AIMS-11 PROGRAMMING MANUAL

from ARBAT(UK)LTD

March 1977

Issue 6

PDP-11 AIMS

CONTENTS

AIMS	Version 2	April 1973	Manual Issue 3
AIMS	Version 3	September 1974	Manual Issue 4
AIMS	Version 3N	August 1975	Manual Issue 5
AIMS	Version 5	March 1977	Manual Issue 6

1. INTRODUCTION

AIMS is a system and application programming language with a syntax based on that of Dartmouth College BASIC.

The main extensions are:-

** The string handling is much improved. String relational operators allow for anchored or embedded searches, and tests for string equality, inequality or telephone-directory type comparisons.

** A new PUT command provides keyword-searching, pattern matching, and string decomposition facilities. These allow the writing of conversational programs which can communicate with untrained users in an approach to natural language.

** AIMS is fully interpretive and is specially designed to allow close interaction with a terminal. All AIMS commands may be executed either as part of a stored program or directly at the keyboard. A running AIMS program may be stopped at any time, the values of variables may be changed, or the program may be edited, and then continued. This makes debugging easy.

** Wherever a number may appear in BASIC, AIMS will accept a numerical expression of any complexity.

** Several commands may be put on one line, and any statement may follow an IF command.

** Numerical expressions may include Boolean, logical, and relational operators.

** All program data, including strings and arrays, is stored in AIMS lines. This allows arrays to be declared or re-dimensioned at runtime, and makes it easy to preset or examine string variables.

** All error conditions may be trapped so that an AIMS program can always retain control if it wishes. This allows the construction of supervisor programs that can structure the interaction between system and user in any desired manner.

** Extensive file handling facilities allow AIMS programs to create, delete, rename, read and write files, either in text or in binary form. It is possible to use an AIMS array as a memory buffer allowing random access to arbitrarily-formatted data. Facilities are provided for saving AIMS programs on a bulk-storage device, either in binary-image form, or as an ASCII file. A running AIMS program may transfer control to such a saved file, either in an overlay mode or in an interleaving mode in which some lines remain unchanged.

** AIMS programs may run in a variety of privileged modes, allowing the construction of applications or system-level programs that are protected from examination or interference by the user.

** All administrative programs for the multi-user system are written in AIMS, allowing the system manager to determine completely the appearance of the system to its users, the login and logout procedures, access restrictions, and so on.

** User memory partitions expand or contract dynamically to meet changing requirements, giving optimum overall memory utilisation.

** User memory partitions are swapped automatically onto disk if the memory requirements exceed the physically available memory. Swapping is fully overlapped with computation and may be done on several disks if desired.

INCOMPATIBILITIES WITH DARTMOUTH BASIC

Calculations are performed using variable-precision integer arithmetic, rather than floating-point.

FOR, NEXT, READ, and DATA statements are not implemented in AIMS. However the extended facilities allow these commands to be emulated with ease.

The END statement is not recognised in AIMS. There is no need to mark the end of the program.

The matrix operations in extended Dartmouth BASIC are not implemented.

DIFFERENT VERSIONS OF THE SYSTEM

Every AIMS system comprises two parts: an operating system and an interpreter. The operating system deals directly with the hardware, controls all peripheral devices, and generally provides a stable environment in which other programs, such as the interpreter, may be executed. The interpreter implements the AIMS language as it is described in this manual.

At present there is a choice of two different operating systems: (1) DOS, a system originally developed by DEC but no longer supported by them, and (2) MONITOR, a system developed and supported by Arbat. DOS will run on the smaller hardware configurations, whereas MONITOR is more powerful but requires extra main memory. Each operating system requires its own version of the interpreter because the systems differ internally. However, the two interpreters both implement the same AIMS language so that the combination of operating system and interpreter is compatible at the application program level. A DOS-based system may be upgraded to a MONITOR-based one without changing the application software (apart from a few details mainly affecting system utility programs).

The two combinations are:-

Op sys: DOS Interpreter: AIMS version 3

Op sys: MONITOR Interpreter: AIMS version 5

Material in this manual that relates to only one of the operating systems is marked with a 'DOS' or 'MON' flag, and similarly differences between the two interpreters are flagged with a 'V3' or a 'V5'.

HARDWARE SUPPORTED

MONITOR is capable of running on any PDP-11 central processor that has the Memory Management facility (both types of KT11 are supported). The 11/35 and 11/40, which have the more restricted type of memory management hardware, cannot support as many jobs as the 11/45 upwards.

All mixtures of core, mos and bipolar memories are supported, with or without the memory parity options. MONITOR automatically uses all the memory that is found to be present.

All available DEC asynchronous line interfaces are supported.

MONITOR is continuously maintained and it will normally be found that the software will support the most cost-effective combination of hardware that is available at any time. However, since it is not known what devices may become available in the future, the above statement should not be taken as a commitment to support all possible configurations.

2. COMMAND SYNTAX AND LINE STRUCTURE

A stored AIMS program is built up by typing in numbered lines at the keyboard. As each line is typed in, it is checked to see that the commands are legal and is then coded into a compact form and stored. Lines may be typed in any order, they are automatically sorted into order by line number.

If a line is typed with the same number as an existing line, it will replace the old line. An existing line may be deleted by typing the line number by itself.

If a line does not begin with a line number, the line is executed immediately rather than being stored away. For example

```
>PRINT 5*6        [the > sign is printed by AIMS when
 30               [it is ready for a command
>
```

Several commands may be typed on the same line by beginning each new command with a colon. For example

```
>LET X=2 :PRINT 5^X
 25
>
```

Commands may be abbreviated like

```
>L X=2:P5^X
 25
>
```

The abbreviation will be properly understood only if it ends with a character that is not a letter. For example

```
>P5
  5
>
```

is ok, but

```
>PX
```

is interpreted as the unknown command PX, rather than the intended PRINT X. A space may be used to delimit such abbreviations.

These abbreviations are automatically reconstituted when a stored program is listed, giving the user the advantages of quick type-in and nicely formatted listings. For example

```
>10L J=0
>20P J J^2:L J=J+1:IF J<5:LOOP
>30P'FINISHED'
>RUN
  0  0
  1  1
  2  4
```

```
  3  9
  4 16
FINISHED
>LI
 10 LET J=0
 20 PRINT J J^2 :LET J=J+1 :IF J<5 :LOOP
 30 PRINT 'FINISHED'
>
```

SIGNIFICANCE OF SPACES AND COMMAS

Spaces in command lines are generally ignored. They may be inserted to improve legibility, but it should be borne in mind that extra spaces increase the size of the program and reduce the execution speed (see section 20).

There are a few contexts in which spaces are significant:-

```
PRINT XY              [prints the value of the variable XY

PRINT X Y             [prints the value of X followed by that of Y

P X                   [the space delimits the abbreviated PRINT
                      [command

LET X=6 Y=5           [is legitimate, but it could be written as

LET X=6Y=5            [to save space
```

There are other contexts in which a comma is necessary to resolve ambiguity:-

```
PRINT X  -Y           [prints the value of X-Y

PRINT X,-Y            [prints the value of X followed by
                      [that of -Y

PRINT A+B (J+X)/5     [is an array reference, whereas

PRINT A+B,(J+X)/5     [is not
```

NOTATION

In the following descriptions we shall use square brackets like [ ] to enclose comments and names representing elements of the syntax. For example

[number] represents any number such as 1, 123, 128 etc.

[ne] represents any numerical expression such as 1+2, 123, 1+X*(X^2) etc.

[se] represents any string expression (see section 6).

Anything not enclosed in square brackets stands for itself.

Note that it is never necessary to use square-brackets when typing commands to AIMS (except for department/user numbers as explained in section 13). The brackets are used in this manual simply as an aid to clarity.

Most of the examples are shown exactly as they would appear to a user typing at a terminal. AIMS always prints a > or * sign when it is waiting for input from the user. Hence it may be assumed that all lines beginning with > or * are typed by the user, and all other lines are printed by AIMS.

When we wish to add a comment on a line that contains an example, the comment is preceded by a [to separate it clearly from the example.

^ denotes the up-arrow key, usually shift N.

_ denotes the back-arrow key, usually shift O.

is usually shift 3. It may be marked as a pound sign.

\ denotes the back-slash key, usually shift L.

The notation 'control-X' represents the single character that is typed by holding down the key marked 'CTRL' and pressing the key marked 'X'.

A vertical line down the left margin indicates new material that has been inserted into the manual since the last issue.

3. GETTING STARTED WITH AIMS

An AIMS system may be used in broadly two ways: programmers use it to develop programs that provide some desired service, and people with no computer expertise make use of this service. For example a team of programmers could use the AIMS language to implement an order-processing system, and this might then be used by the sales personnel in a mail order firm. The appearance of the system to the end-user is determined entirely by the programs with which he interacts and is thus beyond the scope of this manual. We are concerned here with the way in which an AIMS programmer uses the system.

Before any programming can be done the user has to gain access to the system by 'logging on'. This is done by pressing the carriage return key which causes the system to ask for your department/user numbers and password. These will be allocated to you by the system manager. A typical login sequence looks like this

```
                        [User presses carriage return key
MONITOR V1A AIMS V5B J4-K6
DEPT,USER: 100 110      [User types his numbers
PASSWORD:               [User types his password which is not printed

.                       [The dot indicates executive level.
```

Note: when you key in your department and user numbers they may be separated either with a space or with a comma. A space is preferred because comma does not work in European countries where it is used as a decimal point (see page 32).

The password is a security measure to prevent unauthorised people from using the system.

Once a user is logged in he is communicating with the system executive program which prints a dot when it is waiting for a command. This executive program provides a range of services which are useful when developing programs. These are detailed in section 18. For the present it is sufficient to note that there is one executive command, E, which allows the user to write an AIMS program:

```
.E                      [User types E to enter edit mode
>P 5*6                  [Where he can give AIMS commands.
  30
```

The E command transfers the user from executive-level to AIMS-level where he is communicating directly with the AIMS language interpreter. At this level the user can give direct commands (ie. those which are executed immediately like the PRINT command above), or he can type in a stored program like this

```
>100 LET N=0 :PRINT 'CUBES'
>110 PRINT N N^3 :LET N=N+1 :IF N<4 :LOOP
```

The user may now run the program by giving a RUN command:

```
>R
CUBES
   0   0
   1   1
   2   8
   3  27
>
```

The program stops running when there are no more lines to execute. The user may modify the program if he wishes either by changing existing lines or by inserting new ones:

```
>120 PRINT 'DONE' :EXIT
```

At any time the user may return to executive-level by pressing the control-O key or by giving the EXIT command. But it is important to note that on returning to executive the system forgets entirely about the program that you were developing. So if you want to preserve this program for future use it is essential to save it on a disk file. This is done using the SAVE command as follows:

```
>S'CUBE':P'OK
OK
>
```

This saves the program as a disk file called CUBE.BAS as explained further in section 15. The user may now safely return to executive level:

```
>EXIT
.
```

and other activities may be pursued. Later on, the saved CUBE program may be executed by giving its name like

```
.E CUBE
CUBES
   0   0
   1   1
   2   8
   3  27
DONE
.
```

The program is recalled from disk and executed. At the end the EXIT command in line 120 transfers control back to executive. If you want to continue developing the program it may be more convenient to recall the program without executing it. This is done by using the CALL command rather than the EXECUTE one:

```
.C CUBE
>LI
 100 LET N=0 :PRINT 'CUBES'
 110 PRINT N N^3 :LET N=N+1 :IF N<4 :LOOP
 120 PRINT 'DONE' :EXIT
>
```

The program may now be modified and tested and SAVEd again as file CUBE.BAS or with a different name if you want to preserve the old version as well.

When you have finished using the system it is essential to 'log off' using the BYE command. This returns the terminal to its initial state in which it is necessary to press carriage return and go through the log on procedure before the terminal can be used again. If you go away without logging off there is an opportunity for an unauthorised person to interfere with your files.

The BYE command may be given either at executive-level or at AIMS-level:

```
.B
11:15:34 3-FEB-77 J4 K6 100110 user name
Run=0:00:10 Connect=0:01:40 DK disk=216 Bye
```

The message is printed by the system to identify the user who has just logged off. It also gives the time of day, the amount of disk space being used by that user, and his run and connect times for the session. Connect time is the elapsed time between log on and log off, and run time is the amount of central processor time devoted to your job.

4. EDITING COMMANDS

LINE EDITING

When typing commands or data to AIMS, certain characters have special functions:-

RUBOUT — deletes the last character typed, echoes as \

control-Y — cancels the whole line, echoes as @ [newline]

control-X — switches program-generated printout on and off. May be used to suppress a section of printout if the user is not interested in it. Press control-X again to restore printing.

control-S — Pauses printout. Useful with visual displays to give you time to read output before it rolls off the screen.

control-Q — Resumes printout after a control-S pause.

If a ! is echoed when you type a key, this indicates that the computer's input buffer is full and that the character has been ignored. Wait for the computer to catch up before typing more.

PROGRAM EDITING

LIST — List the whole program.

LIST [ne1],[ne2]
List lines [ne1] to [ne2] inclusive. If ,[ne2] is absent it is taken as infinity.

[number] — Delete line [number].

CLEAR — Delete the whole program.

CLEAR [ne1],[ne2]
Delete lines [ne1] to [ne2] inclusive. If ,[ne2] is absent it is taken as infinty.

X[ne] — Print line [ne].

X[ne][string1][string2]
Change the first occurrence of [string1] to [string2] in line [ne]. The strings should be enclosed in quotation marks (see section 6). When the X command is used directly (ie. not in a program), the modified line is printed. The line number itself may be changed, in which case the line will be duplicated in its new position.

Warning: the X command should not be used to examine arrays, see section 12.

Examples

```
>LIST
 10 LET J=0
 20 PRINT J
 30 LET J=J+1
 40 IF J>10 :GOTO 20
 50 PRINT 'DONE'
 60 GOTO 10
>20                [deletes line 20
>LIST 10,30
 10 LET J=0
 30 LET J=J+1
>X40'>''<'         [changes greater-than to less-than
 40 IF J<10 :GOTO 20
>X10"0""5"         [changes the line number from 10 to 15
 15 LET J=0
>CLEAR 50          [clears from line 50 upwards
>LI
 10 LET J=0
 15 LET J=0
 30 LET J=J+1
 40 IF J<10 :GOTO 20
>X40'TO''S'        [changes command word
 40 IF J<10 :GOSUB 20
>
```

When writing a program it is a good idea to number the lines in steps of 10, 5 or 2, so that there is room to insert corrections later. A resequencing program RESEQ.BAS is available for changing the line numbers if large-scale alterations are needed.

5. NUMERICAL EXPRESSIONS

A numerical expression is something that can be evaluated to produce a number. Expressions are in normal infix-operator form. The following operators are allowed:-

PREC	OPERATOR	MEANING
1	-	Unary minus
2	_	a_b: open bit shift of a by b places left if b>0, right if b<0
3	^	Exponentiation
4	/	Division
5	*	Multiplication
6	-	Subtraction
7	+	Addition
	Relational operators	
8	<	Less than
8	=	Equal
8	>	Greater than
	String operators	
9	>	Alphabetically greater than
9	=	Identical
9	<	Alphabetically less than
9	^	a^b: true if string-a begins with string-b
9	_	a_b: true if string-a contains string-b
10	' or "	Quotes enclosing literal string
10	$	Right-associative operator meaning string-name
	Logical operators	
11	&	AND
12	\	Exclusive OR
13	!	Inclusive OR

(section 3 explains where ^ _ and \ are found on the keyboard)

The higher precedence (ie. lower numbered) operators are applied first, and operators with the same precedence are applied from left to right. Round brackets like ( ) may be used to control the order of evaluation, the most deeply nested sub-expressions being evaluated first.

The operators <, >, and = may be combined in any order with an effect derived from the inclusive-OR of the individual conditions. Thus <= means less-than-or-equal-to, and <> means not-equal-to.

The relational operators yield either -1 or 0 according to whether the relation is true or false respectively.

The logical operators perform their operations on their arguments regarded as bit patterns.

Calculations are done using integer arithmetic and fractional results are rounded down to the next whole number.

As a side-effect of a division operation, the system variable QA is set to the positive value of the remainder, truncated to 16 bits. QA thus gives the remainder accurately provided the divisor was less than 32,768.

Examples

```
>P 1+1
  2
>P 3+5/2 QA          [the division occurs first
  5    1             [QA gives the remainder
>P 3^2               [three squared is nine
  9
>P 3*5/2             [the division occurs first
  6
>P (3*5)/2           [the multiplication occurs first
  7
>P -7/2 QA           [the unary minus is applied first
- 3    1             [QA is positive remainder
>P 4<6               [4 is less than 6 so the value is -1
- 1
>P 4<3               [4 is not less than 3 so the value is 0
  0
>P 7&5               [111 ANDed with 101 is 101
  5
>P 5!2               [101 ORed with 010 is 111
  7
>P 7\5               [111 exclusive ORed with 101 is 010
  2
>L X=7 Y=4           [sets X to 7 and Y to 4
>P X<8!Y=6
- 1                  [although Y is not 6, the value is -1 since
                     [X is less than 8
>P X<>Y
- 1                  [X does not equal Y so the value is -1
>P 1+(X+Y)/2
  6
>P 1_15              [1 shifted left 15 bits.  Shifting is
 32768               [equivalent to multiplying by a power of two
>P 65536_-15         [2^16 shifted right 15 bits
  2
>
```

We shall deal with the string relations later.

We shall use [ne] to denote any numerical expression in future. There are some commands in which a [ne] may optionally be omitted. In these cases a context-dependent default value is used, as noted in the command descriptions. Elsewhere, if a [ne] is expected but is not present, a default value of zero is used.

VARIABLE NAMES

User-defined variable names may be upto two alphanumeric characters long and must begin with a letter. For example A, X, A1, AA, Z9.

Warning! All names beginning with the letter Q are reserved for use by the AIMS system, and such names should not be created by programmers.

REPRESENTATION OF NUMBERS

AIMS treats all numerical values as signed integers. The value of each variable is stored internally as a particular number of 16-bit words holding the value in 2's complement binary. The number of words used to hold each value is called the precision or length of the variable. This length is the same for all variables and is normally set at 2 words. The length is important because it determines the largest number that can be stored in a variable or used in a calculation (see below).

ACCURACY OF CALCULATIONS

Since numerical expressions are evaluated with finite precision, some loss of accuracy can arise during a calculation. For example suppose X is a quantity that requires two words to represent it. Then the expression X^3 could require six words, and X^5 ten words, and so on. For reasons of efficiency it is not worthwhile to cater all the time for these very large numbers that will seldom arise in practice.

Hence, numerical expressions are evaluated to a fixed precision of between 1 and 7 words. The precision is determined by the system function EP() which may be set by the user (see section 11). For example if the command

```
LET EP()=3
```

is executed, all numerical expressions will be evaluated to 3-word precision. Error ?0 will occur if a calculation generates a value that cannot be represented in EP() words.

The following table shows the size of numbers that can be handled for each value of EP():-

EP()	Largest Positive Number
1	32,767
2	2,147,483,647
3	140,737,488,355,327
4	9,223,372,036,854,775,807
5	604,462,909,807,314,587,353,087
6	39,614,081,257,132,168,796,771,975,167
7	2,596,148,429,267,413,814,265,248,164,610,047

The largest negative number for each precision is one more than the entry in the table.

EP() is initially set to 2, allowing fast calculations on numbers less than 2000 million. If the user is likely to generate numbers larger than this he should set EP() appropriately beforehand. EP() may be adjusted at any time by a running AIMS program.

