processing. If the user does not use the FSPEC call to set up his DDB then he must use some other form of explicit code to insure that the DDB is set up properly to define the device and file for any subsequent I/O operations.

INIT

The INIT call is the normal means for allocating the dataset buffer and initializing the DDB for processing. The INIT call will locate the device driver (searching area 1,6 on DSK0 if not in memory) and then allocate a standard size buffer based on the size specified in the driver. Bit 6 of the flag byte at DDB+1 will be set to indicate the initialization. The address of the buffer will be set into DDB+2 and the size in bytes will be set into DDB+4.

There are no calls which deallocate the buffer once it has been allocated by the INIT call. Multiple OPEN-CLOSE processes may be performed on the DDB once the INIT has been done. The buffer is temporary and will be deallocated automatically when the program exits to monitor or it can be explicitly deallocated by using the DELMEM call with the address stored in DDB+2. Recall that the buffer is allocated as a standard memory module with a GETMEM call.

LOOKUP

This is a form of the OPEN call which does nothing except search for the file and return an error code if it is not found. The file is not actually opened for processing and an OPENI call must be used if the file is to be subsequently read from. The LOOKUP call is useful for determining if a file that is about to be opened for output already exists so that it can first be deleted by the DELETE call. The LOOKUP call is ignored for devices which are not file structured.

The lookup call is also useful for some system programming techniques since it returns parameters about the file in the DDB work area. The work area is located in the last five words of the DDB. The first three words of this work

area are loaded with the three words of the directory item if the file is found. These three words are the number of records in the file, the number of active data bytes in the last record, and the record number of the first data record in the file. Refer to the section titled FLOPPY DISK STRUCTURE FORMAT for complete details on the directory format.

OPENI

The OPENI call locates a file in a file structured device and sets up the DDB parameters (work area) for subsequent INPUT processing. An error results if the file is not found. The code 01 is set into DDB+34 to flag the OPENI operation. The OPENI call is normally followed by a series of INPUT calls which deliver sequential records from the file to the user buffer. The OPENI call is ignored for devices which are not file structured.

OPENO

The OPENO call first searches the specified device in the specified user area and returns an error if the file already exists. If it does not, the DDB is set up for OUTPUT processing. The code 02 is set into DDB+34 to flag the OPENO operation. The OPENO call is normally followed by a series of OUTPUT calls which transfer data from the user buffer to sequential records in the file. The OPENO call is ignored for devices which are not file structured.

OPENR

The OPENR executes basically the same as the OPENI call but the code stored in DDB+34 is 04 to flag random processing. The file located for random processing must be a contiguous file. The OPENR call is normally followed by a series of INPUT and OUTPUT calls which transfer data between specific records in the file and the user buffer in both directions. The OPENR call is also ignored for devices which are not file structured.

CLOSE

The CLOSE call finishes up logical processing of a file and clears the open code in DDB+34. No further INPUT or OUTPUT operation may occur once a file has been closed. No action is normally done on a file which is open for input. For files open for output, the final record is written out and the file is added to the directory system on the specific device. The CLOSE call is ignored for devices which are not file structured.

READ

This is the physical transfer call for reading input data from a device. No check is made for file open status since the READ call is not a logical file call.

For sequential access devices such as a paper tape reader, the READ call will

deliver one record from the device to the user buffer. The size of this record will normally be the number of bytes specified in DDB+4 but this may not necessarily be true if the driver does not transfer under the rules of the system. If the device is not capable of generating the requested number of bytes per DDB+4 (such as a tape reader which runs out of tape) a lesser number may be transferred in which case the count in DDB+4 will be adjusted to reflect the true number actually transferred to the user buffer.

For random access devices such as disk, the user must specify the record number to be located and read by placing that number into DDB+10 before executing the READ call. Most random access devices will always transfer the requested number of bytes per DDB+4 into the user buffer. An error will result if the record number is not within the range of the specific device. The standard AMOS floppy disk is structured as 500 (decimal) records of 512 bytes each. The legal record numbers therefore range from 0 through 499 decimal.

The system allows interrupt driven devices to be queued and processed in a priority fashion. Normally, the execution of a READ call will suspend the running of the user program until the transfer has been completed at which time the user job will be reactivated. The user may optionally set the realtime bit (bit 2) in the flag byte at DDB+1 to force an immediate return to the program once the transfer has been queued or initiated. The user must then either test the dataset busy bit (bit 7) of the flag byte or use the WAIT call to stall until the transfer has been completed. The dataset busy flag will be reset when the transfer has been completed. The user must also then check for errors. The realtime bit is ignored for devices which are not interrupt driven or whose drivers do not run under the I/O queue system.

WRITE

This is the physical transfer call for writing data to a device. No check is made for file open status since the WRITE call is not a logical file call.

For sequential access devices such as a printer, the WRITE call will deliver one record to the device from the user buffer. The size of this record will be the number of bytes specified in DDB+4. The driver is responsible for the correct transfer count and the user may alter the number in DDB+4 for each new WRITE call to the same device for the writing of variable length records.

For random access devices such as disk, the user must specify the record number to be located and read by placing that number into DDB+10 before executing the WRITE call. Most random access devices will always transfer the requested number of bytes per DDB+4 into the user buffer. An error will result if the record number is not within the range of the specific device. The standard AMOS floppy disk is structured as 500 (decimal) records of 512 bytes each. The legal record numbers therefore range from 0 through 499 decimal.

The system allows interrupt driven devices to be queued and processed in a priority fashion. Normally, the execution of a WRITE call will suspend the running of the user program until the transfer has been completed at which time the user job will be reactivated. The user may optionally set the realtime bit (bit 2) in the flag byte at DDB+1 to force an immediate return to the program once the transfer has been queued or initiated. The user must then either test

the dataset busy bit (bit 7) of the flag byte or use the WAIT call to stall until the transfer has been completed. The dataset busy flag will be reset when the transfer has been completed. The user must also then check for errors. The realtime bit is ignored for devices which are not interrupt driven or whose drivers do not run under the I/O queue system.

INPUT

The INPUT call is the logical equivalent of the READ call for logical processing of datasets. The INPUT call reads a logical record within a file or device dataset under the control of the specific driver in use. A dataset must be opened for input (OPENI) or random access (OPENR) before INPUT calls are performed. The INPUT call first sets the standard buffer size into DDB+4 so the user may not use this call to transfer non-standard record sizes. The number of bytes actually read may be less than the standard record size due to the driver processing or due to an end of file condition. The actual number of bytes transferred will be set into DDB+4 by the driver routine.

The main use of the INPUT call is in logical sequential file processing and the INPUT call therefore sets up the buffer index value in DDB+6 to direct the processing of the data by the user routines. This index value is actually the offset to the first byte of valid data within the user buffer whose base address is at DDB+2. For unit record devices this value will be zero since all data within the buffer is user data. For sequential disk files, however, the first word in each record within the file is a link word to the next record and therefore the value set into DDB+6 by the disk driver will be 2 to start the processing with the third byte in the user buffer.

The following subroutine is normally used to get each byte of data from a sequential file:

```
;SUBROUTINE TO GET NEXT BYTE FROM FILE DEFINED AS INDDB AND LEAVE IT IN R1
INBYTE: CMP     INDDB+6,INDDB+4 ;IS THE BUFFER EMPTY?
        BLO     INBG            ;  NO - GET NEXT BYTE
        INPUT   INDDB           ;READ NEXT LOGICAL RECORD INTO BUFFER
        TST     INDDB+4         ;CHECK FOR END OF FILE (NO DATA TRANSFERRED)
        BEQ     INEOF           ;  GO TO END OF FILE ROUTINE
INBG:   PUSH    INDDB+2         ;STACK THE BUFFER BASE ADDRESS
        ADD     INDDB+6,@SP     ;  AND ADD THE INDEX OFFSET TO GET POSITION
        MOVB    @(SP)+,R1       ;PICK UP THE NEXT BYTE FROM USER BUFFER
        AND     #377,R1         ;INSURE UPPER BYTE IS CLEARED IN R1
        INC     INDDB+6         ;INCREMENT THE BUFFER INDEX FOR NEXT TIME
        RTN                     ;SUBROUTINE RETURN
```

For devices that do not implement special processing of logical calls the INPUT call performs a READ call instead.

A special situation is involved for files opened for random access by the OPENR call. Instead of reading the next sequential record, the specific relative record whose number is in DDB+10 will be read into the user buffer. The user first sets this number up and then executes the INPUT call. The record number is actually relative to the base of the file and has no direct relationship to the physical record on the device as would be returned by a READ call.

OUTPUT

The OUTPUT call is the logical equivalent of the WRITE call for logical processing of datasets. The OUTPUT call writes a logical record to a file or device dataset under the control of the specific driver in use. A dataset must be opened for output (OPENO) or random access (OPENR) before OUTPUT calls are performed. The OUTPUT call will transfer the number of bytes in DDB+4 but it will normally do it as a standard record (depends on the driver in use). The user is discouraged from attempting to use the OUTPUT call for transferring non-standard record sizes. The user is awarded two black stars for such filandering. (Black stars are bad, you realize.)

The main use of the OUTPUT call is in logical sequential file processing and the OUTPUT call therefore sets up the buffer index value in DDB+6 to direct the processing of the data by the user routines. This index value is actually the offset to the first byte position for valid data within the user buffer whose base address is at DDB+2. For unit record devices this value will be zero since all data within the buffer is user data. For sequential disk files, however, the first word in each record within the file is a link word to the next record and therefore the value set into DDB+6 by the disk driver will be 2 to start the processing with the third byte in the user buffer.

The following subroutine is normally used to put each byte of data to a sequential file:

```
;SUBROUTINE TO PUT NEXT BYTE FROM R1 INTO FILE DEFINED AS OTDDB
OUTBYT: PUSH    OTDDB+2          ;STACK THE BUFFER BASE ADDRESS
        ADD     OTDDB+6,@SP      ;  AND ADD INDEX OFFSET TO GET POSITION
        MOVB    R1,@(SP)+        ;MOVE DATA BYTE TO USER BUFFER
        INC     OTDDB+6          ;INCREMENT THE BUFFER INDEX OFFSET VALUE
        CMP     OTDDB+6,OTDDB+4  ;IS THE USER BUFFER FULL NOW?
        BLO     OTBX             ;  NOT YET SO RETURN
        OUTPUT  OTDDB            ;OUTPUT THE CURRENT LOGICAL RECORD TO FILE
OTBX:   RTN                      ;SUBROUTINE RETURN
```

For devices that do not implement special processing of logical calls the OUTPUT call performs a WRITE call instead.

A special situation is involved for files opened for random access by the OPENR call. Instead of writing the next sequential record, the specific relative record whose number is in DDB+10 will be written out from the user buffer. The user first sets this number up and then executes the OUPUT call. The record number is actually relative to the base of the file and has no direct relationship to the physical record on the device as would be written by a WRITE call.

WAIT

The WAIT call is used only for interrupt device processing to stall until the record transfer has been completed. The WAIT call has no effect unless the user has set the realtime flag bit (bit 2) in DDB+2 and the device is interrupt driven. The job will be suspended until the specified dataset is not busy as

marked by the flag bit 7 becoming zero. If the dataset is not busy when the call is executed, an immediate return is made. The WAIT call also checks for an error condition that may have occurred during the transfer process and performs error handling under control of flag bits 0 and 1.

DELETE

The DELETE call deletes a specific file from a file structured device. The filename, extension and ppn (if used) must be set in the DDB before executing the call. An error results if the file is not found. The DELETE call is ignored for devices which are not file structured.

RENAME

The RENAME call renames a specific file on a file structured device. The filename, extension and ppn (if used) must be set in the DDB before executing the call. The new filename and extension must be packed RAD50 into the three words immediately following the DDB in memory. The RENAME call merely locates the directory item for the file and replaces the three words which store the filename and extension. The RENAME call is ignored for devices which are not file structured.

ASSIGN

The ASSIGN call is used to assign a non-sharable device (such as a printer) to the current user's job by setting a flag in the device's entry in the device table in monitor memory. Once a device has been assigned by this call any attempt to assign it by another job will result in an error. The device will stay assigned to this job until deassigned by the DEASGN call. The ASSIGN call performs no action if the specified device is sharable such as a disk.

DEASGN

The DEASGN call is used to deassign a device which has been assigned to the user's job by the ASSIGN call. Once deassigned, the device becomes available for assignment by other jobs. The DEASGN call performs no action if the specified device is sharable of if it is not currently assigned to the user's job. All devices are deassigned when the program exits to the monitor.