The setting of EP() also determines the length of all user-defined variables. If EP() is increased, all the variables in existence at the time are extended to the new length. This operation does not affect the values of the variables. Similarly, if EP() is reduced all variables are made to fit into the new smaller length. This is done by deleting an appropriate number of words from the internal binary representation of each variable, starting with the most significant word of the value. This process will not affect the value of the variable provided it can be represented within the new smaller length. If the value is too large for the new length the variable is marked as being undefined, and this will cause a ?U error the next time that variable is referenced.

DATA TYPES

AIMS supports four different types of data:-

1) Simple Variables: with names like J, X1 and QA. Each simple variable will hold one numeric value. Simple variables are either

   1.1) User-defined: these are defined when the user first sets them, like LET J=2. Do not use names beginning with Q.

   1.2) System-defined: these 'system variables', like QA and QE, are permanently defined and fixed length, see section 11.

2) Arrays: with names like A(J). An array is like a table or vector of cells. Each cell will hold one numeric value. Arrays are defined by the user by means of an ARRAY command.

3) String variables: with names like $1. Each string variable will hold one string comprising any number of characters, see section 6.

4) System functions: with names like EP() and DA(). These are rather like arrays except that they are permanently defined by the system, see section 11.

ARRAYS

Arrays are declared like

```
>10 ARRAY B 7
```

which creates an 8-cell array in line 10. The array name B is treated like a simple variable. When the RUN command is executed the array name will be assigned a value equal to the line number (10 in this case).

Arrays are referenced conventionally like for example

```
>LET B(J)=123 :PRINT B(J)
 123
>
```

where B is the number of the array line, and J is the subscript.

Either B or J may be a numerical expression, so that both the subscript and array name may be computed if desired.

```
>LET (B+5)(J)=C(J+2)
>
```

copies the J+2'th cell of the array C to the J'th cell of the array in line B+5.

References of the kind just described operate on single-word values. Multiple-word values may be stored in arrays, provided the precision is specified explicitly:-

```
>LET B(3,J)=X
>
```

stores the triple-length value X in the array B. The value will occupy cells J through J+2 of the array.

For single-length values, the array subscript J may run from zero upto the dimension specified in the array declaration. For multi-word values, allowance must be made for the additional words occupied.

Restrictions

Arrays must be stored in lines. An array cannot be declared by a direct command. There must not be any other commands on the

same line as an array declaration. The dimension given in an array declaration must always be a number, it may not be an expression.

The CODE command may be used to set up arrays with computed dimensions, see section 12.

Array Initialisation

All the cells of an array are zeroed when it is created. This occurs when the ARRAY line is typed in, or CODEd (section 12), or CALLed from a file (section 15). Once an array has been created, the cells retain their values unless altered by the user.

Note that the cells are not affected by the RUN command, nor by execution of the ARRAY line itself. The mere existence of the ARRAY line is sufficient to create the array, and execution of the line never has any effect apart from causing a slight delay.

When a program containing an array is DUMPed (see section 15), the values stored in the array are written to the dump file and will be restored by a subsequent LOAD. In contrast, when a program is SAVEd the array content is not written to the file and the array will be initialised to zero if the program is later CALLed.

NUMERICAL OVERFLOW

By numerical overflow we mean the attempt to handle a value that is too big. AIMS will abort the operation and give a ?0 error (section 16). Numerical overflows can occur in two contexts:

1) During the evaluation of a numerical expression, if a number is generated that cannot be represented within EP()*16 binary bits. This can be cured by increasing EP().

2) During an assignment, if the value being stored will not fit into the specified destination. If the destination is a user-defined variable, this can also be cured by increasing EP().

The second type of overflow can occur when assigning to system variables like QG, and to system functions like DA(). These overflows cannot be cured since all system-defined objects are of fixed length. The overflow indicates that you are trying to store a value that is too large.

The second type of overflow can also occur when assigning to an array. In this case the length of the destination cell in the array is determined either by default, as in

```
LET A(J)=X
```

or explicitly by the programmer, as in

```
LET A(2,J)=X
```

If you are prepared to take up more space in the array, you can avoid the overflow by increasing the destination length, as for example by

```
LET A(3,J)=X
```

Programmers are sometimes surprised when an overflow occurs sooner than they expected. This is usually due to the following effect: consider the operation of storing a value in a 1-word array cell like

```
LET A(J)=40000
```

In this case the value 40000 is being assigned and this has a binary representation of 1001110001000000. Although this is just sixteen bits long, an overflow error will in fact occur. This is because the sixteenth bit of the value is a 1. If such a value were stored in the 1-word array cell, the sign-bit of the cell would be set to 1, causing the value to appear negative when later extracted. Thus the action of storing and retreiving a value from an array cell would have had the undesired effect of changing its sign. This is why the overflow error is given.

A similar effect happens with 2-word cells and with all other lengths. The largest value that can be assigned without giving an overflow can be obtained from the table given earlier.

Automatic Truncation

There is one situation where the overflow detection can be a nuisance. This is where the values being stored in an array are not regarded as signed numbers. For example, the programmer may be using the array to hold bit patterns representing yes/no answers to a questionnaire. If one of the patterns happens to have the sixteenth bit set, this would cause an unwanted overflow error.

Overflows can be suppressed by using the operator =@ in place of just = in the assignment:

```
LET A(J)=@40000
```

This operator truncates the value to the length of the destination and stores it without any overflow checking.

It is important to realise that the use of this operator simply suppresses the overflow check, it does not prevent the sign-change that results from the overflow. When the number is extracted from the array it will still be changed:

```
LET A(J)=@40000 :PRINT A(J)
-25536
```

The original value may be recovered by masking off the extended

sign bits when extracting the value from the array. A suitable mask is the number

```
1_L - 1
```

where L is the length of the array cell in bits. This quantity in binary is just an L-bit mask of all ones. For a single-word cell the mask is 1_16-1 which is 1111111111111111 in binary.

```
LET A(J)=@40000 :PRINT A(J)&1_16-1
40000
```

When writing a program that does a lot of bit manipulation of this kind it is a good idea to set up the mask in a variable at the beginning, so as to avoid having to work it out each time:

```
100 LET Z=1_16-1
```

Unpacking bytes from arrays

When a value is read from an array the most significant bit is taken as the sign-bit and this is extended to EP() if necessary. Bytes may be unpacked from arrays by dividing A(J) by 256 giving the lefthand byte as the quotient and the righthand byte in QA. If the lefthand byte exceeds 127 A(J) will be negative and since division always produces a positive remainder, QA will not give the righthand byte correctly. The expressions (A(J)&65535)/256 and (A(J)/256)&255 do not always produce identical values in QA.

Errors caused by array references

Errors are described in section 16. An array reference like LET A(J)=X can cause five different errors:

```
?U    either A or J (or X) is undefined
?L    there is no line numbered A
?T    line number A is not an array
?V    J is less than zero or greater than the array dimension
?O    the value of X is too big to fit in A(J)
```

Programmers sometimes make references to arrays by mistake:

```
PRINT X (Y+Z)/3
```

is treated as a reference to the array cell X(Y+Z). You need a comma between the X and the (. Similarly

```
PRINT #5 (Y+Z)/3
```

refers to an array in line 5 but the programmer had intended #5 to specify an I/O channel (section 13).

ARRAY OPERATORS - ACOMP, AMOVE

The command

```
ACOMP A(J) B(K) N
```

compares the contents of array A with that of array B. The command succeeds if array cells A(J) through A(J+N-1) are identical to cells B(K) through B(K+N-1). If a difference is found the command fails and A(J+QA) is the first cell to differ.

The command

```
AMOVE A(J) B(K) N
```

moves the block of N cells beginning at A(J) to a new position beginning at B(K). A may be equal to B if it is desired to move information within one array. When A=B an overlapping block move will occur if K is sufficiently close to J. AIMS takes care of this is an appropriate manner so that overlapping moves in either direction are correctly performed.

There are also two commands, PACK and UNPACK, which convert between strings and arrays (see page 34).

THE SCAN COMMAND

In commercial applications it is often necessary to file a large number of fixed-length records containing customer account numbers, stock lists, and so on. If there is a need to access the records via some key, such as the account number, it will be necessary to organise an index of some sort. The way in which this is done will depend to some extent upon the application and is beyond the scope of this manual. In most cases the operation of finding a record via the index will involve scanning through one or more arrays looking for a match with the wanted key. Because of the interpretive nature of the AIMS language this array scanning is rather slow and may seriously limit the system performance in situations where large indexed files are heavily used.

The SCAN command eliminates this problem by providing a fast way of scanning an array for a given key. The syntax is:

```
SCAN [recsize ne] [keylen ne] A(J) [mode] [key ne] [count ne]
```

The array A() is assumed to contain a number of records each of which is [recsize ne] cells long. Each record contains a key which is [keylen ne] cells long. A(J) specifies a starting position within the array. [key ne] is the key to be searched for. [count ne] if present specifies the maximum number of records to be scanned.

The SCAN command searches the array from the starting position and compares [key ne] with the key within each record, in a manner determined by [mode]:

```
  =  Scans till a record is found with recordkey=[key ne]
 <>  Scans till a record is found with recordkey<>[key ne]
 <=  Scans till a record is found with recordkey<=[key ne]
 >=  Scans till a record is found with recordkey>=[key ne]
```

If the appropriate condition is met before the end of the scan is reached, the SCAN command succeeds and A(J+QA) points to the found recordkey. If no suitable record is found the SCAN command fails and QA is not set.

Two further modes are provided which always scan all [count ne] records (or to the end of the array if there are fewer records than the count):

```
  <  Finds the largest recordkey<=[key ne]
  >  Finds the smallest recordkey>=[key ne]
```

These latter two modes may be used for sorting.

The keys are treated as [keylen ne]-word numbers which are compared arithmetically. Hence EP() must be at least [keylen ne] when the SCAN command is used.

Example

Suppose our records are 8 words long and the first 2 words of each record contain the key. If we use a 512-word array we can read 64 records from the file at a time and scan for the key K using a program like

```
  8 ARRAY BF 511

400 READ #6 BF() :GOTO 410
405 PRINT '?READ ERROR' :STOP
410 SCAN 8 2 BF()=K :GOTO 420
415 GOTO 400
420 PRINT 'FOUND KEY'K' AT RECORD'(PT(5)-512+QA)/8
```

If the keys do not begin at the first word of each record an offset can be specified in the [array] part of the SCAN command. For example

```
SCAN 8 2 BF(3)=K
```

tests the key in the fourth and fifth words of each record.

Note that whatever the significance of the keys to the user, they are treated as multi-word numerical values by the SCAN command. Negative keys may be used and these are ordered in the usual way (ie. the largest negative number comes first).

[keylen ne] must be less than or equal to EP().

6. STRINGS AND STRING EXPRESSIONS

Strings are normally stored in string variables. For example the statement

```
>1$HELLO
```

will create the string variable $1 containing the string HELLO. We may now refer to this string in a print command like

```
>PRINT $1
HELLO
>
```

CHARACTER SET

There is no restriction on the characters that may be stored in strings. The full 128-character I.S.O. set is allowed, including upper and lower case letters. In particular, the null string (ie. a string containing no characters) is allowed. Strings may be of any length, but very long strings are expensive (see section 20).

Such strings are actually stored as lines, so we can list them in the same way that we can list a program:

```
>10 P $1$2
>2$ THERE
>LIST
  1 $HELLO
  2 $ THERE
 10 PRINT $1$2
>RUN                    [Lines 1 & 2 cause no effect when the program
HELLO THERE             [is run.  The message is printed by line 10.
>
```

String variables are usually called 'dollar-lines'.

Another way of storing strings is to type them out literally inside quotation marks. For example

```
>PRINT 'HELLO'
HELLO
>
```

Since these strings are embedded within a program line, they cannot be changed except by changing the whole program line. Literal strings are thus best used for constant strings, such as short messages or strings that we want to search for.

STRING TERMINATORS

Strings may be read from peripheral devices by means of the INPUT command. For example

```
                        [The * is automatically printed by this form
>INPUT $1 :LIST         [of the INPUT command as a cue to the user.
*HO                     [User types 'HO' followed by carriage-return
  1 $HO
>
```

When reading strings in this way the character carriage-return is treated as an end-of-string delimiter which is removed before the string is stored. Thus $1 above contains just the two letters HO

```
>P $1$1
HOHO
>
```

Since printed information is usually required on separate lines, the PRINT command normally appends carriage-return and linefeed, denoted as [newline], to the end of the string to be printed. This explains the newline after the second HO above. This automatic newline may be suppressed by ending the PRINT command with a dangling comma like

```
>P $1$1,
HOHO>
```

STRING EXPRESSIONS

A string expression is anything that can be evaluated to yield a string. String expressions are built up from one or more of the fields listed below. The value of the expression is formed by concatenating the values of the constituent fields.

FIELD	CORRESPONDING STRING
$[ne]	The string in dollar-line [ne]
'[string]'	[string] which may not contain '
"[string]"	[string] which may not contain "
[ne]	The value of [ne] converted to a string under control of the default output format and radix specifications.
[ne]@W[ne1]@A[ne2]@R[ne3]	The value of [ne] converted to a string using the specified format and radix (see below).
;	Carriage-return and linefeed
,	Has no value. May be used as a separator to resolve ambiguity.
%C[ne]	The ASCII character formed by taking the value of [ne] modulo 128.
%R[ne]	The 3-character string formed by unpacking [ne] from the standard DOS radix50 format.
%S[ne1] $[ne2]	The [ne1]'th substring of line $[ne2]. See section 8 - symbol table facility.
%X[ne]	Value string depends on [ne] as follows: 0: the name and version of the interpreter 1: the character used to mark off thousands in numbers 100+V: name of V'th variable (see page 36)

The PRINT command is of the form

PRINT [string expression]

and its effect is to print the value of the string expression followed by a newline. The above syntax thus defines what may occur in a PRINT statement. However for the beginner it is sometimes helpful to look at it the other way round and to say that anything that can be PRINTed is a valid string expression. In particular, the PRINT command provides an easy way of verifying that a string expression does in fact evaluate to the intended string.

We shall use [se] to denote any string expression in future.

Examples

Suppose that we have two simple variables, X and Y, where X has the value 123 and Y the value 5. Suppose also that we have two dollar-lines as follows:-

```
1 $COPY
2 $CAT
```

Here are some of the ways in which this data could appear as elements of a string expression:-

STRING EXPRESSION	VALUE OF EXPRESSION
$1	COPY
$1$2	COPYCAT
$1 $2	COPYCAT
$1' '$2	COPY CAT
$1 ' ' $2	COPY CAT
'QUOTE IT'	QUOTE IT
"QUOTE 'IT'"	QUOTE 'IT'
X	123
X+Y	128
X@W5	123
'THE '$2' IS'X-102@W3' YEARS OLD'	THE CAT IS 21 YEARS OLD
%C65%C54	A6
X Y	123 5
X,Y	123 5
X,,,Y	123 5
X;Y	123 5

```
>P 123+4
127
>P 8 9 10
   8   9  10       [default format is 4 characters wide
>LET J=5 :PRINT 'NO.3-'J@W
NO.3-5
>LIST
  1 $COPY
  2 $CAT
  3 $THE TEMPERATURE
  4 $ IS ABOUT
  5 $BELOW ZERO.
>P $1$2
COPYCAT
>LET T=13
>P $3$4 T@W3 ' DEGREES' ;$5
THE TEMPERATURE IS ABOUT 13 DEGREES
BELOW ZERO.
>CLEAR
>L J=0
>P %C65+J, :L J=J+1 :UNLESS J>26:
ABCDEFGHIJKLMNOPQRSTUVWXYZ>P J
 27
>
```

A [se] may evaluate to the null string, or it may itself be null. For example all the following evaluate to nothing:-

```
''
'',"",,,
[nothing at all]
```

CONVERSION OF VALUES TO NUMERIC STRINGS

When numerical values are converted to strings of digits we need to be able to specify the way in which this is done: how many digits are required, whether we want leading zeroes or spaces, and so on. This is done using the syntax

[ne1][format specifier]

where [ne1] is the value to be converted, and [format specifier] is zero or more elements of the form

@[code][ne]

where @[code] identifies a particular function as follows:

@W[ne]	Overall width of the numeric field
@A[ne]	Number of digits after the radix point
@R[ne]	Radix for conversion
@F[ne]	Format status bits specifying type of sign indication and so on.

Note: the alternative syntax @[ne]#[ne] is obsolete and should not be used.

The format status bits are as follows:

BITS	VALUE	MEANING
0,1	0	Mark positive values with 1 leading space
	1	No special mark for positive values
	2	Mark positive values with a + sign
	3	Reserved
2	4	Fill field with leading zeroes, else spaces
3	8	Enclose negative values in round brackets, else mark with - sign
4	16	Insert commas every 3 digits to left of radix point
5	32	Put sign mark at extreme left of field, else put it just to left of leftmost digit
6,7		Reserved.

If part or all of the format specifier is omitted, default specifications are used as follows:

@W	Default value is taken from the system variable QW
@A	Default value is taken from the system variable QF
@R	Default value is ten
@F	Default value is zero.

These defaults mean that numbers are normally printed in decimal with leading zeroes suppressed, negative numbers marked with a - sign, QF digits after the decimal point, and occupying a field

which is QW characters wide. The system variables QW and QF also have default values which are set by the RUN command. These are QW=4 and QF=0, giving a 4-character field with no decimal point:

```
>P 1; 12; 123; -1; -12; -123
   1
  12
 123
  -1
 -12
-123
```

In the above examples the width is 4 by default and so each number is made to occupy a field of exactly 4 character positions. The digits of the number are right-justified within this field by the insertion of between 0 and 2 leading spaces. If the number is negative it is marked with a - sign. Positive numbers are marked with an additional leading space, so that numbers with different signs still take the same width. Thus a field of width W is really intended for numbers upto W-1 digits long, the remaining character position being reserved for the sign indication.

If you misjudge the format specification and try and print a number that has more than W-1 digits, the value is printed in full even though it overruns the specified field width:

```
>P 123; 1234; 12345; -123; -1234; -12345
 123         [3-digit number in 4-character field, ok
 1234        [4-digit number overruns field, but initial sign
 12345       [mark still there. Similarly for 5 or more digits.
-123         [3-digit negative number, ok
-1234        [4 or more digits overruns.
-12345
```

The following example shows the use of the @F status bits to control position and type of sign indication:

```
>LET QW=6          [Set default width to 6
>P 123
   123             [Positive number marked with leading space
>P 123@F2
  +123             [Positive number marked with + sign
>P 123@F2!32
+  123             [Same, but with sign at extreme left of field
>P -123
  -123             [Negative number marked with - sign
>P -123@F32
-  123             [Same, but with sign at extreme left of field
```

If the number is positive, the use of @F1 to suppress the sign mark makes all W character positions available for digits:

```
>P 12345@F1; 123456@F1
 12345             [5-digit number always fits in 6-char field,
123456             [but with sign suppressed so does 6-digit one.
```

@F4 causes the field to be filled out with leading zeroes rather than spaces:

```
>P 123@F4          [Positive number with leading zeroes
 00123             [Note that sign mark still appears
>P -123@F4         [So negative numbers in same format
-00123             [still line up.
>P 123@F4!1        [But we can suppress the sign mark if desired,
000123             [giving an extra leading zero.
>P -123@F4!1       [But for a negative number in same format
-00123             [the sign cannot be suppressed.
```

@F16 puts a comma every 3 digits to mark the thousands:

```
>LET QW=14         [Set wider width for these examples
>P 12345@F16; 123456789@F16
        12,345     [The numbers are still right-justified
   123,456,789     [within a W-character field.
```

@A inserts a radix point a specified number of digits from the righthand end:

```
>P 12345@A2        [QW is still 14 in these examples
        123.45     [The number is still right-justified
>P 123456789@A2@F16
  1,234,567.89     [We can have commas and radix point
>P -123456789@A2@F16!4
-01,234,567.89     [And signs and leading zeroes as required.
```

In commercial applications it is customary to denote a deficit by enclosing the quantity in round brackets. This can be done with @F8 which marks negative numbers in this way:

```
>P -12345@F8
       (12345)     [The overall field is still W characters
>P 12345@F8        [Positive numbers are shifted left by one
        12345      [so that the figures still line up.
>P -12345@F8!32
(       12345)     [We can left-justify the opening bracket
>P -12345678@A2@F16!8
  (123,456.78)     [Or have commas and radix point as required.
```

In systems programming it is frequently useful to be able to print numbers in octal or binary. This is done with @R:

```
>LET QW=4          [Back to normal width
>P 127@R8
 177               [Converting to octal
>P 5@R2
 101               [Or binary
```

We can use @W to alter the field width locally, rather than changing QW:

```
>P 5@W6@R2
   101             [Five in binary, width 6
>P 5@W6@R2@F4
 00101             [Same with leading zeroes
>P 123456789@W32@F4@R2
 00001110101101111001101000101 01
```

In the above examples we have used things like @F4!1 for clarity. In practice it would be quicker to combine the individual bits of @F into one value like @F5.