DISK SERVICE MONITOR CALLS

The disk presents special problems which require the use of special monitor calls to control the accessing of the directory and bitmap records. These records have a non-sharable attribute associated with them even though the disk in general is a sharable device. For instance, two user programs may not both be updating the same directory records at the same time. The same holds true for the bitmap records. The following monitor calls are used to control the access to these non-sharable records:

DSKCTG - allocates a contiguous file for random processing
DSKALC - allocates the next available record on disk
DSKDEA - deallocates a specific record on disk
DSKBMR - reads disk bitmap and sets reentrant lock flag
DSKBMW - rewrites disk bitmap after user modification
DSKDRL - sets reentrant directory lock for a specific user
DSKDRU - clears reentrant directory lock for a specific user

The access to these records is normally done by the monitor routines as a direct result of normal I/O processing by file service calls. It is a somewhat tricky process and the disk calls should not be used except with extreme caution since misuse could violate the integrity of the file structure on the disk. The following descriptions are directed at those system programmers who are familiar with shared file techniques.

All calls use a standard argument which is the address of the associated DDB to be used for the call. In addition to the first argument which is the DDB, some calls use a second argument for processing. The second argument, if used, will be detailed in the description of the call.

BITMAP AREA FORMAT

The bitmap area is an area in monitor memory which is allocated by the BITMAP program run at system startup time by the BITMAP command in the system initialization command file. This area consists of a status word, a DDB for bitmap reads and writes, and a buffer for the actual bitmap including the hash total words. The format of the bitmap area is as follows:

```
WORD    0               ;Bitmap status word
BLKW    12              ;Partial DDB for bitmap I/O
BLKW    Bitmap-size     ;Bitmap buffer (size depends on device)
BLKW    2               ;Hash total words
```

The device table entry for each drive has the address of the corresponding bitmap area to be used for that drive. More than one drive may share the same bitmap area which will force a rewrite each time a different drive is referenced. This is not efficient timewise but can save some memory for larger devices where the bitmap buffer may be several hundred words or more.

The status word (first word in bitmap area) contains two flags which are used to control bitmap access. Bit 0 is the bitmap lock flag and is set to flag that the bitmap is locked and being read or modified by some user job. The DSKBMR

call sets this flag on and it is up to the user to clear it after he has finished the bitmap access and modification. Bit 1 is the bitmap rewrite flag which is set to indicate that one or more modifications have been made to the bitmap in memory and that it must be rewritten to disk before being discarded. If the user program modifies the bitmap in memory it must set the rewrite flag to insure that the bitmap is rewritten.

The bitmap DDB is a partial DDB due to the fact that no files are ever referenced and the rest of the DDB is not needed. The bitmap is normally allocated as record 2 of each disk and it extends across successive records for those devices which overflow one record.

The bitmap buffer area is the exact size required to contain the entire bitmap from the disk. Two extra words are allocated to contain the hash total which is used to insure the integrity of the bitmap in memory and on disk. Each time the bitmap is read or before the bitmap is rewritten this hash total is checked and an error results if it is bad. The hash total is merely the double-word binary sum of the entire bitmap buffer. The user must update this hash total each time he modifies the bitmap or else an error will result when it becomes time to rewrite the bitmap to disk.

The bitmap itself contains one bit for each logical record on the disk structure and this bit is off if the record is free and on if the record is in use by anyone including the system structure records themselves. Each word in the bitmap can define up to 16 records. The first word in the bitmap defines records 0 through 17 (octal) with bit 0 defining record 0 and proceding upward throughout the word. The second word defines records 20 thorugh 27 and so on. To define the 500 decimal records in a standard AMOS floppy disk we need 32 words (32 times 16 = 512) with the last word not being totally used. The bitmap itself therefore takes up 34 words which includes the two hash total words.

Altering the bitmap is tricky but the sequence recommended is:

1. Read the bitmap using the DSKBMR call
2. Alter the bitmap as necessary (recompute the hash total)
3. Set the rewrite flag (status word bit 1)
4. Clear the bitmap lock (status word bit 0)
5. Rewrite the bitmap using the DSKBMR call
6. Breathe a sign of relief if it worked

DSKCTG

The DSKCTG call is used to allocate a contigous file on a random access device. A standard argument is used as the second argument which represents the number of records to be allocated in the file. A search will be made to find the first available hole on the disk which will fully contain the requested number of records. These records are marked as in-use on the disk bitmap and a file descriptor item is added to the user directory. The word which gives the number of bytes in the last record will be set negative to flag this file as contiguous to distinguish it from the normal sequential files. A device full error will result if no hole can be found on the disk which is large enough to contain the file.

DSKALC

The DSKALC call is used to allocate one record for use by this user as a directory record or as a file record. A standard argument is used as the second argument which represents the word which is to receive the record number of the allocated record. An error will result if there are no free records left on the specified disk. A DSKBMR call is first performed to insure that the current job has access to the bitmap and then the first free record is located and marked in use. The bitmap record is flagged as modified which will cause it to be rewritten at the next DSKBMW call or if it must be swapped out to make room for another bitmap sharing the same area in memory.

DSKDEA

The DSKDEA call is used to deallocate a specific record on a disk and make it immediately available for use by another user (or the same user). A standard argument is used as the second argument which represents the address of the word which contains the record number of the record to be deallocated. No check is made to insure that this record is allocated to either the current user or any other user. A DSKBMR call is first performed to insure that the current job has access to the bitmap and then the specified record's bit is set to zero to indicate that the record is free. The bitmap record is flagged as modified to force a rewrite.

DSKBMR

The DSKBMR call locates the bitmap area in monitor memory for the specified disk and insures that it is not locked by another job. If it is locked, a stall will be made until it is released. It is then locked for this job and a return is made to the user. The address of the bitmap area will be set into the word specified by the second argument in the calling sequence. The second argument is a standard argument in format. Refer to the description of the bitmap area above and note that the second argument receives the address of this area and not the address of the bitmap itself. The user may locate the bitmap itself because its address is in the third word of the bitmap area (second word of the bitmap DDB).

DSKBMW

The DSKBMW call locates the bitmap area in monitor memory for the specified disk and insures that it is not locked by another job. If it is locked, a stall will be made until it is released. It is then locked for this job and rewritten to disk from memory unless the hash total is bad. After the rewrite is complete both the rewrite and lock flags are cleared and a return is made to the user.