In most European countries the role of the comma and point in numbers is interchanged. That is to say, a comma is used to separate the integral and fractional parts of a number, and points are used to mark off the thousands. AIMS can be configured to suit this convention, in which case all the above examples should be read as if the comma and point were interchanged.

A program can find out which convention is being used by means of %X1 which evaluates to the single character, either comma or point, that is used to mark off the thousands.

Programmers writing software that is intended to work with both conventions should bear in mind that in Europe a comma cannot be used to delimit a number. For example, the string 123,456 would be converted as a single six-digit number in Europe, and as two three-digit numbers elsewhere. Department/user numbers may appear as [16,17] or [16.17] or [16 17]; use %F9 to skip any of these easily. People designing command syntaxes and writing user manuals should also be aware of these points.

PAGINATION, TABULATION & GRAPH PLOTTING

AIMS keeps a continuous record of the position of the teletype-carriage as it moves in response to PRINT commands. This information may be accessed via the system variables QC and QL (see section 11) as follows:-

QC position of carriage across the page (ie. column number)
QL position of carriage down the page (ie. line number)

For example

```
>P QL
 17
>P QL
 19
>P 'ABC', :P QC
ABC  3
>
```

QL is incremented by AIMS whenever a linefeed is printed. QC is incremented for all characters except carriage-return and linefeed, and is zeroed when carriage-return is printed. Because the output is buffered it is not always possible to determine the instantaneous position of the output device. QC and QL are updated at the end of every PRINT command and thus reflect the position that is reached after the command has been obeyed. QL is also sensitive to the linefeeds that result from INPUT commands, allowing overall pagination of an interactive conversation.

The two variables may be assigned by the user, so for example we may reset the line counter by saying

```
LET QL=0
```

QL is useful for paginating listings:-

```
 10 $Shopping List
 50 LET P=0
100 LET P=P+1 QL=0 :PRINT '--';;$10, :TAB 40 :PRINT 'Page 'P@W;;
120 ...etc
>RUN
--

Shopping List                          Page 1

>
```

Later on in the program we can test whether a new page is needed by

```
IF QL>71 :DO 100
```

or we can throw to a new page with

```
IF QL<71 :PRINT %C10, :LOOP
DO 100
```

A TAB command is provided to facilitate tabulation and graph plotting:-

```
TAB [ne] [optional se]
```

TAB prints spaces until column [ne] is reached, counting from zero at the left margin. Eg:

```
>PR 'HI', :TAB 9 :PR 'HO'
HI       HO
>
```

If QC is already greater than [ne] the command has no effect. If the [se] is present, the value of [se] will be printed repeatedly instead of the space:-

```
>TAB 11 'ABC'
ABCABCABCAB>          [The last occurrence is truncated if necessary
```

Although we have described the behaviour of QC and QL with reference to a teletype, these variables are really associated with the PRINT and TAB commands. They are thus affected by any PRINT or TAB command, even if the output is directed to some other device such as a disk file (see section 13). If several I/O channels are referenced concurently by a PRINT, TAB or INPUT command, the values of QC and QL will cease to be meaningful. In this case use the LET command to save and restore QC and QL, keeping a separate copy for each channel.

PACK and UNPACK commands

These commands are used for moving large numbers of characters between strings and arrays. The PACK command takes a block of dollar-lines and packs them into an array. A specified string of separator characters is inserted into the array to mark the end of each dollar-line. This enables the dollar-lines to be recovered exactly by a subsequent UNPACK. The UNPACK command scans an array for one or more occurrences of a specified separator string, and stores any intervening characters in successive dollar-lines.

PACK A(J) [se],[ne1] [ne2]

A(J) is the array, [se] is the desired separator string, and [ne1] through [ne2] are the dollar-lines to be packed. There is no need for the dollar-lines to be contiguous; the command simply packs all lines between [ne1] and [ne2] inclusive. The actual number of bytes packed is returned in QA. This will be the sum of the lengths of all the strings, plus the number of strings packed times the length of the separator string. If the packed data does not completely fill the array, a final byte containing 128 is stored beyond the last separator as an end marker. This byte is not counted in the QA value. It prevents the UNPACK command from treating the remainder of the array as a valid string.

UNPACK A(J) [se],[ne1] [ne2]

Unpacks the array into a block of contiguous dollar-lines beginning at [ne1]. A new dollar-line is started each time the separator string [se] is found in the array. The separator string is discarded and does not appear in any of the dollar-lines. [ne2] is optional. If present it specifies a limiting line for the unpack, so that if the array contains an unexpectedly large number of separator strings the unpack will not overwrite lines above [ne2]. The number of the highest dollar-line created by the unpack is returned in QA. If the unpack is not terminated prematurey by [ne2], it will continue until a 128 byte is found or the end of the array is reached.

The UNPACK command can create a large number of dollar-lines in one go. This may require a lot of space and it is the programmer's responsibility to see that QS is adequate before executing an UNPACK (see section 20). If your program repeatedly UNPACKs into the same block of dollar-lines, it is desirable to CLEAR [ne1],[ne2] before each UNPACK.

The LET command and Strings

An extension of the LET command provides a convenient way of packing strings and numbers into an array in a sequential manner:-

LET B(J)<X#2 $10#9 317 Z#3

This command places the 2-word value of X in cells J and J+1 of the array B, the 9-character string $10 in cells J+2 through J+6, the number 317 in cell J+7, and the 3-word value of Z in cells J+8

through J+10.

Similarly, the command

```
LET B(J)>X#2 $10#9 Y Z#3
```

unpacks cells J through J+10 of the array B into the 2-word variable X, the 9-character string $10, the 1-word variable Y, and the 3-word variable Z.

The list following the < or > may be any sequence of elements of the form

[name]#[precision]

or

[dollar line]#[number of characters]

For the packing command the [name] may be replaced by an expression if desired, like

[ne]#[precision]

[precision] is an [ne] specifying the number of words to be occupied by the numerical value. Error ?0 will occur if the value cannot be represented within this length.

[number of characters] is an [ne] specifying the number of characters to be packed or unpacked. Strings are packed 2-characters per word and an extra character is required at the end to terminate the packed string. Thus the number of array cells required by the element

[dollar line]#[ne]

is ([ne]+2)/2 . When packing a dollar-line into an array, the line need not contain exactly [ne] characters. If the string contains less, the array will be padded-out with terminator characters. If the string contains more, it will be truncated after [ne] characters. The terminator character has the value 128.

If the [precision] specification is omitted it will be taken as 1. The [number of characters] specification must always be present.

Obtaining the names of your variables

For diagnostic purposes it is sometimes useful to obtain a complete list of all the simple variable names that exist within a program. This may be done with the %X element in a string expression. %X100+V evaluates to the name of the V'th simple variable in the program. If V is too large the element evaluates to the null string. For example, the following program prints a list of all the variables currently in existance:

```
900 LET V=0
902 PUT %X100+V>$1 :UNLESS $1='' :PRINT $1 :LET V=V+1 :LOOP
```

7. STRING COMPARISONS

The relational operators listed in section 5 may be used to compare strings for equality or alphabetical ordering, or to test if one string contains another.

These comparisons are of the form

[string1] [operator] [string2]

where [string1] is a dollar-line, and [string2] may be either a dollar-line or a quoted string.

EXAMPLE	MEANING
[s1]_[s2]	true if [s1] contains [s2] anywhere
[s1]^[s2]	true if [s1] begins with [s2]
[s1]<[s2]	true if [s1] alphabetically less than [s2]
[s1]=[s2]	true if [s1] the same as [s2]
[s1]>[s2]	true if [s1] alphabetically greater than [s2]
[s1]<>[s2]	true if [s1] not the same as [s2]

The alphabetical ordering is such that 'A' is less than 'B', 'ABC' is less than 'ABD', and so on. The ordering of punctuation characters can be found from the table in section 26.

The <, >, and = operators may be combined to specify less-than-or-equal, etc.

The _ and ^ relations are also true if the two strings are equal.

These expressions evaluate to -1 if the relation is true, and to 0 otherwise.

String comparisons are most often used in the IF command, but any of the above relations may occur as an element of a numerical expression. For example:

```
>1$ABC
>IF $1_'BC' :PRINT 'GOOD'
GOOD
>50$NO
>51$YES
>PRINT $1_'B'
- 1
>PRINT $1_'X'!5<6
- 1
>P $50-($1^'AB')
YES
>
```

The IF command simply evaluates the expression and continues along the line if the low-order 16 bits of the value are non-zero.

When performing an embedded match like

```
IF $1_'ABC'
```

the system variable QI (see section 11) is set to the number of times the first character of 'ABC' is found in $1. This is useful when searching symbol tables. See section 8 for an example.

8. THE 'PUT' COMMAND

In its simplest form the PUT command is a left-to-right string assignment of the form

PUT [string expression] > [destination string]

where the > is the assignment operator.

```
>PUT 'HELLO'>$1 :LIST
  1 $HELLO
>
```

A slightly more complex example is:

```
PUT 'ABC123DEF' >$1 X >$2
```

This may be read as 'PUT the string ABC123DEF into dollar-line-1, look for a number and assign it to the simple variable X, and put the rest of the string into dollar-line-2' .

```
>LIST :PRINT X
  1 $ABC
  2 $DEF
  123
>
```

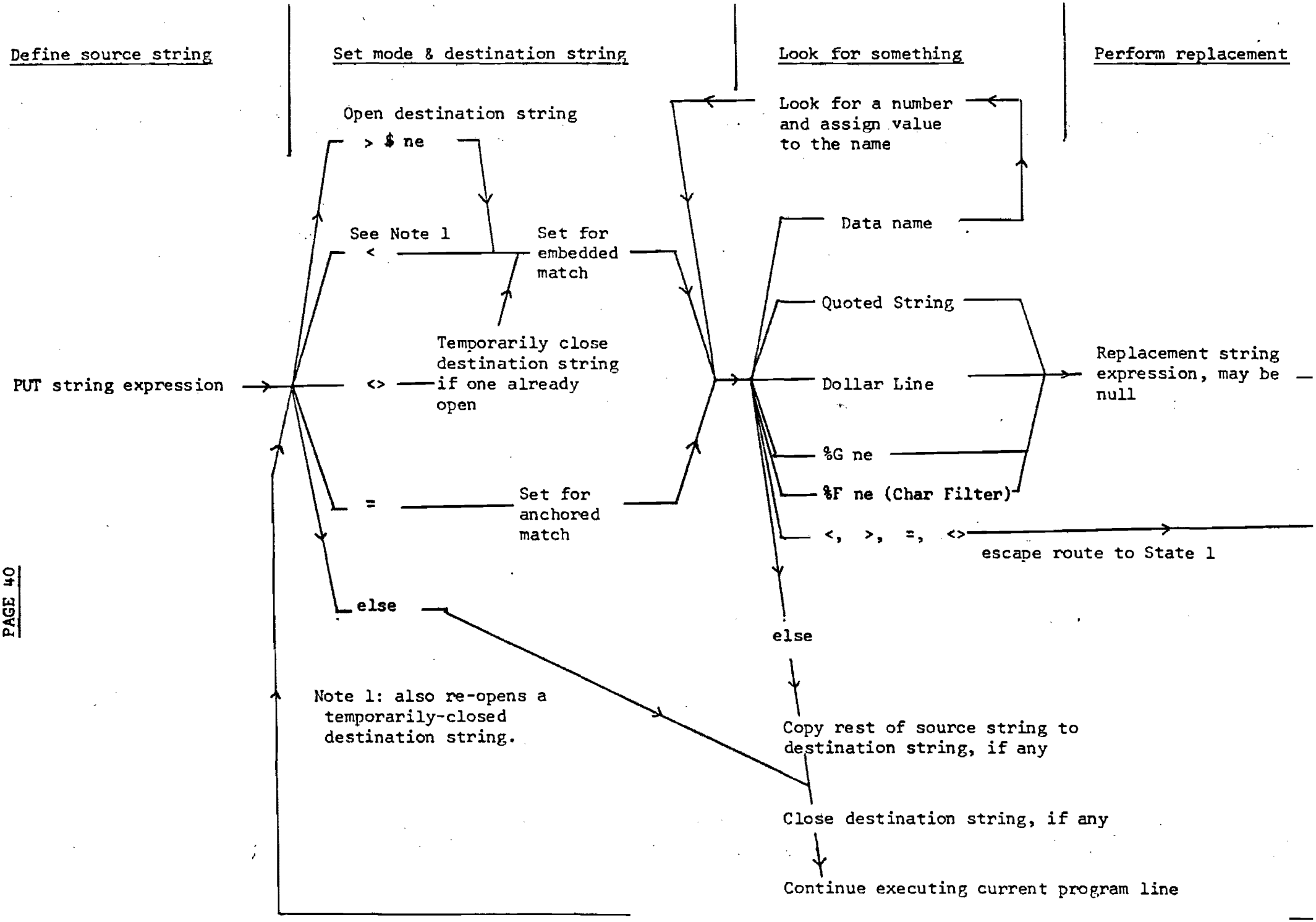
Another possibility is

```
>PUT 'ABCDEF' >$1 'CD' 'ZAM' >$2 :LIST
  1 $ABZAM
  2 $EF
>
```

This may be read as 'PUT ABCDEF into dollar-line-1, look for CD, replace it by ZAM, and PUT the rest into dollar-line-2' .

The PUT processor is probably best understood in terms of a state diagram. Every PUT command has an easily recognised lefthand side, which is the [string expression] shown above. The PUT processor begins by evaluating this string expression and creating a copy of the resulting string. We shall refer to this string as the 'source string'. The rest of the PUT command-line specifies how the source string is to be decomposed, converted, or copied into other strings, variables, or arrays.

The diagram on the next page gives an 'exploded' view of the possible PUT commands which may help to clarify the description below.


Define source string
Set mode & destination string
Look for something
Perform replacement
PUT string expression
Open destination string
> $ ne
See Note 1
<
Set for embedded match
Temporarily close destination string if one already open
<>
=
Set for anchored match
else
Look for a number and assign value to the name
Data name
Quoted String
Dollar Line
%G ne
%F ne (Char Filter)
<, >, =, <>
escape route to State 1
Replacement string expression, may be null
else
Copy rest of source string to destination string, if any
Close destination string, if any
Continue executing current program line
Note 1: also re-opens a temporarily-closed destination string.

The righthand-side of the PUT command-line may be indefinitely long. We may regard the PUT processor as going round a cycle of three main states until the end of the command.

```
PUT [se] [search mode][destination]  [look for]  [replace by]
        ^-------state 1----------^  ^-state 2-^ ^--state 3--^
       ^--------------may be repeated indefinitely-----------^
```

STATE 1: DEFINE SEARCH MODE

In this state we expect one of 4 operators

>$[ne]	If line [ne] is a dollar-line, delete it. Open $[ne] as the current destination string, and set embedded search mode.
=	Set anchored search mode.
<	Set embedded search mode. No destination.
<>	Set embedded search mode. Temporarily close the current destination string, if any. The old destination string may be re-opened later by using the < operator in state 1.

Goto state 2.

If none of the above alternatives are found, goto state 5.

STATE 2: LOOK FOR SOMETHING

In this state we expect to be told what to look for in the source string. One of the following may occur:-

$[ne]	Look for an occurrence of the string in $[ne]
'[string]'	Look for [string]
"[string]"	Look for [string]
[name]	Look for a number and assign it to [name] and goto state 2. Number will be scaled according to decimal point if one present. A number is defined as zero or more spaces, an optional minus sign, followed by at least one digit or a decimal point.
[name]#[ne]	As above but expect [ne] digits after the point. Surplus digits discarded. Number scaled by 10^[ne] whether point present or not.
%G[ne]	Copy [ne] characters
%F[ne]	Look for a class of characters (see page 44)

The interpreter now searches the source string for the specified data. If the search mode is anchored, the search will fail if the data is not immediately found at the current point in the source string. In this case goto state 4. If the search is embedded, characters are copied from the source string to the destination string until the search is satisfied, or the source string is exhausted. If exhausted goto state 4. If found, skip over the found data in the source string and goto state 3.

If none of the above alternatives is found, copy the rest of the source string to the destination string and goto state 5.

STATE 3: PERFORM REPLACEMENT

In this state we expect to find a string to be appended to the destination string in place of the data that we have just found in state 2. Two forms are allowed:

[string expression]	The expression is evaluated and appended to the destination string, if any.
[null]	No action

Goto state 1.

STATE 4: SEARCH HAS FAILED

If this was an embedded search, delete any characters copied to the destination string whilst performing the last search. Close the destination string if it exists. Proceed to the next line of the program.

STATE 5: COMMAND SUCCESSFULLY COMPLETED

Close the current destination string if it exists. Continue executing the current program line.

Note that both states 1 and 2 exit to state 5 if no valid alternative is found. There is an important difference however: the exit from state 2 copies the remainder of the source string to the destination string, whereas the exit from state 1 does not. Thus

```
PUT $1>$1 'FRED' 'SAM' :GOTO ...
```

will replace the first occurrence of FRED by SAM in $1, and will delete the rest of $1. But

```
PUT $1>$1 'FRED' 'SAM' < :GOTO ...
```

will do the replacement and retain the whole of $1.