DSKDRL

The DSKDRL call locks the directory for the specified drive for modification by the user program. It is used by such file service routines as CLOSE for output files, DELETE and RENAME calls. If the directory is already locked by another

job a stall will be made until it is released. The user program or routine must unlock the directory via the DSKDRU call after the modifications have been made.

DSKDRU

The DSKDRU call unlocks the directory for the specified drive after it has been locked by the DSKDRL call for modification. No action will be performed if the directory is not locked by the current job.

TERMINAL SERVICE MONITOR CALLS

The AMOS monitor has several calls which deliver data to and from both the user terminal and other terminals connected to the system. A terminal is defined as an ASCII character-oriented device which is capable of both output and input. This is the formal definition and does not preclude the use of output-only devices on terminal designated ports. Also, the system includes software terminals known as "pseudo terminals" which can be used to control jobs that are not actually associated with a hardware interface on a designated port address. The calls listed here normally input from or output to the terminal which is controlling the job that the call is being executed by. Some calls (as specified) will input from or output to another terminal not connected to the current job or to a pseudo terminal controlling another job.

Programs which make use of the standard terminal service calls that communicate with the user terminal can be run in a job controlled by a pseudo terminal without modification. Keyboard input calls and terminal output calls will always go to the controlling terminal regardless of which job they are running in. The user, therefore, need not be concerned with the physical port address or attributes of the terminal which is controlling the job. The monitor routines handle all this automatically.

Due to a holdover from older system terminology most terminal output calls reference the device name of "TTY" which used to define the teletype device on systems which normally used teletypes as terminals. The input device of the teletype was then called the keyboard and the calls reference the device name of "KBD". These are strictly mnemonics and hold no true bearing to the attributes of the physical terminals which now are more commonly the higher speed video CRT terminals.

Each terminal has associated with it a terminal line table which is a work area in monitor memory set up to contain the parameters and work areas associated with the control of the terminal device. Most of the items in this terminal line table are for internal use only and the user need not be concerned with them. The TIDX call is provided to set an index (R0) to the associated terminal line table so that the user may inspect or modify the items within. More details on this will be given in another manual dealing with monitor modifications and functions. The only item that the user normally need be concerned with is the terminal status word which is the first word in the terminal line table. This word has certain flags in it which the user may modify to alter the operation of his terminal calls. The terminal status word has the following flag positions defined:

Bit 0 - user sets to force image mode input (see KBD call)
Bit 1 - user sets to suppress echoing of input characters
Bit 2 - internal flag to indicate backslash sent during rubout
Bit 3 - internal flag to indicate input character was lost
Bit 4 - user sets to allow lower case input (disables conversion)
Bit 5 - user sets to force image mode output (see TTY call)
Bit 6 - user sets to suppress "^C" echo for control-c inputs

The terminal status word is cleared each time the user program exits back to monitor mode upon program completion thereby restoring normal terminal operation

regardless of program operation.

KBD

The KBD call accepts one full line of input from the user terminal into a monitor line buffer and then sets index R2 to the base of that buffer for the user reference. During the inputting of the line, the user job is set into the terminal input wait state thereby consuming no CPU time until the line is finished. All normal line editing features are active (rubout, control-U, tab, etc) and a control-c input will abort the job unless the user has set up control-c trapping via the JOBICP item in the JCB for the job. The line will be terminated when a carriage-return or a line-feed is entered. The carriage-return will have a line-feed automatically appended to it by the monitor and a null byte will be set after the line-feed character.

If the echo-suppress flag is set in the terminal status word, normal echoing of the input characters will be suppressed such as when the password is being entered for the LOG command. If the image-mode input flag is set the KBD command takes on a whole new lease on life. No editing is performed and instead of one line being accepted, only one character is accepted and it is delivered back to the user in register R1 instead of setting register R2 to the monitor line buffer. Image mode input echoing is still under control of the echo-suppress flag as in normal line mode.

TTY

The TTY call outputs one character from register R1 to the controlling terminal and then returns. Tabs are echoed as spaces up to the next modulo-8 carriage position unless the image-mode output flag is set in the terminal status word. If the job is running under the command of a control file, the character will only be output to the terminal if the output suppress command is in the normal state (:R revives it, :S silences it).

TIDX

Sets register R0 to the address of the base of the terminal line table for the controlling terminal of this job. The user may then modify bits in the terminal status word which is located at @R0.

TIN

Gets the next input character from either the terminal input buffer or from the command string if the job is controlled by a command file. The character is delivered in R1. This call is normally only used within the operating system itself and not by user programs.

TOUT

Outputs one character to the controlling terminal of the job or to the job which has this job attached (by the address in the JOBATT item). This call differs from the general TTY call in that the command file status is not checked by the TOUT call. The TOUT call, like the TIN call, is normally only used within the operating system itself.

TAB

This is a convenience call which outputs a single tab character to the user terminal. It is in effect the same as the code sequence:

```
MOV     #11,R1
TTY
```

CRLF

This is a convenience call which outputs a carriage-return and line-feed pair to the user terminal. It is in effect the same as the code sequence:

```
MOV     #15,R1
TTY
MOV     #12,R1
TTY
```

TTYI

The TTYI call outputs a string of characters which follows the call itself up to but not including a null byte. The call could be used as follows to output two lines of data to the user terminal:

```
TTYI
ASCII   /LINE 1 DATA/
BYTE    15
ASCII   /LINE 2 DATA/
BYTE    15,0
EVEN
```

The TTYI call will also automatically append a line-feed to all carriage-returns which are included in the string.

TTYL

The TTYL call is similar to the TTYI call in that it outputs a string of ASCII characters up to a null byte. The string of characters for the TTYL call may be anywhere in memory and not inline with the call itself in the program flow. The TTYL call takes one standard argument which is the address of the message to be output. The TTYL call is therefore useful for outputting from a table of messages by setting an index to the specific message within the table (per some

numeric director code) and then using that register as the argument to the TTYL call. The TTYL call also appends a line-feed to each carriage-return in the string.

PTYIN

The PTYIN call allows one job to force a character into the input buffer of another job which is probably controlled by a pseudo terminal. The PTYIN call takes two standard arguments. The first argument is the data byte to be sent to the other job and the second argument is the address of the JCB of the job into which the character is to be forced. The PTYIN call is the method by which the FORCE operator command does its dirty work.