Examples

```
>PUT 'ABCDE'='AB'>$1 :P $1  [Anchored search for AB
CDE                        [No replacement, put rest into $1.
>PUT 'ABCDE'='BC'>$1 :P $1  [Anchored search for BC
>                          [Fails since string does not begin
                           [with B, $1 unchanged.
>PUT 'ABCDE'<'BC'>$1 :P $1  [Embedded search for BC
DE                         [No replacement, put rest into $1.
>PUT 'ABCDE'>$1'BC'>$2 :P $1;$2          [Copy into $1
A                          [Embedded search for BC
DE                         [No replacement, put rest into $2.
>PUT 'ABCDE'>$1'XY'>$2 :P $1;$2          [Copy into $1
>P $1                     [Embedded search for XY fails
ABCDE                      [Everything goes into $1, $2 unchanged.
>PUT 'ABCDE'>$1'BC''XY'>$2 :P $1;$2      [Copy into $1
AXY                        [Embedded search for BC, replace with
DE                         [XY, put rest into $2.
>PUT '123'=X :P X          [Anchored search for a number.
 123                       [Value is assigned to variable X.
>PUT '  45'=X :P X         [Leading spaces are skipped when
 45                        [searching for a number.
>PUT '  -45'=X :P X        [A minus sign is accepted
 -45                       [as a negative number.
>PUT '- 56'=X :P X         [But only directly before a digit.
>P X                       [It failed
 -45                       [so X is unchanged.
>PUT 'AB(CD)EF'>$1'('<>')'< :P $1        [Copy into $1
ABEF                       [Look for (, turn output off, look
                           [for ), turn output on again.
>PUT 'ABC123DEF'<X='D'>$1 :P X;$1        [Embedded search for
 123                       [a number and assign it to X,
EF                         [insist number is followed by D
                           [and put the rest into $1.
>PUT 'ABCDEF'>$1%G3 :P $1  [Copy into $1
ABC                        [and pass the next 3 characters only.
>PUT 'XYZABCD'>$1%G3>$2 :P $2;$1         [Copy into $1
XYZ                        [take the next 3 characters only
ABCD                       [and put the rest into $2.
>PUT 'AB/CD/EFG'>$1'B'>$2%G1>$3$2>$4 :P $1;$2;$3;$4
A                          [Copy into $1 until you find a B
/                          [Put the next character into $2
CD                         [Then start copying into $3 until
EFG                        [you find $2, and put the rest into $4.
```

The next example shows the use of the PUT command to implement a simple interpreter for another language.

```
>LIST
100 PRINT 'COMMAND', :INPUT $1
110 PUT $1 < X 'PLUS' < Y :PRINT X+Y :GOTO 100
120 PUT $1 < X 'MINUS' < Y :PRINT X-Y :GOTO 100
130 PUT $1 < X 'TIMES' < Y :PRINT X*Y :GOTO 100
140 PRINT "I DON'T UNDERSTAND" :GOTO 100
>RUN
COMMAND *WHAT IS 4 PLUS 6?
 10
COMMAND *TYPE 7 MINUS 4
  3
COMMAND *123TIMES2
246
COMMAND *WHAT IS 8 OVER 2
I DON'T UNDERSTAND
COMMAND *HOW MANY TIMES MUST I ASK YOU?
I DON'T UNDERSTAND
COMMAND *
```

%F CHARACTER FILTER

When using the PUT command to analyze data strings, we often need to search the source string for a class of characters, rather than for a specific string. This is done by using a character filter like

%F[ne]

in state 2 of the PUT command. The [ne] specifies a class of characters as follows:

[ne]	CLASS OF CHARACTER
1	Separators (ASCII codes between 0 and 40 octal)
2	All digits (0,1,2,3,4,5,6,7,8 and 9)
4	All letters (A to Z, and a to z)
8	The remaining characters (IE. those not listed above)
16	Convert letters to lower case (else to upper case)

These values may be combined to specify classes which are the union of those given above. For example, %F6 specifies the class consisting of all letters and digits.

The action of the filter is to move characters from the source string to the destination string (if any), provided the characters belong to the class specified by [ne]. As soon as a character is met that does not belong to the class specified, the moving action stops and the next part of the PUT command is executed. The source string pointer is left pointing at the character that stopped the filtering action, and this character is thus the first one to be processed by any subsequent searching or copying operation. Note that, unlike most state 2 operators, %F cannot cause the PUT command to fail. Execution always continues with the rest of the command line, even if there are no characters that match the filter class.

```
>PUT $1=%F1>$1          [Removes leading spaces etc from $1
>PUT 'ABC12(?*+DEF'>$1%F4>$2 :P $1;$2
ABC                     [Copies letters into $1
(?*+DEF                 [and puts rest into $2.
>PUT '12-NOV-78'=D<%F8>$1%F4<>%F8=Y :P D Y;$1
 12 78                  [Looks for a number and assigns it to D
NOV                     [skips punctuation, copies letters to $1,
>                       [skips punctuation, and assigns number to Y.
>PUT 'LABEL2: OPR'>$1%F6>$2%F8<>%F1>$3 :P $1;$2;$3
LABEL2                  [Copies letters or digits into $1, then
:                       [copies punctuation into $2, skips spaces
OPR                     [and puts the rest into $3.
```

Note that all letters copied by %F4 are converted to upper case. This is useful when writing programs to accept user commands in both upper and lower case. This convention can be reversed by %F16 which causes all letters to be converted to lower case.

QI - SYMBOL TABLE LOOKUP

When performing an embedded match, like for example

```
IF $1_$2
```

AIMS will scan along the string $1 looking for a match with the first character of the string $2. As a side-effect of this comparison, the system variable QI is set to the number of such matches that occur upto the point at which the whole of $2 is found (if it is).

```
>LI
 10 $,BEAR,FOX,ELEPHANT,COW,HORSE,
100 INPUT $1 :PUT ',$1>$2
110 IF $10_$2 :PRINT QI :GOTO 100
120 PRINT 'ANIMAL NOT KNOWN' :GOTO 100
>RUN
*FOX
  2
*COW
  4
*SEAL
ANIMAL NOT KNOWN
*ELEPH
  3
*
```

In the above example the string $10 forms a table of animal names, each one beginning with a comma. Line 100 reads a name from the keyboard and places it in $2 with a comma in front of it. The comparison in line 110 then performs an automatic symbol-table lookup and QI gives the position of the animal in the table.

This technique is often convenient for decoding command strings, in which case the action taken at line 110 would probably be something like

```
GOTO 200+50*QI
```

providing an immediate switch to a specific routine for each command (section 19).

We may think of the first character in $10 as a delimiter which divides the string into a number of substrings. After an embedded search, QI gives the number of the substring that matched. Notice that partial matching is possible, as in the 'ELEPH' example given above. We can prevent this if necessary by putting a comma at both ends of the test string: the string ',ELEPH,' will not be found, whereas ',ELEPHANT,' will.

%S - SELECTION OPERATOR

As noted in section 6, the element %S[ne1] $[ne2] may occur in a string expression. This operator is complementary to the QI facility described above. The element evaluates to the [ne1]'th substring of $[ne2], without the enclosing delimiter characters. For example, using $10 as given above:-

```
>PRINT %S2 $10
FOX
>P %S5 $10
HORSE
>
```

If [ne1] is greater than the number of delimiter characters in the string, the %S construction evaluates to the null string.

9. THE 'INPUT' COMMAND

The INPUT command is a way of reading strings or numbers from the user's terminal or other device. The syntax is

INPUT [echo] [timeout] #[ne1] ?[ne2] [as r.h.s. of the put command]

Note: most of the items are optional but if present they must occur in the order given above.

INPUT is exactly like PUT except that the source string is obtained from the device on channel [ne1], rather than from a string expression. When the string has been read AIMS enters the PUT processor at state 1 just as if it had found a > sign. The command

INPUT [echo] [timeout] #[ne1] ?[ne2] [etc]

is thus equivalent to

PUT [string from channel [ne1]] > [etc]

After an INPUT command control normally resumes along the same line. Control will fall through to the next line if a device error or end-of-file condition occurs or if the implied PUT command fails. The various failure conditions are distinguished by the value of QE, see section 16.

If #[ne1] is omitted, channel 2 is used by default.

The ?[ne2] is optional. If present the string in dollar-line [ne2] is printed as a cue to the user. If absent, * is printed. This only occurs if the input channel is a terminal. For example

```
 10 $HELLO MATE:
100 INPUT ?10 $10 :LOOP
>RUN
HELLO MATE:GOOD AFTERNOON          [User types 'GOOD AFTERNOON'
GOOD AFTERNOON                     [which changes cue line
```

CONTROL OF ECHOING

Characters typed at the keyboard are normally printed back immediately so that you can see what you have typed. This automatic echoing may be suppressed by means of the < operator placed directly after the INPUT command word:-

```
>INPUT < $1 :PRINT ;'YOU SAID '$1
*                        [User types 'HELLO' which does not echo
YOU SAID HELLO
>
```

Note that when echo is suppressed there is no automatic carriage-return or linefeed at the end of an input line. The newline after the * above is caused by the PRINT ; command.

This facility allows a program to obtain confidential information from a user without leaving a printed record. For example, The login program turns off the echoing when it asks for your password.

The carriage-return and linefeed that occurs when the user terminates his input line may be suppressed by means of the = operator. This allows several questions to be asked on the same line. For example

```
>20 $YOUR NAME?
>21 $AND AGE?
>INPUT = ?20 $1 :TAB 20 :INPUT ?21 $2
YOUR NAME? FRED    AND AGE? 36
>
```

INPUT Timeout

When an INPUT command is executed the program is normally suspended until a line has been typed by the user. If the user fails to respond or goes away, the program could remain suspended indefinitely. The programmer may avoid this by using the [timeout] feature:-

```
INPUT [echo]>[ne] ...
```

This causes the INPUT command to terminate after [ne] seconds, even if nothing has been typed by the user. Control resumes on the same line as the INPUT command, and the destination string contains whatever the user typed upto the moment of timeout. The system variable QD indicates the type of termination as follows:

QD	Termination reason
0	User typed a line ending with carriage-return.
1	User typed a line ending with ESCAPE, ACCEPT or ALTMODE.
2	Timeout.
3	User typed a line ending with linefeed.

INPUTTING SINGLE CHARACTERS

The INPUT command reads a complete line of text upto a carriage-return or linefeed character. An alternative command, called ACCEPT, is provided to allow a program to obtain characters from the keyboard one at a time. The syntax is identical to that for the INPUT command.

```
>ACCEPT $1 :PRINT $1
*AA                 [User presses key 'A'
>
```

The control-Y and RUBOUT editing facilities are not available in ACCEPT mode; these characters are treated like all others. The control-C and control-O abort characters still function. The carriage-return key appears as carriage-return followed by linefeed.

CONVERSION OF NUMERIC STRINGS TO VALUES

Although AIMS treats all numbers as being integers, we described in section 6 a facility for printing values with a decimal point, as

```
>LET X=12345 :PRINT X@A2
 123.45
>
```

Thus it is possible to perform fixed-point real arithmetic provided all numbers are scaled by the appropriate power of ten (100 in the above example).

When numbers are input, AIMS provides the capability either of accepting a decimal point anywhere, with appropriate scaling, or of specifying the number of digits required after the point. See description of state 2 in section 8. For example

```
>INPUT X Y#2 :PRINT X Y :LOOP
*123 123.45
 123 12345
*123.45 123
 12345 12300
*123456 123.456
 123456 12345
*123.456 123456
123456 12345600
*
```

10. TRANSFER OF CONTROL

IMPLICIT TRANSFERS

Control normally proceeds horizontally along the program line currently being executed. When this line is completed control passes to the line with the next higher number.

Certain commands exhibit conditional properties, like for example the IF command, which only continues along the line if the condition is satisfied. If the condition is not satisfied control will 'fall through' to the next line.

This convention, whereby successful commands proceed horizontally and unsuccessful commands proceed vertically, is extended in AIMS so that many commands have an implied conditional. For example:-

```
100 READ #5 B() :WRITE #6 B() :LOOP     [the # specifies an
110 PRINT 'END OF FILE' :STOP           [I/O channel number
```

This program copies a file by reading it into the array-buffer B, and writing it out again. Control will remain on line 100 until the end of the file is reached, at which point the READ command will fail, causing line 110 to be obeyed.

EXPLICIT TRANSFERS

Apart from these conventions, several commands are provided for explicitly changing the order of program execution.

GOTO [ne] — This causes a jump to line [ne] of the program. Error ?L will result if the line does not exist.

[ne] may be a simple line number, or a more complicated numerical expression, enabling computed and assigned goto's to be performed.

... :LOOP — This command only makes sense when placed at the end of a line. It causes the line to be executed again.

LOOP is quicker than a GOTO because the interpretor already knows where the current line is.

SUBROUTINE TRANSFERS

GOSUB [ne] This is used for calling part of a program as a subroutine. The current line-number is remebered on a system stack, and a 'GOTO [ne]' is executed. The program thus entered should eventually return control by means of the RETURN command. It is meaningless to place commands beyond a GOSUB on the same line.

RETURN The line-number previously stacked by the last GOSUB command is unstacked into the system variable QA. A 'GOTO NL(QA)' is then executed.

(Note that NL(J) is a system function whose value is the next line above line J of the program.) The RETURN command effectively transfers control back to the line following the GOSUB command that entered the subroutine. For example

```
100 LET J=0
110 GOSUB 500
120 PRINT 'HI' :GOSUB 500
130 PRINT 'END' :STOP
500 LET J=J+1 :PRINT 'HO' J :RETURN
>RUN
HO  1
HI
HO  2
END
>
```

As can be seen from the above example, GOSUBs may be nested; that is to say, a piece of program that has been entered via a GOSUB may itself do a GOSUB to some other program, and so on. The maximum depth to which this nesting may be carried is an AIMS assembly parameter. It is normally set at sixteen.

Two variants of the RETURN command are provided to cater for multiple return-points and for situations where one does not want to return at all:-

RETURN [ne] As RETURN except that it performs a 'GOTO NL(QA+[ne])' . This permits a return to a point several lines beyond the line containing the GOSUB.

RETURN : ... If any command follows a RETURN on the same line, the GOTO is not executed. The RETURN in this case simply unstacks the return line-number.

This allows a jump out of a subroutine without causing an accumulation of return links on the gosub stack. It is useful, for example, in cases where a subroutine detects an unusual error condition.

Note that 'RETURN :GOSUB NL(QA)' implements a coroutine switch.

A further command is provided principally to save program space and typing:-

DO [ne] — Causes a temporary transfer of control to line [ne]. The effect is to execute line [ne] as if it occupied the position of the DO command. When line [ne] is completed, control goes to the line following the DO command, unless line [ne] performs a further transfer of control.

DO commands may be chained to any depth. They always operate as if the 'done' line occupied the position of the highest-level DO command.

```
100 PRINT 'HA' :DO 200
110 PRINT 'HO' :STOP
200 PRINT 'HE' :DO 300
300 PRINT 'HI'
>RUN
HA
HE
HI
HO
```

TRANSFERS BETWEEN RUN & EDIT MODES

AIMS is initially in edit mode. This mode allows programs to be entered and direct commands to be executed. When a program is established the RUN command may be used to start execution:-

RUN [ne] — Makes all user-created simple variables undefined. Sets system variables to their default values. Scans the program for ARRAY declarations and performs the necessary assignments. Starts the program at the next line greater than or equal to [ne].

STOP — Stops program execution and switches back to edit mode. The system variable QA is set to the number of the line containing the STOP command. All variables, arrays, lines and files remain as they were at the time of the STOP.

Since all commands may be used in both run and edit modes, we also have:-

GOTO [ne] — As a direct command. Starts the program at line [ne] without performing the initialisation associated with the RUN command. Useful for restarting a program.

RUN [ne] — As a stored command. Used for its initialising effects. (eg. deletes unwanted variable names, sets up array names, resets system variables to a standard state.)

THE 'WAIT' COMMAND

WAIT [ne]	This command suspends the user for [ne] tenths of a second. Control resumes along the line after this interval. Note that the interval is only resolved to the nearest tenth of a second, so that a WAIT N command may actually delay the user for any time between (N-1) and (N+1) tenths of a second. See also section 21.

TRANSFERS TO AIMS EXECUTIVE PROGRAMS

EXIT	Runs the privileged AIMS executive program, see section 23.
BYE	Runs the privileged AIMS program 'LOGOUT' which logs the user off. See section 23.

11. SYSTEM VARIABLES AND FUNCTIONS

A number of simple variables and functions are permanently defined. The functions allow the user to call certain machine-code routines that are conveniently accessed in a functional way. The system variables are just like any other variable in that they may be referenced and assigned by the user, but they are also modified by AIMS, often as a side-effect of a command.

SYSTEM VARIABLES

QA Contains miscellaneous auxiliary information. See individual command descriptions.

QC Column counter. See section 6.

QD Indicates reason for termination of INPUT command. See section 9.

QE When an error occurs the error number is placed in QE. See section 16.

QF Global output format. Default is 0. See section 5.

QG Garbage-collection threshold. Preset to 350. See section 20.

QI Set up as a side-effect of string comparisons. When searching for string2 in string1, QI counts the number of times the first character of string2 has been found in string1. See section 8.

QL Line counter. See section 6.

QQ When an error occurs QQ is examined. If QQ is zero, a standard error message is printed. If QQ is non-zero, a 'GOSUB QQ' is executed and QQ is set to zero. A user program may thus trap errors by setting QQ to the line number of an error-handling routine. Default is zero. See section 16.

QS Amount of free space in characters available within the user's existing memory area. See section 20.

QW Global output width. Default is 4.

All system variables are reset to their default values by the RUN command.

SYSTEM FUNCTIONS

DA() The day of the month (1 to 31)

DA(1) The month (1=Jan, 12=Dec)

DA(2) The year - 1900

DA(3) Day of the week (0=Mon, 6=Sun)

DA(4) Day number within year (1st Jan=1, 31st Dec=365 or 366)

DA(5) 416*(year-1970) + 32*month + day

DA(6) 416*(year-1965) + 32*month + day

DR(n) Accesses DR11C hardware registers if present (n=0 to 16)

EP() Controls the precision with which numerical expressions are evaluated. See section 5.

FC() MONITOR: total available real memory in bytes. Equals total real memory minus the size of all resident monitor program and data structures.
DOS: Total amount of free space in characters available for program expansion. Indicates the amount by which the user's memory area could be expanded by the CORE command.

GV(J) The J'th global communication variable. See section 21.

JS() User's job status vector. See section 22.

LE(J) Value is number of characters in dollar-line J, or dimension of array in line J.

NL(J) Value is the number of the next line above line J, or zero if no higher line.

PK(L) Privileged function enabling examination (both) and modification (DOS only) of real memory locations. MONITOR: argument L is byte-address of word in kernel virtual address space (L must be even). DOS: argument L is absolute word-address of location to be accessed (Ie. PDP-11 byte-address over 2).

PT(J) I/O channel pointers. See section 14.

QX(c,n) I/O channel status information. See section 13.

SS() System status vector. See section 23.

TA(N) Bit tally. Value is the number of bits that are 1 in N.

TI() The time of day in tenths of a second past midnight. Use
LET H=TI()/10 H=H/3600 M=QA/60 S=QA
to get time of day in hours, minutes and seconds. Note that dividing by 36000 does not work because QA is only valid if remainder is less than 32768.

12. THE 'CODE' COMMAND

CODE [string expression]

The string expression is evaluated, and is then treated exactly as if it had been typed in as a command to AIMS.

```
>LIST
 10 LET J=30
 20 CODE J"PRINT 'HO'"
 40 LIST
>RUN
HO
 10 LET J=30
 20 CODE J"PRINT 'HO'"
 30 PRINT 'HO'
 40 LIST
>
```

This command allows running programs to modify themselves. This is useful for coding arrays with computed dimensions, and for implementing programs that compile into AIMS.

It should be understood that the CODE command does treat the string exactly as if it had been typed as a command. So that

```
CODE J                    [Deletes line J of the program

CODE 'LET X=6'            [Assigns 6 to X

10 CODE '10LET X=6'       [Overwrites itself
```

It is not possible to put other commands on the same line beyond the CODE command. Control always goes to the next line when the CODE is completed (unless the coded command involves a direct transfer of control, such as a GOTO).

CODING ARRAYS

Arrays may be dimensioned at run-time by coding the array line. For example

```
100 CODE L 'ARRAY ' $1 D
```

will create an array of dimension D in line L of the program, the array name being taken from the string variable $1. Note that the action of declaring an array, either by a direct command or by coding it, does not define the array name. The array name is a simple variable that is assigned a value equal to the array line number by the RUN command. Therefore, when using CODE to create arrays at run-time, the array name must be assigned explicitly like

```
100 LET AB=L :CODE L 'ARRAY AB ' D
```

Note that the text following the command CODE above is a string expression which will be evaluated as described in section 6. If QF or QW is non-zero the CODE command may fail due to the appearance of commas or decimal points in the values of L and D. It is safer to zero the conversion format explicitly like

```
100 LET AB=L :CODE L@W@A 'ARRAY AB ' D@W@A
```

In order to save space, programmers sometimes use one array for several different purposes and alter its size appropriately using CODE commands. Thus the statement

```
100 CODE '10 AR A 'D@W@A
```

might be executed at a time when there is already an array in line 10 with perhaps a size different from D. This technique is perfectly legitimate, but it should be borne in mind that when the CODE command is executed, space is needed for both the old and new versions of the array A. Thus the attempt to save space may actually cause a temporary need for twice the space. This may be avoided by deleting the old array before coding the new one like this:

```
100 CLEAR A,A
110 CODE '10 AR A 'D@W@A
```

THE 'X' COMMAND

The line editing command

```
X [ne] [etc]
```

is actually equivalent to

```
PUT [listing of line [ne]] > $[temp]  [etc]
CODE $[temp]
[if it was a direct X command, list the changed line]
```

where $[temp] is a temporary string variable invisible to the user.

Although the X command is mainly useful when typing in a program, it can be used within an AIMS program. When used in this latter way, the line is not printed. For example, the following program performs an editing function over a specified range of lines:-

```
  2 PRINT 'CHANGE', :INPUT $1
  3 CODE 6 'X F' $1
  4 PRINT 'RANGE', :INPUT F T
  5 IF F>T :STOP
                          [Line 6 created by line 3
  7 LET F=NL(F) :GOTO 5
```

It is sometimes useful to place a listing of a particular program line into a string variable, where it may be manipulated with the PUT command. Note that an X command like

X [ne] ```1$

will put a listing of line [ne] into $1. After manipulation, a CODE $1 will restore the line.

Warning

Note that since the X command involves a CODE, it creates a completely new instance of the edited line. Consequently if the X command is used to edit an ARRAY line, any data in the cells of the array will be lost. Also, space is temporarily required for two copies of the array, as explained above.

13. INPUT/OUTPUT FACILITIES

Data transfer is performed via logical I/O channels. Upto eight channels may be used by each job at the same time. They are numbered from 1 to 8. Note that use of an I/O channel ties up scarce monitor resources; programmers should minimimise the number of channels that are used concurrently.

CHANNEL/DEVICE ASSOCIATION

Before a channel is used it is necessary to connect the channel to a specific peripheral device. This is done by the command

INIT #[channel] [device name]

which initialises the channel and attaches the specified physical device to it. This command will fail with a code in QE if the device is not available.

The device name should be specified as a string expression evaluating to one of the following:-

DOS	MON	DEVICE
DP u	RPcu	RP03 moving-head disk
	RPEcu	RP04 moving-head disk
	RPFcu	RP05 moving-head disk
	RPGcu	RP06 moving-head disk
DK u	RKcu	RK05 moving-head disk
	RKGcu	RK06 moving-head disk
	RSDcu	RS03 fixed-head disk
	RSEcu	RS04 fixed-head disk
	RXcu	RX11 floppy disk
DF u	RFcu	RF11 fixed-head disk
PR u	PRcu	paper tape reader
PP u	PPcu	paper tape punch
	TAcu	TA11 cassette tape
DT u	TCcu	TC11/TU56 DECtape
LP u	LPcu	line printer
TM u	MTcu	TMA11/TU10 magnetic tape
	TUcu	TU16 magnetic tape
	TRHcu	TR07-F magnetic tape
KB u	KBn	terminal input
PT u	PTn	terminal output
PC u	PCn	pseudo-console (for slaved jobs, see section 24)
SD	SD	system disk (for executive use only)
	UD	standard user disk (application program default)

MON: c is a letter distinguishing different controllers of the same type, c=A for first controller, c=B for second, and so on. u is a digit from 0 to 7 specifying the unit number. DOS: u is the device unit number and at least one space is required between the device name and the unit number.

The presence of a device in the above table does not necessarily mean that the device is currently supported by DOS or MONITOR.

When all I/O has been completed, the association between channel and device may be terminated by means of the command

RELEASE #[channel]

The RELEASE command is necessary to free buffer space. In the case of non-shareable devices like paper tape readers, the command also frees the device for use by someone else. It is essential to RELEASE channels as soon as they are no longer needed.

Once the channel has been initialised, any I/O command can be made to reference that device by specifying the appropriate channel number in the command. For example, if we said

INIT #5 'PP'

which attaches channel 5 to the paper tape punch, then the command

PRINT #5 'YOUR NAME IS ' $2

would punch that string. This example is oversimplified because for file-oriented devices like disks and DECtape we also need to specify the file to which I/O is directed. This is done by means of the 'OPEN' command.

CHANNEL/FILE ASSOCIATION - OPEN & CLOSE COMMANDS

The OPEN command associates a named file and data mode with a specified channel. All channels must be OPENed before I/O can be performed.

OPEN #[channel] [mode] [filename] [[dept user]]

where

[mode] specifies the type of file access required as:

0 Sequential reading of a text file (DOS openi).
1 Create a text file for sequential writing (DOS openo) An error will occur if the file already exists.
2 Obsolete, do not use.
3 Obsolete, do not use.
4 Open existing contiguous file for random access.
5 Open channel for physical I/O (use with great care).
6 DOS: Similar to mode 4, see Fast Access Directory.

[dept user] specifies the department and user numbers in square brackets. If absent the current user is assumed.

This command searches the device directory for the specified file and connects the file to the channel. The command may fail with a code in QE due to file not found or protection violations.

If the device is not file structured the [filename] and [dept user] specifications are ignored and may be omitted.

If an OPEN command is given to a channel that has not been initialised, the channel is automatically initialised to the default disk for user files, as given by UD (MON) or SS(8) (DOS).

File names must consist of letters or digits. A name is composed of two parts: (1) a name upto 6 characters long, and (2) an extension upto 3 characters long. A period is used to separate the two parts. Both parts are significant, so that all the following are different valid filenames:-

JACK.BAS JACK.DMP DIRECT.BAS DIRECT.DAT JACK TEST34

The file extension is normally used to indicate the general type of the file, and is chosen from a small set of standard mnemonics:-

EXTENSION	TYPE OF FILE
.BAS	Linked text file containing a saved AIMS program
.CTL	Text control file for BATCH or OBEY processing
.DAT	Fixed-length contiguous data file
.DMP	Fixed-length contiguous file containing one or more dumped AIMS program overlays
.V3	As .DMP but for AIMS version 3 executive programs only
.LNK	Linked text file
.LOG	Log file written by BATCH processor
.SYS	Contiguous file for use by executive programs
.TMP	Temporary file

The element [filename] in an AIMS command should be specified as a string expression which evaluates to the desired name.

Examples

```
>LI
  1 $JACK.LNK
  2 $JACK
>OPEN #5 0 $1 :OPEN #6 1 $2'.TMP' :PRINT 'OK
OK                      [JACK.LNK opened & JACK.TMP created
>OPEN #7 1 'JACK.LNK' :PRINT 'OK
>PRINT QE/256           [command failed because
   2                    [file already exists
>
```

Decimal department/user numbers may be specified in square brackets like

```
OPEN #5 0 'JACK.LNK[16 17]'
```

or

```
OPEN #5 0 'JACK.LNK['DN UN
```

if DN and UN are variables with the appropriate values.

When the user has finished reading or writing a file he should close it by means of the command

CLOSE #[channel]

In the case of an output file this command causes the device directory to be updated to include the new file. In the case of an input file the CLOSE may be necessary to permit other users to access the file. The CLOSE command also frees buffer space. The RELEASE command performs a CLOSE automatically if necessary.

DATA TRANSFER COMMANDS

When a channel has been initialised and opened, data may be transferred using any of the following commands:-

INPUT #[channel]	Reads a string
PRINT #[channel]	Writes a string
TAB #[channel]	Tabulation command
READ #[channel]	Random access binary
WRITE #[channel]	transfers

Data may be filed in two forms:-

1) Text files. These are DOS-compatible linked files in even parity ASCII mode. They may be used to store strings, but may only be accessed sequentially.

2) Random-access binary files. These are DOS-compatible contiguous files of fixed length. Data is transferred in a random-access mode between a point on the file and an AIMS array. The LET command may then be used to unpack the data into variables or strings. See section 14.

SIMPLIFIED I/O CONVENTIONS

The above rather complicated scheme provides the user with the full power of the DOS I/O monitor. However, a set of default conventions is provided to cater for the normal requirements of the majority of users.

When a user logs onto the system, channels 1 and 2 are initialised for output and input to the user's terminal. In addition, each I/O command references a default channel if none is specified. Thus we have

CHANNEL	DEVICE	DEFAULT CHANNEL FOR
1	PT	AIMS error message output the LIST, PRINT and TAB commands
2	KB	input of commands to AIMS the INPUT command
3	def	CALL and SAVE program filing commands DELETE, RENAME and ALLOC commands

```
4       def     LOAD and DUMP program filing commands
5       def     OPEN, CLOSE, MTAPE, READ and WRITE commands
```

where 'def' represents the default disk for user filing. Channels 3 to 8 may be initialised explicitly by the user if desired. If not, they will be initialised automatically to the default user filing disk by the OPEN command.

SIMPLE USE OF FILES

For those who are unfamiliar with the DOS monitor, the basic technique for sequentially accessing a text file is illustrated below:-

```
TO READ AN EXISTING FILE          TO WRITE A NEW FILE

define filename:
     OPEN #5 0 'ABC'              OPEN #6 1 'ABC'

transfer data:
     INPUT #5 $1 :...             PRINT #6 $1

close the file:
     CLOSE #5                     CLOSE #6
```

When writing a new file, the CLOSE at the end causes the device directory to be updated. The new file cannot be read until a CLOSE has been done.

A TEXT EDITING EXAMPLE

The following program shows the use of AIMS to perform a simple editing function. It copies a text file and replaces all occurrences of the word 'BASIC' with the word 'AIMS', creating a new file called 'TMPFIL'.

```
 10 $FILENAME*
100 INPUT ?10 $1 :OPEN #5 0 $1 :GOTO 120
110 PRINT 'FILE '$1' NOT FOUND' :GOTO 100
120 OPEN #6 1 'TMPFIL' :GOTO 140
130 PRINT 'CANNOT CREATE OUTPUT FILE' :STOP
140 INPUT #5 $1 :GOTO 160
150 CLOSE #5 :CLOSE #6 :PRINT 'OK' :GOTO 100
160 PUT $1>$1'BASIC''AIMS'< :LOOP
170 PRINT #6 $1 :GOTO 140
>RUN
FILENAME*GUNKO
FILE GUNKO NOT FOUND
FILENAME*MANUAL
OK
FILENAME*
```

Note the use of the < in line 160 to make sure that the whole of $1 is copied.

DIRECTORY MANIPULATION COMMANDS

DELETE #[channel] [filename]

Deletes the specified file from the directory of the device INITed on [channel]. Default channel is 3, default device is the user filing disk. Command will fail with a code in QE if the file does not exist or is protected against deletion. See section 16 for error codes. MON: a file cannot be deleted if it is open on any other channel of this or any other job.

RENAME #[channel] [new filename se],[old filename se]

Renames the file as specified. Defaults as for DELETE. A decimal file protection code may be included in the [new filename] specification if it is desired to alter the file protection:

RENAME 'TEMP.SRC<237>','TEST.SRC'

Gives the new file a protection of 355 octal.

DEFAULT AND SYSTEM DISKS

As mentioned above, if you attempt to OPEN a channel that is not currently initialised, the system initialises it by default. The device thus obtained is the one that is recommended for normal use by all application programs. This depends on the available hardware and will therefore vary between installations. The default device is chosen automatically at system initialisation time and will normally be the largest disk in the configuration.

MON: The special devicename UD is automatically translated by MONITOR into the name of the current default device. Channels can be connected to the default device explicitly if required by INITing 'UD'.

DOS: The default devicename is stored in radix50 in SS(8). Channels may be connected to the default device explicitly by means of the command

INIT #[channel] %RSS(8)' 0'

Obviously, the default device can also be obtained without doing an INIT, by making sure that the channel is RELEASEd before OPENing it.

The name SD translates to the name of the system device. This is the device used by the executive programs and for swapping. SD is not necessarily the same as the default device.

CHANNEL STATUS INFORMATION - QX(C,N)

The system function QX(c,n) gives information about the current state of I/O channel number c:

QX(c,0)	DOS .STAT status (0 if channel not INITed)
QX(c,1)	Device blocksize in words (set by INIT)
QX(c,2)	Length of file in blocks (modes 4 to 6 only)
QX(c,3)	Block number where file begins (modes 4 to 6 only)

MON only:

QX(c,4)	State of slave job if channel connected to a PC: 0: No slave job and line o/p buffer empty 9: Slave job in TI wait and line o/p buffer empty 10: Slave output is available in buffer any other value indicates slave job busy
QX(c,5)	Device status/error information applicable when last operation on this channel finished (see below)
QX(c,6)	Slave job number if channel connected to a PC
QX(c,7)	Number of the PC to which channel c is connected
QX(c,8)	Latest known position of device. (eg: disk cylinder number, magtape record count)

The QX function returns 2-word values and therefore needs EP()>1.

Functions 4, 6 and 7 are only meaningful when the channel is connected to a pseudo-console. The use of these functions is described in section 24.

Channel Status Word (MON only)

The function QX(c,5) delivers a positive value indicating the latest known state of the device connected to channel c. This function may be examined to get further information if a command fails due to a device problem. It is especially useful after a READ or WRITE command that fails due to a device problem. The value is bit-coded as follows:-

Dec	Octal	Name	Meaning
Status			
1	000001	NXU	Nonexistant unit
2	000002	UNS	Unit unsafe (hardware fault)
4	000004	OFL	Unit offline
8	000010	WLK	Unit write-locked
16	000020	MOV	Unit in motion (eg: heads, tape moving)
32	000040	TMK	Magnetic Tape-Mark sensed
Errors			
256	000400	PSU	Position unknown (eg: disk seek incomplete, bad magtape)
512	001000	HDE	Disk: wrong header found after seek
		BOT	Magtape: at Beginning-of-Tape
1024	002000	WLE	Attempted write when write-locked
2048	004000	NXA	Nonexistant device address
		EOT	Magtape: physical End-of-Tape sensed
4096	010000	EOM	End of medium reached
			Magtape: long record read
8192	020000	DAT	Data invalid (checksum, parity, CRC)
16384	040000	MIS	Data missed due to timing constraints (eg: UNIBUS latency problem)
32768	100000	NXM	Nonexistant memory addressed

DEVICE-DEPENDENT OPERATIONS (MON only)

Most I/O activity is done using the INIT, OPEN, READ, WRITE, CLOSE and RELEASE commands which are implemented for all devices in a uniform manner. The programmer need not concern himself with the detailed characteristics of the device which is being used. However there are some devices that have special characteristics or facilities that cannot be subsumed under the standard scheme. The DDOPR command provides program control of these device-dependent features. By its very nature the DDOPR command performs a different set of functions for each type of device, so programs using the command must be aware of the device that is being used.

DDOPR #[channel] [command se]>$[reply ne]

The command performs the operation specified by [command se] on the device connected to [channel]. Reply information may be returned in $[reply ne]. The >$[reply ne] may be omitted if no reply string is wanted. [command se] is generally a single word identifying the operation required. The command fails if this operation is not applicable to the device. Otherwise the command always succeeds and its effect may be found from the reply string and by examination of the channel status word QX(c,5) when the operation is completed.

Successful execution of the DDOPR command does not imply that the operation thus initiated has been successfully completed; it merely indicates that the operation is applicable to the device. Some DDOPR functions suspend job execution until the operation has been completed, whilst others merely initiate device activity and continue job execution without delay. This depends upon the operation and the device type, and reference should be made to the description of the DDOPR functions for each device.

Note that the channel status word QX(c,5) is set by the monitor when control returns to the user after an AIMS command. For those DDOPR functions that cause no delay, the channel status after the command gives the state of the device when the operation was initiated; this will almost certainly differ from the device state when the operation has been completed.

The DDOPR command '?' is applicable to all devices for which DDOPR is implemented and it returns in the reply string a list of the DDOPR command words for the device, separated by commas. For example, we can get a list of the DDOPR commands for the TMA11/TU10 magnetic tape transport as follows:

```
>INIT #5 'TMA0:' :DDOPR #5 '?'>$1 :PRINT $1
SPACE,WMARK,UNLOAD,REWIND,PARITY,DENSITY
>
```

Some command take arguments. For example the recording density of a TMA11 transport may be set to 556 BPI with the command

DDOPR #c 'DENSITY=556'

The '?' facility may also be used to obtain a list of the acceptable command arguments. For example

```
>DDOPR #5 'DENSITY=?'>$1 :PRINT $1
200,556,800,D800
>
```

This is a list of the valid density settings for the particular type of tape transport connected to channel 5.

DDOPR commands may be issued at any time. The monitor will wait for the device controller and/or drive to become free if necessary. Jobs that become suspended on DDOPR functions are shufflable and swappable.

MAGNETIC TAPE - GENERAL INFORMATION

The two ends of a magnetic tape are marked with a short strip of reflective material that is stuck onto the back of the tape. The tape transport detects these markers photoelectrically and sets status bits which can be read by the program. The marker at the front of the tape is called the Beginning-of-Tape or BOT marker. It defines the LOAD POINT, which is the earliest point on the tape where data may be stored. The REWIND operation always returns a tape to its load point. The marker at the far end of the tape is called the End-of-Tape or EOT marker. Information cannot be written beyond this point. The BOT and EOT status is available to the program in the Channel Status word QX(c,5) under MON, and via the MTAPE command under DOS.

The area between the BOT and EOT markers is available for data storage. Depending on the type of transport, there are either 7 or 9 channels across the width of the tape where a bit may be stored. These channels taken together constitute a FRAME capable of storing one parity bit and 6 (for 7 channel) or 8 (for 9 channel) data bits. A number of frames are written contiguously along the tape to form a RECORD.

Two recording techniques are used: NRZI and Phase-Encoding. Both techniques use a vertical parity bit (VRC) associated with each frame as already described. A dual-gap recording head enables the hardware to perform a read-after-write check on each frame as it is written. With the NRZI technique a longitudinal parity bit is calculated for each channel over the whole record, and these bits are then written as an extra frame at the end called the LPC. For 9 channel NRZI tapes a cyclic redundancy check byte (CRC) is computed over the whole record and stored immediately before the LPC. The LPC and CRC are generated and checked automatically by hardware. The Phase-Encoded technique is more reliable and it is used only with 9 channel tapes and at the higher recording densities (1600 BPI upwards). There is no LPC or CRC, but the hardware is capable of detecting a drop-out on any channel. If a frame is read and one of the channels is found to have dropped out, the hardware automatically reconstructs the missing data bit by making use of the VRC parity. If more than one channel drops out an error is signalled.

With 9 channel tape there are 8 data bits in each frame so one PDP-11 byte occupies one frame on the tape. If you write an array A() to tape, A(0)&255 goes into the first frame, A(0)_-8 into the second, A(1)&255 into the third, and so on.

With 7 channel tape there are only 6 data bits per frame and two different formats are available: Industry Compatible format, and Dump format. In Industry Compatible format each 8-bit PDP-11 byte occupies one frame on the tape, and the two most significant bits of each byte are not used (ie. ignored on write, set to zero on read). Thus an array is written as A(0)&63, (A(0)_-8)&63, A(1)&63 and so on. In Dump format each PDP-11 byte occupies two frames on the tape. The low order 4 bits go in the first frame and the high order 4 bits in the second frame. The remaining two data channels in each frame are not used. For practical purposes the 7 channel industry compatible format is only useful for processing

tapes to suit other machines (eg: IBM).

The area of tape between the BOT and EOT markers can contain zero or more RECORDS, each record being separated from the next by an inter-record GAP. Gaps are created automatically when records are written. They exist mainly to allow time for the transport to start and stop inbetween records.