PTYOUT

The PTYOUT call allows one job to get a character from the terminal output buffer of another job which is controlled by a pseudo terminal. If no output is available from the specified job, the calling job is put to sleep until a character is available. The PTYOUT call takes two standard arguments. The first argument is the address of the byte which will receive the data character and the second argument is the address of the JCB from which the character is to be stolen.

TTYIN

The TTYIN call allows one job to get waiting input data from the terminal input buffer of another job. This call has not yet been fully inplemented (nor have we found a good use for it either).

TTYOUT

The TTYOUT call allows one job to put data into another jobs terminal output buffer. This call, like the TTYIN call, is not yet fully implemented.

MESSAGE CALLS

There are three calls which have been defined in SYS.MAC as macros using the TTYI call. These calls are for the convenience of the programmer and to make the program more readily understandable. They all take a single argument which is an ASCII message string to be output to the user terminal. Due to the way that macro arguments are processed, if the message has leading or trailing spaces or if it has imbedded commas, it must be enclosed in angle brackets or part of it will be lost. The three calls are:

```
TYPE    msg     ;Types the message on the user terminal as is
TYPESP  msg     ;Types the message and appends one space to it
TYPECR  msg     ;Types the message and appends a CRLF pair to it
```

The macros are defined in SYS.MAC as follows:

```
DEFINE  TYPE    MSG
        TTYI
        ASCII   /MSG/
        BYTE    0
        EVEN
        ENDM

DEFINE  TYPESP  MSG
        TTYI
        ASCII   /MSG' /
        BYTE    0
        EVEN
        ENDM

DEFINE  TYPECR  MSG
        TTYI
        ASCII   /MSG/
        BYTE    15,0
        EVEN
        ENDM
```

It should be noted that the message may not contain any slashes since these are used as delimiters for the ASCII statement in the macros.

NUMERIC CONVERSION MONITOR CALLS

The AMOS monitor contains two calls which perform conversions from a single binary word value to an ASCII formatted decimal or octal string. Options for the conversion allow the string to be sent to the user terminal, to an output file or to a buffer in memory. Options also allow the control of the format which the result is in. Both calls have the same general format and take two arguments, each of which must be an expression that evaluates down to a byte value within the specified range. The two calls are:

```
DCVT    size,flags      ;Convert binary number in R1 to decimal
OCVT    size,flags      ;Convert binary number in R1 to octal
```

The size byte determines the number of digits in the output result. A zero size specifies a floating format in which the number of digits used will be just enough to fully contain the result. A non-zero size specifies a fixed number of digits for the result with leading zeros being replaced by blanks. In either form, if the R1 value is zero at least one zero digit will be output as the result.

The flags byte contains six flags which control the destination of the result string and also some other formatting options. The following list gives the flag bit positions and the action taken when the flag is set:

Bit 0 - disables leading zero blanking
Bit 1 - outputs the result to the user terminal
Bit 2 - outputs the result to the file whose DDB is indexed by R2
Bit 3 - puts result in memory at buffer indexed by R2 and updates R2
Bit 4 - adds one leading space to the result
Bit 5 - adds one trailing space to the result

Note that the maximum value which can be displayed using these calls is the maximum value of a 16-bit word. All numbers are considered unsigned so the largest decimal number is 65,535 and the largest octal number is 177777.

The following examples may clarify things a bit. All examples assume the value in R1 is 964 (decimal) and the letter "b" in the result field indicates a blank.

```
DCVT    0,2     prints 964
DCVT    0,22    prints b964
DCVT    0,42    prints 964b
DCVT    5,2     prints bb964
DCVT    5,3     prints 00964
DCVT    5,43    prints 00964b
DCVT    5,63    prints b00964b
DCVT    2,2     prints 64 (the 9 is lost)
```

RAD50 CONVERSION MONITOR CALLS

Radix-50 packing is used throughout the system where the packing of filenames and other data entities lends itself. Radix-50 (RAD50) packing is a system by which 3 ASCII characters may be packed into a single 16-bit word using a special algorithm based on the value of octal 50. The character set that may be packed RAD50 is limited in scope to the alphanumeric characters, the period, the dollar sign, and the blank. The following list gives the legal characters that may be packed RAD50 and their equivalent octal codes:

Character	RAD50 code
blank	0
A-Z	1-32
$	33
.	34
0-9	36-47

There is no character for the RAD50 code 35.

The packing algorithm for a three-character input to a 16-bit RAD50 result is:

1. The first character code is multiplied by 3100 octal (50x50)
2. The second character code is multiplied by 50 and added to the first
3. The third character code is added to the above to form the result

The unpacking algorithm merely reverses the above sequence to get the triplet.

There are two monitor calls which perform the above packing and unpacking algorithms. Both calls use registers R1 and R2 as indexes to the components and require no calling arguments.

PACK

The triplet (3 ASCII characters) indexed by R2 is packed into RAD50 form and the result is left in the word indexed by R1. R1 is incremented by 2 to receive the next result word for multiple packing. R2 is left indexing the first character which was not included in the packing of this triplet. The PACK call will terminate packing and force blank fill for any input which does not contain 3 valid RAD50 characters. For the PACK call, a blank will be considered an illegal input character and will terminate packing.

UNPACK

The word in the address indexed by R1 will be unpacked and the triplet will be left in the three bytes beginning with the byte currently indexed by R2. R1 will be incremented by 2 for the next word and R2 will be incremented by 3 for the next triplet result. Blanks are legal in unpacking and will be placed into the result if they are decoded from the input word.

INPUT LINE PROCESSING CALLS

When a program is executed by an operator command, register R2 is left pointing to the first non-blank character on the command line which follows the command name itself. The remainder of the line is normally interpreted by the particular program and used to determine the files to be acted on, the record number to be dumped, the devices to be accessed, etc. For example, the MACRO call requires the name of the program and any switch options to follow the MACRO command name on the same line. The macro assembly program then processes the program name and the switch options by way of the R2 index which was left indexing the rest of the command line. This command line is actually the JOBKBF buffer in the user's JCB.

In addition to the command input line, the KBD monitor also leaves R2 set to the input line buffer which contains the user input data. Also, various translators and file processing programs may read in a line of data and then set index R2 to the base of that line for scanning. For this reason, there exist a number of monitor calls which perform scanning and conversion functions based on an input line which is indexed by R2. Some of the calls merely test the character indexed by R2 for a specific condition and return with flags set based on the result of the test. In these instances R2 is not modified. In calls which perform scan conversions, R2 is updated to point to the character which terminated the conversion. With the exception of the FILNAM call, none of these calls require any arguments. Conversion results are always delivered back to the user in register R1.