There are two kinds of record that can be written on the tape: (1) Data Records, containing information supplied by the user when the record is written, and (2) TAPE-MARKs. A Tape-Mark is a special short record containing no data. Tape-Marks can only be written by giving a special write-tape-mark command to the hardware. Tape-Marks are useful because they are detected automatically by the hardware causing a status bit to be set. They may be used to mark important points on the tape such as end-of-file (EOF) or the last data record on the tape. The latter point is called the Logical End-of-Tape (LEOT) and is not to be confused with the physical EOT marker. Any space between LEOT and EOT is spare tape that has not yet been used. An LEOT may be overwritten with additional records if desired. Eventually the tape will become full and this will be shown by the presence of the EOT status bit following a write operation. It is possible to write a record that extends upto the physical EOT marker, but records should not be written when at or beyond the EOT marker, otherwise you may run off the end of the tape.

Whenever the tape is stationary the read/write heads are resting in an inter-record gap. A write command creates one new record on the tape with the record length being determined by the transfer bytecount. The hardware requires that records be at least 16 and not more than 4000 frames. Successive records may be written with different lengths if desired, but this is not recommended because it requires a more complicated program to read the tape.

Each read command reads one whole record. If the transfer bytecount is larger than the record on the tape, this will be indicated by the fact that PT() is incremented by less than the transfer bytecount. If the record on the tape is longer than the transfer bytecount, the whole record is still read but only the requested number of bytes are given to the user and the End-of-Medium bit (octal 10000) will be set in the channel status word (MON only).

Spacing operations are provided which enable the tape to be positioned without data transfer. The forward space operation is given a record count and it moves the tape forwards over that number of data records. If a Tape-Mark is encountered it is spaced over and the operation is then terminated. Thus you can skip to the next Tape-Mark by giving a space command with a large record count. A forward space operation is also terminated if the EOT marker is met. A backwards spacing operation is also available and it works identically except that it is terminated by the BOT marker rather than the EOT one. Note that when a Tape-Mark terminates a space operation, the Tape-Mark has always been spaced over. Consequently the tape stops in a different position if a forward space is terminated by a Tape-Mark than if a backwards space had been terminated by the same mark.

MON: For spacing and error recovery purposes the monitor keeps track of the current tape position by means of a counter. This counter is set to zero when the tape is at BOT, and is incremented by one for every data record or Tape-Mark that is passed over in the forwards direction. Similarly the counter is decremented by one for every record or Tape-Mark that is passed over in the backwards direction. At any moment the counter thus indicates the absolute tape position in terms of records from the BOT. This position count is available to the programmer via QX(c,8).

Due to mechanical imperfections the tape does not necessarily come to rest with the heads exactly in the middle of the inter-record gaps. This means that head position differs according to the direction from which a gap was entered. One consequence of this is that if a tape is initially at BOT and is spaced forwards and then backwards by one record, the final position may not be near enough to the marker to raise the BOT signal. This does not matter, in that the heads are positioned correctly for reading the first record, but it does mean that an apparently balanced sequence of spacing operations may not reproduce the original status.

Under certain unlikely error conditions the hardware may lose track of where the tape is and QX(c,8) will become invalid. This is indicated by the Position Unknown error bit (octal 400) in the channel status word (MON only). If the program is aware of the required position count it may be able to recover by rewinding the tape and doing further space operations.

Data transfer operations always advance the tape position by one record. Space operations however may be terminated prematurely by errors, Tape-Marks, or BOT/EOT markers. A program intending to skip over a particular number of records should compute the required position in terms of QX(c,8) and verify that it has arrived there when the spacing operation completes (MON only). The monitor software has a built-in error recovery procedure that will retry a failed space operation several times before giving up, so if a space operation fails due to an error it is probably irrecoverable. Programs that are aware of the detailed organisation of a particular tape may be able to recover further by searching for a record that contains some known information.

DOS-compatible Files on magnetic tape

Files written on magnetic tape by the DOS operating system have the following format:-

1) A 7-word file header record.
2) One or more 256-word records containing file data.
3) An end-of-file indication. This is just one Tape-Mark.

The file header contains the file name in format:

0 File name in radix50
1 ditto
2 File extension in radix50
3 Department/user numbers
4 File protection code in low-order byte
5 Creation date in DOS format
6 spare

512 bytes of file data are stored in each 256-word tape record. This differs from linked disk files where the first word of each block is a pointer to the next block.

Any number of files may be written on the tape. There is no explicit directory structure since each file header contains the department/user number of that file. Obviously file retrieval will be quicker if files with the same department/user number are kept together on the tape.

Two contiguous Tape-Marks are used to indicate the logical end of tape. This LEOT indicator is written beyond the end of the last file. Like all files the last file ends with an EOF indicator (which is one Tape-Mark). Hence if there are any files on a tape there are always three contiguous Tape-Marks at the LEOT. If an additional file is written at a later time, its header record is written over the last two Tape-Marks.

TMA11/TU10 Tape Transport

These are available in two types: 7 channel and 9 channel. The 9 channel type is preferred unless compatability with other machines is required. The recording method is NRZI. With the 7 channel transports there are three program selectable recording densities and two data formats (industry compatible and Dump). The 9 channel transports always operate at 800 BPI.

The following DDOPR functions are provided (MON only):

SPACE=n Skips forwards over n records
SPACE=-n Skips backwards over n records
WMARK Writes a Tape-Mark
UNLOAD Rewinds and makes unit inaccessable until readied by operator
REWIND Fast winds tape to BOT
PARITY=EVEN
PARITY=ODD
Sets parity (default is odd)
DENSITY=d Sets recording density and format as specified by d. For 9 channel d must be 800. For 7 channel d may be 200, 556, 800 or D800. D800 specifies Dump format which is only available at 800 BPI.

The SPACE and WMARK commands suspend the job until completion. During these operations both the transport and the controller are busy. The UNLOAD and REWIND commands resume immediately leaving the transport busy (ie. rewinding) but the controller free. A function may be initiated on another unit if desired whilst the rewind proceeds. The PARITY and DENSITY commands resume immediately without affecting controller or transport (they simply store information in the monitor).

MTAPE command (DOS only)

This command provides for control of magnetic tape drives. The syntax is:

MTAPE #[channel] [function ne] [argument ne]

where [function ne] specifies the function to be performed:

1: UNLOAD. The tape is rewound and switched offline.
3: REWIND.
4: FORWARD SPACE. Skips forward over [argument ne] records. Stops on EOF or EOT with remainder count in QA.
5: BACK SPACE. Skips backwards over [argument ne] records. Stops on EOF or BOT with remainder count in QA.
6: SETS DENSITY & PARITY from [argument ne] which should be Density*256+Parity

0=200 BPI	0=odd
1=556 BPI	1=even
2=800 BPI	
3=800 BPI dump mode (default).	

7: READ UNIT STATUS to QE. This is bit-coded:

QE&7	Last command was:
	0=Unload
	1=Read
	2=Write
	3=Write EOF
	4=Rewind
	5=Forward space
	6=Back space
128	Tape has just moved over an EOF mark
256	Tape at BOT
512	Tape at EOT
1024	Write locked
2048	Even parity (else odd)
4096	7-track (else 9-track)
8192*density	(as above)
32768	Error caused by last command

The tape unit status is also returned in QE by all other functions of the MTAPE command.

If the spacing functions terminate because an EOF mark is met, the EOF mark is spaced over and counted and then QA is set to the difference between [argument ne] and the number of records or marks actually spaced over.

The MTAPE command actually executes a DOS .SPEC EMT with the function code taken from [function ne] and SPCBLK+4 set from [argument ne]. On completion QE is set from SPCBLK+2 and QA is set from SPCBLK+6. For further information see the DOS Device Driver Package manual. The MTAPE command can also be used to control any other device that implements the .SPEC EMT.

14. RANDOM ACCESS FILING

The READ and WRITE commands allow data to be transferred between an AIMS array and any region of a contiguous file. These commands are not applicable to linked files.

Associated with each I/O channel is a pointer denoted by PT(N) where N is the channel number. The value of this pointer is a number which designates a particular word of the associated file. A value of zero indicates the first word of the file, and so on. The channel pointer is automatically set to zero when a channel is opened. These pointers are 2-word quantities that cannot be referenced when EP()=1.

```
READ #[channel] A(J) [V5: optional bytecount ne]
WRITE
```

When a READ or WRITE command is executed, the file address for the transfer is taken from the current value of the channel pointer. After the transfer, the pointer is automatically incremented by the number of words transferred. The pointer may be referenced in an AIMS program just like an ordinary variable. In particular, the LET command may be used to change the value of the pointer at any time. This mechanism provides a completely general random access capability.

Before using the READ or WRITE commands the data file must be opened using mode 4 or 6. For example, the following program reads words 253 through 268 of the file called 'ACCTS.DAT' into the array B:-

```
 10 ARRAY B 15
100 OPEN #5 4 'ACCTS.DAT' :GOTO 120
110 PRINT 'CANNOT OPEN FILE' :STOP
120 LET PT(5)=253 :READ #5 B() :GOTO 140
130 PRINT 'ERROR READING FILE' :STOP
140 PRINT 'THE POINTER IS NOW'PT(5)
>RUN
THE POINTER IS NOW 269
>
```

Line 120 initialises the channel pointer and executes a READ command. AIMS reads words from the file, starting with the 254'th word, and places them in successive cells of the array B. This continues until the end of the array is reached. The number of words read is thus determined by the array dimension. After the transfer the channel pointer is updated to point to the word following the last one read.

The array name may be subscripted, giving the capability of reading into part of an array like

```
120 LET PT(5)=253 :READ #5 B(9) :GOTO 140
```

In this case 7 words will be read into cells 9 through 15 of the array. V3: note that the transfer always continues to the end of the array. V5: Transfers may be terminated before the end of the array by specifying a non-zero [bytecount ne].

ERRORS WITH READ AND WRITE COMMANDS

The READ and WRITE commands will fail if an attempt is made to transfer over the end of a file. In this case the transfer is aborted as soon as the attempted overrun is detected, and the amount of data actually transferred can be obtained from the channel pointer value. Do not assume that the transfer has been done upto the end of the file; in most cases no data is transferred.

The READ command can also fail due to device errors such as parity or seek failures. DOS: QE contains the DOS .TRAN error status reply. MON: The channel status word QX(c,5) gives a precise description of the error. Take note: the information contained in the channel status word is useful both for distinguishing different types of failure and for diagnosing hardware errors. Programs should print QX(c,5)@R8 and programmers should pay attention to it.

STRUCTURED DATA

The READ and WRITE commands transfer words directly between a file and an array without regard to the format of the data within the array. The LET and PACK commands may be used to fill the array with numbers, bit-patterns, or strings of characters, forming a logical record in any desired format. For example, some of the numbers in the array may be pointers to other records, allowing the construction of hierarchical data structures.

CREATING CONTIGUOUS FILES - ALLOC COMMAND

DOS supports two types of files: linked files and contiguous files. Linked files are used mainly for storing text, and they consist of a number of blocks scattered anywhere on the storage medium. The blocks for a particular file are chained together in a linked list, enabling the whole file to be accessed sequentially once the address of the first block is known. Since the only way to locate a particular block of a linked file is to trace the chain from the first block, linked files cannot be used for random access. Linked files are created using mode 1 of the OPEN command. This enters the specified filename into the directory and also stores the address of the first data block of the file. Additional data blocks are linked on as required whilst the file is being written. When the file is CLOSEd additional information is stored in the directory entry, such as the length of the file and the address of the last block.

In contrast, each contiguous file occupies a single region on the storage medium. A contiguous file is thus described completely

by the device address where it begins and its length. Contiguous files are created by means of the ALLOC command:

ALLOC [ne] #[channel] [filename]

This command searches the directory for the specified file and fails if the file already exists. Otherwise it searches the storage medium for a free area of at least [ne]*64 words. The command fails if no suitable region can be found. If all is well [ne]*64 words of the found region are allocated to the new file, which is entered into the directory together with the size and starting device address.

Once a contiguous file has been created by an ALLOC command, the associated region on the storage medium may be used for random access filing. To do this it is necessary to open the file in mode 4 or 6. The OPEN command in these modes simply searches the directory for the file and remembers the starting device address and file length. Random access transfers may then be done using the READ and WRITE commands without incurring any directory overheads.

A major problem with contiguous files is the difficulty of finding a suitable contiguous region on the storage medium. Although the total free space on the medium may be quite large, it often happens that there is still no region big enough for the file that one is trying to allocate. This happens because of the linked files, which may be scattered all over the medium causing the free regions to be split up into lots of small areas. The solution to this problem is to allocate all required contiguous files when the storage medium is relatively empty. If it is not known in advance exactly what files will be required, a single large contiguous file may be allocated to reserve a suitable area. This file may later be deleted and re-allocated to one or more contiguous files as required.

INPUT/OUTPUT TO PHYSICAL DEVICE ADDRESSES

For certain applications, such as listing disk directories, it is necessary to gain direct access to the storage medium without the constraints of a file structure. This may be done by INITing the device on a particular channel, and then OPENing the channel in mode 5. The READ and WRITE commands may then be used as described under 'Random Access Filing' above. In this case the channel pointer specifies a physical word-address on the device rather than a relative address within a file.

This direct access method should be used with extreme care since it allows corruption of the device file structure. The mode 5 open is only available to EXEC-privileged programs. See section 22.

As an example, the following privileged program uses mode 5 to read the MFD blocks on device DK0 and print a list of all department/user numbers:-

```
>LIST
  8 ARRAY A 255
100 INIT #5 'DK' :OPEN #5 5 :GOTO 120
110 PRINT '?CANNOT OPEN DISK IN MODE 5' :STOP
120 LET PT(5)=256 :GOSUB 800 :REM READ MFD BLOCK 1
130 IF A()=0 :PRINT 'END OF MFD' :STOP
140 LET J=1 PT(5)=256*A() :GOSUB 800 :REM READ NEXT MFD BLOCK
150 PRINT (A(J)65535)/256',QA, :TAB 10 :PRINT A(J+1)
160 LET J=J+4 :IF J<256 :GOTO 150
170 GOTO 130
800 LET P=PT(5) :READ #5 A(): RETURN
810 PRINT '?ERROR 'QE' READING DISK BLOCK 'P/256 :STOP
>RUN
  1,  1    2
 16, 16  990
 16, 17 1000
END OF MFD
>
```

File Structures and the MOUNT command (MON only)

The MONITOR system of file directories is written not on a physical device but on a logical entity called a STORAGE STRUCTURE. This is a virtual device providing a vector of bytes numbered from zero upto some maximum depending on the capacity of the medium. The device is randomly accessible and word addressable. Each storage structure is mapped onto one or more physical device units by the MONITOR software:

a FILE STRUCTURE is a system of directories, bitmaps, etc.
which is written on a

STORAGE STRUCTURE, a virtual device which is mapped by the
monitor onto

one or more physical device units.

Each file structure has its own name which can be upto six alphanumeric characters long. This name need not bear any resemblance to the names of the devices on which the structure is stored. A file structure residing on a particular diskpack might be mounted one day on unit RPA0 and the next day on unit RPA2. It could have the same file structure name on each occasion.

Physical unit names are listed on page 59.

In practice nearly all jobs are concerned with accessing files rather than physical devices, so most INITs will be specifying file structure names rather than physical unit names. For example the command INIT #5 'DATA:' will connect channel 5 to the file structure called DATA, and the program need not know which physical unit the volume is mounted on.

The MOUNT command provides a way of telling the system that a file structure is present on a particular physical unit.

MOUNT #[channel] [function ne] [se]

The functions are:

0 MOUNTs a file structure and defines its name
1 DISMOUNTs a file structure, making it inaccessible

Function 0 - MOUNT

Function 0 causes the monitor to take note of the file structure already existing on a particular volume. [se] specifies the physical unit name where the volume is mounted, and also the logical name by which the associated file structure is to be known. [se] should evaluate to a string of the general form

[physical unit name]:[structure name]

For example MOUNT 0 'RPA1:FRED' makes files on RP11 disk drive number 1 accessible via the file structure named FRED. After this operation any job in the system may execute an INIT 'FRED:' command to access the file structure (subject to user capabilities).

The execution of a MOUNT command causes the monitor to read relevant directory information from the volume. The command will fail if the volume does not have a recognised directory.

Function 1 - DISMOUNT

Function 1 is used to withdraw a file structure from use. [se] should evaluate to a string of the form

[file structure name]:

The command will fail if the file structure is currently being referenced by any job in the system. If the structure is not in use the file structure name is deleted from the monitor's table of structure names, and all monitor information about the file structure (eg: bitmaps) is removed from memory.

After a DISMOUNT the file structure can no longer be accessed by name, and any job attempting to INIT it will get a device not found error. The corresponding physical devices can still be accessed in mode 5 (subject to user capabilities).

Note that the DISMOUNT command only operates on resident monitor data structures, it does not affect the state of the files on the storage media.

Warning! when exchanging diskpacks it is essential to DISMOUNT the old pack before removing it and bringing up the new one. Failure to do so may result in the new pack being overwritten with information pertaining to the old one.

The AIMS executive program provides MOUNT and DISMOUNT commands for convenience. The STRUCTURES and RESOURCES commands

both give a list of the currently existing structures.

FAST ACCESS DIRECTORY - MODE 6 OPEN (DOS only)

The DOS file handling operations are rather inefficient and involve several disk transfers to find a given file. This can cause unwarranted delays when several data files have to be opened one after another. Programmers may avoid this to some extent by dedicating I/O channels to the most often used files. These channels may then be opened once at the beginning of the program and will give fast access to the data files thereafter. But this technique cannot be used if there are more than two or three such files since there are only 8 channels and 1, 2 and 4 are usually in use for terminal I/O and program overlaying.

The mode 6 OPEN, which is an AIMS option, eliminates this problem for contiguous files. This option provides an in-memory directory which stores the particulars of the most often referenced files. The OPEN command in mode 4 or 6 searches this directory before searching the device. If the file is in the fast access directory the OPEN is instantaneous and does not involve any device transfers. If the file is not found in the fast access directory the device is searched in the normal way.

The fast access directory can only hold a small number of files. This number is a configuration parameter which is normally set to 32. The files can come from any mixture of disks. The directory is initially empty when the system is started. The programmers control which files get entered into the directory as follows: only those files that are OPENed in mode 6 are entered into the directory. Once a file is in the directory both modes 4 and 6 benefit from the faster access.

Files are entered into the directory in a circular fashion, so that if the directory capacity is exceeded the oldest entry will be overwritten.

Warning! there is no automatic way of deleting entries from the fast access directory when a diskpack is dismounted. If you change a diskpack without clearing the fast access directory the system may continue to access the old files on the new diskpack, causing catastrophic corruption of the file structure.

In order to get the full benefit from the fast access directory, it is essential to avoid channel INITs, since the INIT operation itself involves DOS overlaying activity. It is necessary for a channel to be INITed before it can be OPENed, but this need only be done once at the beginning of the program, or only when switching to another device. Since the RELEASE operation disconnects the channel from the device, RELEASEs should also be avoided. Thus the correct procedure is

At beginning of program:

INIT channel to required device, or ensure channel
is released if you want the default device.

To open the next file:

```
OPEN #[channel] 6 [filename]
```

Since the OPEN command performs a CLOSE implicitly, it is not necessary to CLOSE the channel before going on to OPEN the next mode 6 file.

You can examine the mode 6 directory by giving the EXEC command

```
.DIR M6:
```

Note that mode 6 is really a fast way of opening contiguous files in mode 4. The fast access directory does not work for any other mode of opening and does not apply to linked files.

15. FACILITIES FOR FILING AIMS PROGRAMS

CALL AND SAVE

The SAVE and CALL commands allow programs to be saved as text files in a format that is compatible with other DOS programs such as the Editor and PIP.