ALF

The character indexed by R2 is tested for alphabetic (A-Z) and the Z-flag is set if it is or cleared if it is not. R2 is not changed.

NUM

The character indexed by R2 is tested for numeric (0-9) and the Z-flag is set if it is or cleared if it is not. R2 is not changed.

TRM

The character indexed by R2 is tested for a legal terminator defined as a blank, tab, comma, semicolon, carriage-return, or line-feed. The Z-flag is set if the character is a terminator and cleared if it is not. R2 is not changed.

LIN

The character indexed by R2 is tested for a legal end-of-line defined as a semicolon, carriage-return, or line-feed. The Z-flag is set if the character is an end-of-line character and cleared if it is not. R2 is not changed.

BYP

Index R2 is advanced past all characters which are blanks or tabs and left indexing the first non-blank, non-tab character it finds.

GTDEC

Index R2 is used to process a decimal number whose value may be from 0-65535 in the input line (leading zeros are legal) and deliver the resultant binary value back in R1. The N-flag will be set if there was an error (result was greater than 65535). R2 will be updated to point to the character following the decimal input number. In the case of an error, R2 will be left indexing the digit that would have caused the overflow past 65535 for double-word processing techniques.

GTOCT

Index R2 is used to process a octal number whose value may be from 0-177777 in the input line (leading zeros are legal) and deliver the resultant binary value back in R1. The N-flag will be set if there was an error (result was greater than 177777). R2 will be updated to point to the character following the octal input number.

GTPPN

Index R2 is used to process a project-programmer number in the standard format of proj,prog and deliver the resultant binary code back in R1. The format dictates that both proj and prog be octal numbers with a value between 1-377. The N-flag will be set if the PPN was not in valid format. R2 will be updated to point to the character following the PPN.

FILNAM

Index R2 is used to process a filename.extension input string and leave the RAD50 packed 3-word result in the 3 words starting with the address specified as the first argument of the call. This argument is a standard monitor call argument in format. The second argument is a 1-3 character extension which is to be used in case no explicit extension is entered in the input string. R2 is updated to index the terminating character. The Z-bit will be set if there was no filename to process (the first character was not a leagl RAD50 character).

PRINTING CONVERSION MONITOR CALLS

There are three calls in the monitor which accept a system unit input and convert the unit to standard printable form and then output it to the user terminal. These calls are used to print out file specifications, filenames, and project-programmer numbers. Each call takes one standard argument which addresses the system unit to be converted and printed.

PFILE

The argument addresses a file DDB and the PFILE call extracts the parameters in the file specification words and prints them in the standard format of dev:filnam.ext[ppn] on the user terminal.

PRNAM

The argument addresses a 3-word filename.extension block (packed RAD50) and the PRNAM call prints the converted result in the standard format of filnam.ext on the user terminal.

PRPPN

The argument addresses a 1-word project-programmer code and the PRPPN call prints the converted result in the standard format of proj,prog on the user terminal.

MISCELLANEOUS MONITOR CALLS

This section deals with the orphan monitor calls who don't seem to have a home in any other section covered thus far. Poor things.

EXIT

This is the normal means that a program uses to terminate processing and return to monitor command mode. The EXIT call takes no arguments. The monitor, upon executing the EXIT call, will delete all temporary memory modules in the user partition and reset any parameters that are program dependent such as JOBICP, JOBBPT, etc. All assigned devices are also released at this time. The user terminal is then placed in the monitor command mode ready to process another operator command. The user effectively forces an early EXIT call from the program when he aborts it by typing a control-c (unless control-c trapping is in effect).

SLEEP

This is a simple call which puts the user job to sleep for a specified number of line clock ticks. The argument is a standard argument which specifies how many clock ticks to sleep for. After the specified number of clock ticks have elapsed the job is automatically awakened and execution proceeds with the instruction following the SLEEP call. Caution - a sleep call with an argument of zero clock ticks will put the job to sleep forever or until awakened by some external event or other job which clears the J.SLP flag in the JOBSTS status word for this job. The normal AM-100 system runs with a clock frequency of 60 Hz so each clock tick therefore has a value of 16.7 milliseconds. Also, the first clock tick may occur any time within the first 16.7 milliseconds (not necessarily a full clock tick).

SCHED

This call forces a job scheduler scan to see if any other jobs are active. It is not normally used by user programs but may be if the user fully understands the job scheduling methods. The normal use of a SCHED call is when some event either sets or resets a scheduling flag in the JOBSTS word of its own or some other job. The SCHED call forces a job scheduling scan which guarantees that the change will be picked up. The SCHED call may be executed from a processor locked condition since the scheduler always unlocks the processor before beginning its scan.

IODQ

This call is used by file service routines and device drivers to dequeue a queued I/O request running in interrupt mode. This call has not yet been fully defined and should not be used until further documentation on interrupt driven processing is released.

FLOPPY DISK STRUCTURE FORMAT

The AMOS monitor supports a flexible floppy disk file system which relieves the programmer of the task of keeping track of files, links and record counts. The structure of the standard floppy disk used in the AMOS system will be described here for those programmers who wish to do some disk file manipulation or system software programming.

PHYSICAL RECORD FORMAT

The standard IBM compatible floppy disk has 2002 128-byte physical records on 77 tracks, each track having 26 sectors numbered 1 through 26. The AMOS system uses a logical record size of 512 bytes (256 words) for each record so the actual record is made up of four standard size 128-byte records on the floppy disk itself. The disk driver routine is responsible for translating the AMOS record number (0-499) to the proper four physical records on the disk. There are only 500 records of 512 bytes each as far as the programmer is concerned and the last two 128-byte records on the floppy disk are lost to his use.

The driver translates the AMOS record number into a starting record number which is four times as great. In addition, a physical sector interleave factor is used so that a 512-byte record requires only one rotation of the disk instead of four which would be the case if an attempt was made to access four physically contiguous sectors on the floppy disk. The interleave factor is 5 meaning that there are four sectors between each logically contiguous pair of sectors. Enough of this, however.

DISK RECORD TYPES

There are six different record types in use in the AMOS system, categorized by their use in the logical processing of files. Each record is 512 bytes long but their internal structure differs due to different usage in the system. The six record types are:

1. Diskette ID record
2. Bitmap record
3. Master File Directory record (MFD)
4. User directory records
5. Sequential file data records
6. Contiguous file data records

The Diskette ID record is always record 0 and is not currently used by the AMOS system. It has been reserved for use by user routines which may want to store diskette identification information in it. It is permanently allocated so it will not accidently be used as a data record by any system routine.

The bitmap record is always record 2 and is used to contain the bitmap for the entire diskette. 32 words of 16 bits each define the allocation of all 500 records on the diskette. The bit is set if the record is in use and cleared if it is free. Words 33 and 34 are a double-word hash total used to maintain bitmap integrity during processing. The remaining 222 words of the bitmap

record are unused. The bitmap record itself is permanently allocated but contains no links to other system disk records. If you destroy the bitmap record you can run the DSKANA program to recover it.

The master file directory record is always record 1 and forms the root of the file structure tree. It contains one entry of four words for each user PPN which is allocated to this disk by the SYSACT program. A maximum of 63 users may be allocated on any one diskette since only one MFD record is available.

The above three record types take care of records 0-2 which are the same on all diskettes. Initializing the disk by using the "I" command in the SYSACT program will write out record 1 (empty MFD of all zeros) and record 2 (bitmap with records 0-2 allocated) which logically clears the disk of all users and files and makes all remaining records (3-499) available. These records are then allocated as either user directory records or file data records.

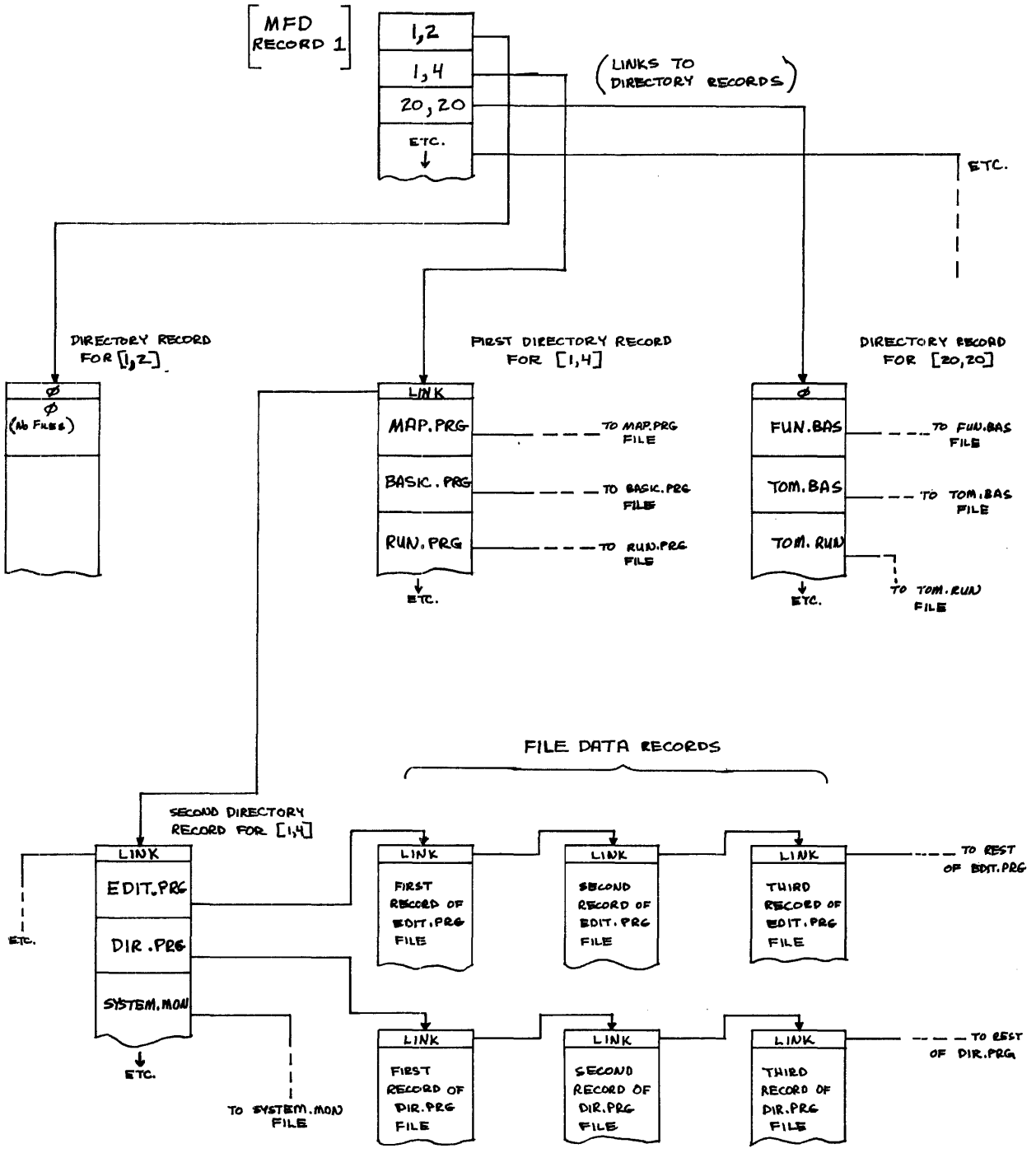
User directory records contain up to 42 entries of six words each to describe user files in the corresponding PPN. The first word of each directory record is a link word to the next directory record in the event that more than 42 files are allocated in the current user area. The final directory record will have a zero link word indicating no more directory records follow.

Sequential file data records have a link word and 255 data words. The link word is the record number of the next record in the file. A zero link word indicates this is the last record in the file. The last record in the file may have anywhere from 0-509 active data bytes in its data area. The directory record item contains this number. Sequential files are normally processed as one long string of bytes from start to finish.

Contiguous file data records have 256 data words and no links. Contiguous files must be allocated as a block of records with no intervening records belonging to other files. Contiguous files must be preallocated before their use while sequential files are allocated one record at a time as they are required. Contiguous files allow random access processing since any record may be located as a direct offset relative to the base record.