SAVE #[ne1] [filename],[ne2],[ne3]

Creates the specified file on the device assigned to channel [ne1], deleting any previous file with that name. Writes program lines [ne2] to [ne3] inclusive to the file in the same format as that produced by the LIST command. Closes the file.

If ,[ne2],[ne3] are absent, the whole program is saved.

If #[ne1] is absent, channel 3 will be used.

CALL [ne1] #[ne2] [filename]

Opens the specified file on channel [ne2]. This file must be in ASCII format. Reads the file line by line and CODEs each line. This adds the CALLed program to the program already in memory (if any) in an interleaving mode. Existing lines are unchanged unless they have the same number as lines of the called program.

When the end of the file is reached, control goes to the line following the CALL command if [ne1] is absent. Otherwise a RUN [ne1] is performed, except that system and user-defined simple variables are not changed. Array names are defined as with an ordinary RUN.

Since SAVE operates by translating the binary-image of the program into a text file, the values of variables and the contents of arrays are not written to the file and will not be restored when the file is later CALLed.

LIBRARY DIRECTORY

Directory [16 17] is designated as a library area where commonly used programs may be kept. The CALL command will search this area if the specified file is not found in your own directory.

LOAD AND DUMP

The LOAD and DUMP commands transfer a binary-image of a program between the user memory area and a specified region of a contiguous file. The transfer begins at the point specified by the channel pointer, and ends when the whole program has been transferred. The channel pointer is updated to point by the amount transferred. (channel pointers are explained under 'Random Access Filing' in section 14)

DUMP #[channel]

LOAD [ne1] #[channel] [ne2] [ne3]

All the arguments are optional. If #[channel] is omitted, channel 4 is assumed. If [ne1] is present and non-zero, it specifies the program line number at which execution is to begin. If [ne2] [ne3] are absent, the LOADed program will completely overwrite the program that executes the LOAD command. If [ne2] [ne3] are present, they specify a range of lines of the existing program that are to be preserved. The range consists of lines [ne2] through [ne3]-1 inclusive. These lines will be inserted into the LOADed program, overwriting any LOADed lines with the same numbers. This enables selected string or numerical data to be passed from one program overlay to another.

Any number of program overlays may be stacked one above another in one file. It is the user's responsibility to remember the position of each overlay in the file, and to set the channel pointer appropriately before executing a LOAD or DUMP command. If the pointer is set incorrectly, causing garbage to be read into memory, AIMS will reload EXEC. Note that the pointer values are assumed to be of the form [file block number]*[blocksize] and if the pointer is not a multiple of [blocksize] it will be rounded down to the nearest multiple. [blocksize] is 64 for the RF11 fixed-head disk, and is 256 for most other devices. DOS: After a LOAD or DUMP the pointer is incremented by the number of words transferred rounded up to the next multiple of [blocksize].

Initially the user must create a contiguous file of a suitable length by means of the ALLOC command. Before the first LOAD or DUMP operation the user must open the file by means of the OPEN command in mode 4 or 6. Subsequent LOADs and DUMPs are then performed directly between the user memory area and the device with zero directory overhead.

The following example shows how to create a file containing three program overlays. The three programs are assumed to be available as 'saved' files PROG1, PROG2 and PROG3. The RK11 disk is assumed so that [blocksize] is 256.

```
>ALLOC 48 #4 'PROG.DMP' :P 'OK
OK                          [allocate a 3072-word contiguous file
>OPEN #4 4 'PROG.DMP' :P'OK
OK                          [open it in mode 4
>CLEAR
>CALL 'PROG1                [call in the first part of the program
>DUMP #4 :P PT(4)           [dump it
```

```
 1024                          [clearly it is under 1024 words long
>CLEAR
>CALL 'PROG2                   [call in the next part
>DUMP #4 :P PT(4)              [dump it, starting at word 1024 of the
 1536                          [file.  This one is only 512 words long
>CLEAR
>CALL 'PROG3                   [call in the last part
>L PT(4)=PT(4)+512 :DUMP #4 :P PT(4)
 3072                          [last part is 1024 words long
>L PT(4)=0:LOA1 #4             [load and run the first part
```

In this example we incremented PT(4) after the second DUMP to reserve a 512-word area on the file in case PROG2 is expanded in the future. We were also careful to print PT(4) after every DUMP command so that we know where each overlay begins.

When modifying an existing overlay of a dumped file it is useful to know the exact size of the overlay before dumping, so as to avoid overwriting the following overlay. Whilst the system function UC() gives the overall size of the program in memory, the size of the dumped overlay may be significantly smaller. Unfortunately it is not easy to calculate the exact size in advance. It is approximately (UC()-QS)/2 words, but this estimate cannot be relied upon. There are three ways of getting round this problem: (1) reserve extra space for each overlay so that an increase in size does not affect the next one, (2) dump the overlay into an auxiliary scratch file and print PT(4) to find out the exact size, (3) be prepared to re-dump all the overlays. This may be done effortlessly by using an OBEY file as described in section 24.

Unless disk space is limited it is probably easiest to allocate a fixed amount, say 1024 or 2048 words, for each overlay. The N'th overlay may then be accessed in a standard way. Since contiguous files are of fixed length, it is also wise to ALLOCate space for one or two more overlays than are planned. Otherwise it will be necessary to delete the file, re-ALLOCate it, and re-DUMP all the overlays when you run out of space.

It is recommended that LOAD and DUMP be used for normal filing and overlaying of frequently-used programs, and that CALL and SAVE are used to keep permanent back-up copies of all programs. Save-files have the long term advantage that they are compatible with future versions of AIMS, whereas dump-files may become unuseable due to changes to the internal AIMS coding. If this occurs, a dump file can be converted to the new coding by loading and saving it under the old AIMS, and calling and dumping it under the new AIMS.

16. ERRORS

Error conditions fall into two classes:-

1) ERRORS: These are caused by incorrect use of the language or by logical mistakes in your program. For example a badly specified command or a reference to a variable that has not yet been set. In these cases the program will almost certainly need correcting and so AIMS normally stops and prints an error message.

2) FAILURES: These occur when a correctly written program encounters an unexpected event. A typical example is an attempt to open a file that does not exist. This is not necessarily an error, it may have been caused by a user typing the wrong name. In these cases the command will fail causing execution to 'fall through' to the next line of the program (see page 50). If a command can fail for several reasons these are distinguished by a code number that is set into the system variable QE.

As explained in section 11, an AIMS program may trap all errors by setting the system variable QQ to the line number of an error handling routine. If an error occurs this routine will be entered via a 'GOSUB QQ' with an error code in QE.

If QQ is zero, a standard error message is printed and the program stops. This message is of the form

? [error letter]
[the line that caused the error]

A question mark is inserted in the line to mark the point at which the error was detected.

CODE	LETTER	MEANING
0	S	No space left.
1	C	Command word not recognised
2	E	Error in syntax
3	V	Value is outside allowed range
4	U	Reference to undefined variable
5	L	Reference to non-existant line
6	T	Referenced line is the wrong type
7	G	GOSUB overflow or RETURN underflow
8	A	Bad function or array reference
9	I	Error in I/O operation
10	O	Loss of accuracy in calculation
11	Q	Use of control-C abort key
12	Y	No more memory available for this job
13	Z	Command not implemented
14	P	Inadequate capabilities. Attempt to perform a privileged operation which the appropriate bits of JS(0) do not allow
15	F	No more space left on storage medium.

Note that QQ only traps the errors tabulated above. It is not possible to trap failures using QQ.

QA when execution Stops

If execution stops due to an error, the system variable QA is set to the number of the line containing the error.

If the program stops due to a STOP command, QA is set to the number of the line containing the STOP command.

If the program stops due to the user typing ^C, QA is set to the number of the line that was about to be executed. In this case execution may be resumed at the correct point by giving the command GOTO QA (assuming the user has not changed QA since the stop).

A program that stopped due to an error or a STOP command is best resumed by the command GOTO NL(QA) which will continue execution at the line following the error or STOP command. A GOTO QA would merely re-execute the offending line, possibly executing part of the line a second time, and certainly causing another error or stop.

The DO command adds a complication to all the cases described above: if a program stops during execution of one or more DO commands, QA is always set to the number of the line containing the highest-level DO command. If the stop is due to an error or ^C, the relevant program line is normally printed, and this will be the line indicated in the preceeding paragraphs. With DO commands this line will probably differ from that indicated by the value of QA. If DO commands are chained several levels deep, a ^C can interrupt execution at any level, and the only indication of the interruption point is the program line printed. Execution of such chains cannot be correctly resumed, except by restarting at the line containing the highest-level DO command.

Examples

```
>LIST
  5 $HI
 10 ARRAY A 5
100 PRINT 'TIME IS 'T"0'CLOCK"
>RUN
?U
100 PRINT 'TIME IS 'T?"0'CLOCK"
>P QA QE
 100   4
>                          [the variable T is undefined
>PRINT A(6)+2
?V
  0 PRINT A(6)?+2          [subscript 6 is larger than array dimension
>PRINT $A
?T
  0 PRINT $A?              [line A is not a dollar-line
>PRINT 5(2)
?T
```

```
  0 PRINT 5(2)?          [line 5 is not an array
>PRINT $6 $5
?L
  0 PRINT $6? $5         [line 6 does not exist
```

Note that the letter following the ? indicates the type of error as given in the table above. It is important to remember this letter if you are going to ask someone else to explain an error. The offending program line is printed and sometimes a ? is inserted to mark the error point. In this case the error is always to the left of the mark. It has been found in practice that programmers often assume that an error is of a certain type without looking at the error letter actually printed. This procedure is suboptimal. See also the discussion of array errors on page 21.

ABORT KEYS

control-C

This is the normal way to stop a running program. AIMS will return to edit mode and will print the program line it was about to execute. Also causes error 11 as noted above. If error traps are enabled this effectively causes an interrupt to line QQ of the AIMS program.
If you type control-C when the program is waiting for terminal input it is not effective until you press carriage-return to terminate the input-wait.

control-O

Aborts a running AIMS program and returns to the AIMS executive. Equivalent to execution of an EXIT command in the program.

FAILURE ERROR CODES RETURNED BY MONITOR (MON only)

When a command fails the monitor returns an error code in the system variable QI. This code is in two parts: QI = 256*ER + EC. EC identifies a general class of errors, and ER identifies each type of error uniquely. Programmers will find debugging much easier if they print QI/256 and QA and take note of their values. It has been found in practice that programmers often assume that a failure is of a particular type when in fact it is of another type. Failure to observe QI assists this process.

If you think that the monitor returns a misleading code in QI under certain circumstances this should be reported to Arbat.

Under DOS an error code is returned in QE rather than QI. To maintain compatability AIMS V5 translates the MONITOR error code into DOS form and stores it in QE. Hence programs originally written to run under DOS should run ok under MONITOR. New programs written for AIMS V5 should use QI rather than QE because the QI codes are more specific and reliable.

The possible EC and ER codes are tabulated below. Note that all failures are uniquely identified by the ER value, and the EC value may be regarded as a classification of the ER values. There are however some failure conditions that are not specifically covered by an ER code, and these only return an EC code (leaving ER zero). All codes are even numbers, and EC is never zero.

NAME	QI/256	REM	FAILURE CONDITION
EC.PGM		2	Error in program
ER.MIO	16	2	Illegal MIO code
ER.BOB	17	2	Parameter block address out of bounds
ER.AOB	18	2	Particular argument address out of bounds
EC.VAL		4	Illegal value
ER.COM	24	4	Invalid command code
ER.CHA	25	4	Invalid I/O channel number
EC.IFN		6	Inappropriate function
ER.CNO	32	6	Channel not open
ER.IMD	33	6	Channel open in wrong mode
ER.IDF	34	6	Illegal function for this unit or structure
ER.IFF	35	6	Illegal function for this file
EC.RDY		8	Device not ready
ER.DWN	40	8	Device down
ER.OFL	41	8	Device offline
ER.GON	42	8	Device withdrawn
ER.WLK	43	8	Device write-locked
EC.END		10	End of data (including end of medium, end of file)

EC.TRA		12	Error during data transfer, see Channel Status word QX(c,5) for further details
ER.DAT	56	12	User data, or UFD transfer
ER.BMR	57	12	File system bitmap read
ER.BMW	58	12	File system bitmap write
EC.SPA		14	Not enough space on device
ER.FUL	64	14	Storage medium full
ER.QEX	65	14	User's quota exhausted
ER.ALC	66	14	Insufficient contiguous space available
EC.SYN		16	Syntax error in file or device specification
ER.DEL	72	16	Illegal delimiter character
ER.LEN	73	16	Name or path too long
EC.PTH		18	Cannot access path
ER.DNF	80	18	Device or structure not found
ER.DPR	81	18	Device or structure protected
ER.DAS	82	18	Device assigned to someone else
ER.UNF	83	18	No such device or structure
ER.ARU	84	18	File area not found
ER.APR	85	18	File area protected
EC.FIL		20	Canot access file
ER.FNF	88	20	File not found
ER.FPR	88	20	File protected
ER.FLK	90	20	Someone else has exclusive access to file
EC.FEX		22	File already exists
EC.CAP		24	Inadequate capabilities
EC.NIM		26	Facility not implemented in this system

In the case of the EC.RDY and EC.TRA errors, further information about the state of the device can be obtained from the Channel Status word QX(c,5).

ERRORS FROM INPUT/OUTPUT COMMANDS (mainly DOS)

These events are all failures, except those marked with a ? which are errors as described above.

CONTEXT	EVENT	MEANING
#[ne]	?V	Channel number [ne] less than 1 or greater than 8
INIT	?Y	Not enough memory left for channel control-block
INIT	fail	No such device (QE irrelevant)
OPEN	?Y	Not enough memory for data-buffer
OPEN	fail	Error code in QE (see below)
INPUT	?I	Channel not INITed, or not OPEN, or device not capable of input
INPUT	fail	Error code in QE (see below)
PRINT	?I	Channel not INITed, or not OPEN, or device not capable of output
LOAD	?Y	Not enough memory to load program, or not enough for use by DOS
LOAD	.	Attempt to load garbage
LOAD	fail	Error code in QE (see below)

CONTEXT	QE/256	MEANING (DOS)
OPEN in mode 0 (=openi)		
	2	File nonexistant, or being modified by someone else, or nonexistant directory
	3	More than 62 OPENs without corresponding CLOSE
	6	File protected
	7	Device not capable of input
OPEN in mode 1 (=openo)		
	2	File already exists
	7	Device not capable of output, or directory is protected against file creation
	11	No user file directory for given [dept user]
	13	Illegal file name
OPEN in mode 4 or 6		
	13	Illegal file name
	for other error conditions, QE is bit coded as follows	
	bit 6	0 file is not contiguous
	bit 7	0 file does not exist

CONTEXT	QE/256	MEANING (DOS)
ALLOC		
	2	File already exists
	7	Directory protected against file creation
	10	Directory full
	11	No file directory for given [dept user] number
	13	Illegal file name
DELETE		
	2	File does not exist anyway
	6	File protected
	12	File in use
	13	Illegal file name
RENAME		
	2	Old name does not exist
	2	New name already exists
	6	File protected
	14	File in use
	15	Illegal file name
INPUT		
	bit coded as follows:-	
	64	End of file, or end of medium
	32	Device parity error
	4	Character parity error
	2	Checksum error
	1	Invalid line terminator, or not enough space to read in complete line
PRINT		
	?F	Device full.
	Apart from ?F, the PRINT command always succeeds.	
READ and WRITE		
	Bit coded as follows:-	
	32768	Hardware error such as parity or seek failure
	16384	End of medium (see note below)

The end-of-medium bit is only set for sequential media like paper tape. If you are using mode 5 to access bulk-storage devices like disks and DECtape, an attempt to access beyond the end of the medium will cause a fatal DOS error. When using mode 4 or 6 to access a contiguous file, AIMS will safely trap any attempt to access beyond the end of the file, but the end-of-medium bit is not set. So programmers should test the harware error bit first and assume an end-of-file condition if the bit is not set.

Care should be exercised when testing QE for error codes. Codes that appear in QE/256 should be tested as such, since the low order bits of QE may contain random junk. Bit-coded conditions should be tested using the logical & operator rather than testing for equality, since QE may contain other random bits. Note also that if bit 32768 is set, QE will appear to be negative. If you intend to print QE it is advisable to mask it with a sixteen bit mask like 1_16-1.

TRAPPING ERRORS WITH QQ

In a production environment is is often desirable to prevent program errors from being communicated to the end-users of the system. This may be done by setting QQ non-zero. Should an error occur, AIMS will GOSUB QQ without printing any message. The program at line QQ may then take corrective action.

The programmer should exercise exteme care when using this error trapping facility since it can give rise to subtle problems that are not easily diagnosed.

The following example shows an error trap handler that stops the current activity if the user types control-C, and exits to a special error recovery overlay if any other error occurs:-

```
100 LET QQ=900

900 RETURN :IF QE=11 :PRINT 'STOPPED' :LET QQ=900 :GOTO 500
910 PRINT 'FATAL ERROR 'QE' AT LINE 'QA
920 LET PT(4)=4096 :LOAD 1
```

It is possible to resume execution after an error trap by using the stacked return line number inconjunction with the GOTO or RETURN commands, or using the NL() function to skip a trap-causing program line. Note however that programs containing DO commands cannot always be correctly resumed because the stack does not contain sufficient information.

17. COMMAND SUMMARY

ACCEPT ?[ne] [assignment list]

Reads a single character from a terminal keyboard.

ACOMP A(J) B(K) N

Compares the block of N cells in the two arrays.

ALLOC [ne] #[channel] [filename]

Creates a contiguous file of length 64*[ne] words on the device assigned to [channel], with name [filename].

AMOVE A(J) B(K) N

Moves the block of N cells from A(J) to B(K).

BYE [ne]

Logs job [ne] off the system. Default job is self.

[number1] ARRAY [name] [number2]

Sets up an array in line [number1] of the program. [name] is a 1 or 2-character variable name which is automatically assigned the value [number1] by the RUN command. [number2] specifies the highest legal array subscript. The array will have [number2]+1 cells numbered from 0 through [number2].

CALL [start] #[channel] [filename]

Calls the specified program file and performs a 'RUN [start]', except that previously existing simple variables remain defined

CLEAR [ne1],[ne2]

Clears program lines [ne1] through [ne2].

CLOSE #[channel]

Closes the specified channel. Default is 4.

CODE [se]

Obeys the string [se] as if it had been typed as an AIMS command.

CORE [ne]

Adjusts the user's memory allocation so that the free space as given by QS is at least [ne] characters. Command will fail if insufficient memory available.

DDOPR #[channel] [command se]>$[reply ne]

Performs device-dependent operations.

DELETE #[channel] [filename]

Deletes a file from the device assigned to channel.

DIAL [dialler ne] [phone number se]

Initiates a phone call.

DO [ne]

Executes program line [ne]. The line is executed as if it occupied the position of the DO command. Control returns to the line following the DO command.

DUMP #[channel]

Dumps the whole program in binary-image form.

EXIT

Returns control to the AIMS executive program EXEC.

GARB

Performs a garbage collection.

GOTO [ne]

Transfers control to line [ne] of the program.

GOSUB [ne]

Stacks the number of the current line and transfers control to line [ne].

INPUT [echo] [timeout] #[channel] ?[ne] [assignment list]

If ?[ne] is present, prints line $[ne] as a cue to the user. If ?[ne] is absent, prints * as a cue. Reads a line of text from the specified channel, default 2. Decomposes the string as for the PUT command.

IF [ne] ...

Continues execution of the current line if the value of the low order 16 bits of [ne] is non-zero.

INIT #[channel] [devicename]

Initialises the channel and attaches the specified device.

LET [name1]=[ne1] [name2]=[ne2] ...

Assigns the value of [ne1] to the variable [name1], and so on.