FILE STRUCTURE

The file struture is depicted in figure D-1 and resembles a tree with the MFD record as its root. The MFD record has one item for each allocated user on this disk. Each MFD item then contains the record number of the first user directory record for that PPN number. The user directory record has one item for each data file in this user's area. Each directory item then contains the record number of the first data record in he file. Sequential files then chain through the data records by link words as shown in the diagram. The two files that are partially depicted are EDIT.PRG and DIR.PRG in user area [1,4] which just happens to be the system program area. Contiguous files have no link words and must occupy physically adjacent records beginning with the first record as addressed in the directory item. Contiguous files are not depicted in the diagram since they are so pleasingly simple and besides, I ran out of room on the paper.



FLOPPY DISK FILE STRUCTURE

FIG D-1

MFD ITEM FORMAT

Each MFD item is four words long and contains the PPN, user directory link, and password. The format of the item is:

Word 1 - user PPN (proj and prog are each one byte)
Word 2 - record number of first user directory record
Words 3-4 - password packed RAD50 (up to 6 characters)

Word 2 is zero if no files have been allocated to this user yet meaning no directory records have yet been allocated. Words 3-4 are zero if no password is required to gain access to this user account when logging on via the LOG command.

MFD items are added, deleted, and changed by the SYSACT program.

USER DIRECTORY ITEM FORMAT

Each user directory item is six words long and contains information about the data file which it defines. The format of the item is:

Words 1-3 - filename.extension of the file packed RAD50
Word 4 - number of data records in this file
Word 5 - number of active data bytes in last record
Word 6 - record number of first data record in file

Word 1 is -1 (octal 177777) if this file has been erased and the directory item is available for another file definition. Word 1 is zero to mark the logical end of the user directory. The byte count in word 5 is negative if this is a contiguous file. It will also represent the negative active byte count of the file if the contiguous file has been opened for output and written into sequentially.

SYSTEM COMMUNICATION AREA

There is an area in monitor memory starting at location 100 (octal) which is called the system communication area. It is defined mnemonically in SYS.MAC and contains specific parameters that deal directly with singular system resources and root addresses. The user should not mess aroun' with these parameters 'cause he might create a disaster. They are briefly defined here for those users who wish to gingerly reference them for whatever diabolical reasons programmers dream up. All references to these parameters should be made symbolically in the absolute addressing mode. An example: the instruction MOV @#JOBTBL,R0 should be used to set the base of the user job table into index register R0. Get the picture?

SYSTEM

This word is used to contain system attribute and status flags. Currently it is only used to indicate that the system has been properly loaded when bit 0 is set on. Don't count on this to remain the case, however.

DEVTBL

Set up by the DEVTBL program in the system initialization command file, this word contains the absolute address of the device table in monitor memory. The format of this table remains a mystery to all.

DDBCHN

This is the base of the active DDB chain for interrupt driven routines. It is set up and altered by the file service routines as new I/O DDB's are queued for transfer requests and goes to zero each time that there are no requests pending. Not used for non-interrupt driven devices.

MEMBAS and MEMEND

These two words define the beginning and end of the complete user memory area. MEMBAS is the address of the first word following the complete resident monitor including the system memory area for user resident programs. MEMEND is the address of the last word in the total physically contiguous RAM memory in the machine. It is set up by the INITIA program when the monitor first starts up by a memory scan technique which locates the last available 1K bank.

SYSBAS

This is the address of the system memory area which is used to contain any user programs set up by the SYSTEM command in the system initialization command file. It is zero if no system memory area exists.

JOBTBL

This is the address of the user job table which contains one JCB entry for each user allocated via the JOB command in the system initialization command file. For a complete description of the job table and JCB entries refer to the section titled JOB SCHEDULING AND CONTROL SYSTEM.

JOBCUR

This word always contains the address of the JCB for the job that is currently running and has control of the CPU. For the user program, it will always point to your own JCB as long as you are running. Obviously if you are referencing this word you must be running, right? JOBCUR is updated only by the job scheduler in the timesharing monitor.

JOBESZ

This word is set up by SYSGEN which builds the monitor and contains the size in bytes of the JCB entry in the job table. This way, when the JCB item expands the programs which scan the job table will not have to be reassembled since they get the JCB size dynamically from JOBESZ. This includes routines within the monitor itself.

TIME

This is a two-word field which is incremented each time the line clock interrupts. It represents the current time of day, or at least it will once I write a simple program to accept the current time and convert it to preload the TIME parameter. At any rate, it is the parameter which you can reference if you want to keep track of the time it takes to do something on the machine. Remember, TIME is used to count clock ticks and not seconds or milliseconds. To calculate the actual time in seconds you should divide the elapsed time in ticks by the clock frequency which is stored in the CLKFRQ constant described further on. This of course optimistically assumes that the CLKFRQ command has been used in the system initialization command file to properly set the constant up for your particular frequency (50 Hz overseas, remember?).

JOBTIM

This word is used by the job scheduler to keep track of the amount of time the current job has been running for the present time-slice. Each time the job scheduler gives the CPU to another job for running, its software priority count is transferred from JOBPRI (in its JCB) to the JOBTIM constant here. Each line clock interrupt will decrement this counter until it goes to zero and then - POW - no more time for this job and another job gains control of the CPU. That's the way the old timesharing ball bounces. It is considered cheating for a user job to occasionally bump this constant up to some higher value to gain more time for its time-slice. Shame on you for even thinking of such a dastardly deed.

HLDTIM

This is a two-word area which controls the head-load timing for the AM-200 floppy disk system. The second word (at HLDTIM+2) is set up by the HEDLOD program in the system initialization command file to the number of clock ticks desired to wait before unloading the disk heads during periods of inactivity. Each time the head is loaded or another disk transfer is initiated, the count in the second word is transferred to the first word. Each time the clock interrupts, the count in the first word is decremented and if it ever gets to zero - ZAPPO! - the head is unloaded. Nifty, huh?

CLKFRQ

This word is set up by the CLKFRQ command in the system initialization command file to contain the frequency at which the line clock is running. It is used by routines which compute elapsed time based on counting the clock ticks in the TIME constant. Normally set to 60 for systems in North American countries and to 50 for systems running overseas. Achtung!

DYSTAT

This is the address of the resident DYSTAT program entry point for asyncronous updating of the video monitor. It is set up when the DYSTAT program is initially executed by a command in the system iniaitlization command file. If this word is zero it means the DYSTAT system is inactive.

TRMSER

This is an index to the terminal service routine control table in the TRMSER program within the monitor. It has not been totally implemented yet so don't mess around with it.