LINE [line ne] [function ne] [argument]

Command for controlling communication lines and terminal interfaces.

LIST [ne1],[ne2]

Lists lines [ne1] through [ne2] of the program on channel 1.

LOAD [start] #[channel] [from] [to]

Overlays the AIMS program area with the binary-image stored in the file open on [channel]. Starts the program at line [start]. Lines [from] through [to]-1 passed as arguments.

... :LOOP

Executes the current program line again. Only makes sense as the last command on a line. Quicker than a GOTO. May be put in by terminating the program line with colon.

MOUNT [function ne] [se]

Mounts/dismounts detachable storage media, and associates logical file-structure name with physical device.

MTAPE #[channel] [function ne] [argument ne]

DOS: Device-dependent control command for use with magnetic tape. Returns tape unit status in QE and residue count in QA (if applicable).

OPEN #[channel] [mode] [filename]

Opens the specified file.

PACK A(J) [separator se],[ne1] [ne2]

Packs dollar-lines [ne1] through [ne2] into array, using [separator se] to mark the end of each string in the array.

PRINT #[channel] [se]

Writes the string [se][newline] onto [channel], default is 1.

PRINT #[channel] [se],

As above, but does not append [newline].

PUT [se] [search mode] [destination] [look for] [replacement]

Decomposes the string [se] as specified.

READ #[channel] [array name]([subscript]) [V5: opt bytecount]

Fills the array in line [array name] with binary data read from the file open on [channel].

RELEASE #[channel]

Performs a CLOSE if channel open. Then releases the device.

REM [comment]

[comment] is ignored. Control goes directly to next line.

RENAME #[channel] [newname],[oldname]

Renames a file. [oldname] and [newname] are separate string expressions, separated by a comma.

RETURN [ne]

Unstacks the line number previously stacked by the last GOSUB command and assigns it to the system variable QA. Transfers control to the next line greater than or equal to QA+[ne]. If [ne] is absent a value of 1 is used.

RETURN : ...

Unstacks the return line number into QA and continues executing the current line.

RUN [ne]

Deletes all simple variables, sets all system variables to their default values, and performs ARRAY name assignments. Then starts the program at line NL([ne]-1).

SAVE #[channel] [filename],[ne1],[ne2]

Saves program lines [ne1] through [ne2] as specified text file.

SCAN [recsize ne] [keylen ne] [array] [mode] [key ne] [count]

Scans the array for the specified key and sets QA.

SETNAM [se]

Sets the job's program name to the first six characters of [se]. This name is used by the SYSTAT printout.

STOP

Stops execution of the AIMS program and sets QA to the number of the stop line.

SWIFT [line ne] [function ne] [arg]

Special command for communication with S.W.I.F.T message switching network.

TAB #[channel] [ne] [se]

Prints the [se] repeatedly until column [ne] is reached. Default channel is 1. Default [se] is one space.

UNLESS [ne] ...

Continues execution of the current program line if the value of [ne] is zero.

UNPACK A(J) [separator se],[ne1] [ne2]

Unpacks array into block of strings in lines [ne1] through [ne2], using [separator se] to detect end of each string in the array.

VGARB

Deletes all user-defined variables that are not referenced in the current program.

WAIT [ne] [wake mask ne]

Suspends execution for [ne] tenths of a second or until specified event happens.

WAKE [se]

Wakes all jobs named [se] if they are wake-enabled.

WRITE #[channel] [array name]([subscript]) [V5: opt bytecount]

Writes the content of the array in line [ne] in binary to the file open on [channel].

X [number] [look for] [replace by]

Changes [look for] to [replace by] in line [number] of the program and prints line [ne].

18. AIMS EXECUTIVE PROGRAM - EXEC

EXEC is a privileged AIMS program that is run when you have logged onto the system or whenever you give the EXIT command. The main purpose of EXEC is to provide an environment in which the user may quickly and easily develop his own AIMS programs. There are EXEC commands for loading, running, calling, copying, deleting and renaming files, for listing device directories, and for examining the state of the system.

Most EXEC commands may be abbreviated to a single letter. In the command descriptions the element [filename] denotes a file name, with optional device name, unit number, file extension, and department/user numbers. The full syntax of a [filename] is

[dev][unit]:[name].[ext][[dept user]]

For example:

FRED FRED.BAS DT2:FRED FRED[16 16] DK1:FRED.BAS[40 42]

Some commands allow a class of files to be specified rather than just one. This is done by means of a [filespec]. The syntax of a [filespec] is identical to that of a [filename] except that the character * may be substituted for either the [name] or the [ext] fields. For example:

FRED.*	denotes all files whose name is FRED, with any extension.
*.BAS	denotes all files whose extension is BAS, with any name.

Most of the commands that take a [filename] as an argument will also accept a list of [filename] fields separated by commas.

It is sometimes necessary to qualify a command by specifying some optional feature. This is done by means of a 'switch', which is defined to be of the form /X where X is the name of the switch.

Whenever EXEC is waiting for a command it prints

.

You can get a help message for any command by typing the command word followed by a question mark. For example
.REN ?
will give help on the RENAME command. The following commands are allowed:-

CALLING PROGRAMS

RUN [filename] Loads and runs a dumped AIMS program. The default file extension is DMP. This command is used for running most production programs. The library area [16 17] is searched if the program is not found on your own area. Eg:

```
.R ED
EDITOR V1J
FILE:
```

EXECUTE [filename]
Calls and runs a saved AIMS program. Default file extension is BAS. The library area [16 17] is searched if the the program is not found on your own area.

EXECUTE E command without a filename enters AIMS edit mode allowing program development. Eg:

```
.E
>
```

LOAD [filename] As RUN except it does not start program execution. This is used to bring a dumped program into memory so that it can be modified.

CALL [filename] As EXECUTE except it does not start program execution. This is used to bring a saved program into memory so that it can be modified.

All of the above commands may be abbreviated to a single letter. Any command may be prefixed with the letter Z to give the program executive privileges if these are allowed. (Ie: ZR, ZE, ZL AND ZC)

If the executive does not recognise a command word, it treats the word as a filename and attempts to RUN the corresponding .DMP file. This permits user-defined extensions to the executive command repertoire. For example, since ED is not a valid full or abbreviated executive command, the command

```
.ED
```

is equivalent to .RUN ED and will cause the file ED.DMP to be loaded and executed.

FILE CONTROL

COPY [destination]=[source]
Copies a file from one place to another. [source] and [destination] are [filename] specifications.

Switches:

/U Update. IE. if [destination] file already exists delete it before copying. Without the update switch, an existing contiguous file may be copied into, provided it is large enough. /U forces re-ALLOCation of the file.

/G:n Sets QG to n. Useful when copying files containing very long strings.

Examples

```
.COPY TEST.BAS=ABC.BAS
.CO DK1:TEST.BAS=ABC.BAS
.CO TEST.BAS=ABC.BAS/U      [delete TEST.BAS if necessary
.CO DK1:ACCTS.SAV=DK0:ACCTS.DAT
.CO DK1:=MEMO.TXT           [destination name same as source
.CO PP:=MEMO.TXT[16 17]
.CO FRED.DAT=SAM.DAT/U      [reallocate FRED.DAT
```

DEL [filespec] Deletes the specified files.

Switch:

/P Print names of deleted files.

DIR [filespec] Lists the specified subset of the user's file directory.

Switches:

/F list in abbreviated (Fast) format

/S list in expanded (Slow) format

/O list UFD entry in octal

FREE [dev]: Gives the number of free blocks left on device [dev].

MAP [dev]: Prints a map for device [dev] showing which physical blocks are occupied by files.

RELEASE Releases I/O channels 5 to 8. Useful for closing files left open by last program.

REN [newname]=[dev]:[oldname]
Renames the specified file. [newname] may include an octal <protection> field.

TYPE [filename] Lists any text file in a paginated format. Useful for getting neat listings of saved AIMS programs.

Switches:

/L:n Length of page in lines. Default 69. The number of lines of text appearing will be 7 less than this since the heading takes 7 lines.

/W:n Margin. Defines width of paper. Default 72.

/T:n Sets tab width to n characters.

/G:n Sets QG to n. Useful for typing files containing very long strings.

VIEW [filename] For viewing a file on a visual display. Pauses at the bottom of each screen; press space to get next screen. Will also search for strings. Type VIEW ? for list of switches.

ADMINISTRATIVE

BYE — Logs the user off the system.

CPUTIME — Prints the user's runtime

DAYTIME — Prints the date and time of day.

HELP — Prints a list of all EXEC commands.

JOB — Gives a summary of the state of each job.
JOB n — Gives summary for job number n only.
JOB . — Gives summary for your job only.
Options: any or all of JLAMSIRC to select subset and order of printout. J ? for further information.

KILL Jn — Forcibly logs out job number n.

KILL Kn — Forcibly logs out the job controlled by console number n.

METER — Prints accurate system load statistics.
Options:
T Totals since system started
M Mean in last 1-second period, exponentially weighted with 10-second time constant
C AIMS command utilisation (in some systems)

MOUNT unitname:structurename
MON: Makes structure accessible.
DISMOUNT structurename
MON: Makes structure inaccessible.

RESOURCES — Lists the resources available on the system. Devices, memory size, etc.

SCHEDULE — Prints current access restrictions from SS(1).
SCHEDULE N Sets normal access
SCHEDULE R Restricts access to departments<17

SEGMENTS — Gives position, size and state of all memory segments.
options:
S Name, origin and state in SID order
M MAP OF PHYSICAL MEMORY

SET — Allows on-line alteration of various system parameters. Options are:

```
SET ECHO *    - turns echoing on and off
SET ICASE *   - controls lower-to-upper case input conversion
SET OCASE *   - controls lower-to-upper case output conversion
SET PARITY *  - turns even-parity printout on and off
SET PAUSE *   - turns ^S/^Q pause mode on and off
SET IMAGE *   - turns line image mode on and off
SET SPEED     - selects baud rate
SET FILL      - sets number of filler characters
SET MEMMAX    - sets system-wide per-job memory limit
```

* denotes ON or OFF as appropriate.

SYSTAT — Gives a summary of the state of the system.

Options:

C	Memory utilisation
E	Error statistics
S	Swapping statistics
T	Run and up times
X	Extra information

Default: all except X

TELL [destination] [message]
Sends a one-line message to the specified place. [destination] may be Jn for job number n, or Kn for console number n.

UFDS [dev]: — Lists the master file directory giving the number of files and amount of space being used by each [dept user] number.

UNLOCK [filespec]
Unlocks the specified files. Files are 'locked' when they are in the process of being created. If the system is stopped due to a crash or operator intervention whilst a file is locked, the file will remain locked when the system is next started. This is a nuisance because locked files cannot be deleted or renamed. The UNLOCK command is provided to cure this problem. The DIRECTORY listing command will show if a file is locked.
Warning! Files must be unlocked as soon as the system is restarted. If other files are created when a locked file exists, and the locked file is then unlocked, the device directory will be corrupted. To minimise the risk of this happening, the SYSINI program scans the system disk for locked files and unlocks them everytime the system is started. But the problem may still arise on other disks in a multi-disk configuration. Because of the danger of unlocking files at the wrong time, the UNLOCK command is only available to privileged users.

VFIDIR — MON: Lists the volatile file directory which contains data about files currently or recently in use.

WHO — Shows who is currently using the system. Short form of JOBS.

EXAMPLE SESSION SHOWING SOME EXEC COMMANDS

```
MONITOR v1a AIMS v4h j7-k0
dept,user:16 17
password:

.help
commands are:-
0) run,load,execute,call,priv,bye,help
1) directory,unlock
2) copy
3) listlp
4) delete,ttyin
5) rename,type,view
6) tell,force,kill,broadcast,cputime,daytime
7) set,release,fcore,resources,schedule
8) free,map,ufds
9) obey
10) jobs,who
11) systat
12) segments
13) structures
14) vfidir
15) queue,submit
16) user
17) meter

.day
11 hrs 6 mins 20.6 secs on thursday 3-feb-77

.sys
monitor v1a, AIMS v4h

uptime 0:29:04, null time 0:16:32
= 43.1% run + 0.0% iowait + 56.9% idle.
mean uptime 14:31:31 in 1011 sessions.

total memory 64k(2048p)
monitor: 394p program + 71p fixed data + 313p buffers (6752=67%
 free)
available memory 40640(1270p),  28448(889p) occupied (70% full)
  memmax=20000(625p) vfimin=1536

swapping statistics:
total swap space 2384p, 2384p free
0 jobs swapped now, total segments swapped 0. rate limit 320p/sec.

.jobs
jb lin -area- --segment sizes in w--- status -runtime- irun -age--
 1 k12 040056 interp=6528 editor=4352 ti 1        04:23 .108     29
 2 p0  016016 interp=6528 watch = 800 sl 1z          01 .000     30
 3 p2  016016 interp=6528 batch =2720 sl 1z          01 .000     30
 4 k4  060060 interp=6528 idle  =2400 ti 1           03 .000     30
 5 p4  016016 interp=6528 lucifa=4608 to 2z       05:23 .122     15
 6 k14 050057 interp=6528 view  =2144 ti 1z          48 .026     08
 7 k0  016017 interp=6528 jobs  =2336 r4 1z          04 .032     03
```

```
.dir/f
dk0-016017-
diatyp.doc  pc.dmp      dsklst.bas  gorefs.bas  odt.bas
dkcopy.bas  dskrat.bas  maclst.bas  io.bas      pc.bas
fcomp.bas   ed.dmp      bchess.bin  im.bas      im.dmp
nikers.bin  imsys.bin   reseq.bas   draws.bin   cref.bas
s.mac       rantty.bas  numer8.bas  dskluk.bas  qtest.bas
tsubs.bas   dskrat.doc  ee.dmp      watch.doc   fscan.bas
milk.mac
 213 blocks in 31 files.

.dir /s[16 16]*.dmp

directory listing for dk0-016016 on 3-feb-77, block size=256
logout dmp    4c  <233>    9-apr-76 @4753
login  dmp   19c  <237>   29-oct-76 @4491
exec   dmp  107c  <233>    9-apr-76 @3937
batch  dmp   14c  <277>   14-aug-76 @4377
diatyp dmp   50c  <233>   27-oct-76 @3453
total of 194 blocks in 5 files.

.bye
11:15:34 3-feb-77 j6 k0 016017 AIMS library
run=0:00:05 connect=0:00:30 dk disk=216 bye
```

19. SOME PROGRAMMING TIPS

This section contains a mixture of general advice and specific programming 'tricks' which have been found useful.

Program Development

When programs are being developed one often gets several diferent versions of a program stored in different places under the same filename. So it is a good idea to make the program identify its version for the benefit of the user. It is also useful for the programmer if each program contains a REMark line indicating the date when it was last changed. Even with these precautions the maintenance of a large and changing suite of programs is still a major activity. It is helpful to keep a log book which records all modifications.

It is a good idea to establish a standard format for most programs. Efficiency considerations suggest that strings and arrays should be stored in low-numbered lines, see section 20. It is often confusing to have dollar-lines scattered throughout a program, especially if these are data strings that do not need to be saved with the program. Since every line is numbered it is often difficult to see where subroutines begin and end. A REMark line is helpful before each routine. It is easier to modify a program if it conforms to a standard line numbering scheme. For small programs it is convenient to use multiples of ten such as 100, 110, 120 etc. Modifications can then be inserted at 105, 115 and so on. For larger programs an interval of two is better, giving 100, 102, 104, 106 initially and modifications on odd lines. When the modifications have been checked out the program can be resequenced using RESEQ.BAS and a fresh listing obtained.

Program layout is to some extent a matter of taste, but the following scheme has been found satisfactory:

```
  0 -  9: Working dollar-lines, initially empty
  8 - 18: Arrays
       19 REM program version and date last modified
 20 - 80: Fixed dollar-lines, and space for small tables of data
          strings.
 80 - 99: Normally empty, useful for extra initialisation code
          when testing a program.
100 up:   Start of real program.
1000-1999: Reserved for library routines.
2000 on:  For large dollar-line tables.
```

Decoding User Commands

The first serious AIMS program that you write will most probably be one that reads a string from the terminal, recognises a set of 'command words', and goes to a specific routine for each valid command. Suppose the commands are BUY, SELL, NEWSTOCK, and DELETE. The commands should be chosen so that they are easily remembered by the user and suggest to him the function that the command performs. Experienced users will become irritated if they have to type long commands so we should choose command words that can be abbreviated to one or two letters without ambiguity, and the decoding program should be written to accept such abbreviations.

We might be tempted to start like this:

```
 20 $COMMAND:
100 INPUT ?20 $1
110 IF $1^'B' :GOTO 200 :REM BUY
120 IF $1^'S' :GOTO 300 :REM SELL
130 IF $1^'N' :GOTO 400 :REM NEWSTOCK
140 IF $1^'D' :GOTO 500 :REM DELETE
150 PRINT '?' :GOTO 100
```

This simple scheme has several disadvantages. Although it recognises abbreviated commands, it does not check the command word properly: it would accept BONGO as a command and interpret it as BUY. The program would not recognise a valid command if it had a few spaces in front of it. The decoding routine does not remove the command word from $1, so if there is any other information following the command this will have to be split off by each command routine. The program is also rather large and will increase by one line for each new command. Finally, the program offers no help to the user if he does not know what commands are available. It is good practice when implementing a complicated program to provide the user with a standard command which he can always give to elicit help from the program.

An improved version is given below. It begins by removing leading separator character's from the users input, and then strips off the next sequence of letters which is taken to be the command word. This leaves $1 containing any arguments that followed the command. The program has a list of all valid commands in $21 and it uses the QI facility of section 8 to validate the received command.

```
 19 REM V1 1-AUG-76
 20 $COMMAND:
 21 $,BUY,SELL,NEWSTOCK,DELETE,
 22 $B-UY STOCK, S-ELL STOCK, N-EWSTOCK, D-ELETE STOCK
100 INPUT ?20 $ :IF $_'?' :PRINT $22 :LOOP
102 PUT $=%F1>$%F4=%F1>$1
104 PUT ',$>$2 :IF $21_$2 :GOTO 100*(QI+1)
106 PRINT '? '$;'TYPE ? FOR HELP' :GOTO 100
```

In this example the program goes to line 200, 300, 400 etc. depending on the position of the recognised command word in $21 as given by QI. In more complicated applications it may be necessary to call a different overlay for each command. This could be done

by replacing line 104 with:

```
104 PUT ',$>$2 :IF $21_$2 :LET PT(4)=1024*QI :LOAD 1 #4 0,2
```

This loads the appropriate overlay and passes to it $1 which contains the argument string.

Unpacking Strings as Integers

For very intricate string manipulations it is sometimes useful to be able to access a string one character at a time. Although this can be done by using %G1 it is more efficient to place the string into an array and then use the arithmetic operations to access particular bytes.

To get the string into the array we use the packing facility of the LET commmand as described in section 14.

```
  8 ARRAY A 60
100 LET A()<$1#120
```

If $1 contains the string ABCDE these characters will be packed into the array like

```
A(2): 256*128+ E
A(1):  256*D + C
A(0):  256*B + A
```

The characters are packed two-per-word and the end of the string is marked with 128. All the characters are stored as integers in the range 1 to 127 according to the ASCII code shown in section 26.

The following subroutine may be used to obtain the J'th character of the string:

```
700 IF J&1 :LET C=A(J/2)_-8&255 :RETURN
702 LET C=A(J/2)&255 :RETURN
```

See also the discussion of byte unpacking on page 21.

Printing the Date and Time of Day

The date and time of day are often needed in headings. A nice format is

```
13:25 HRS WEDNESDAY 13-JUN-78
```

and this is returned in $0 by the routine

```
800 LET T=TI()/10
802 PUT T/3600@W2@F1':'QA/60@W2@F5' HRS '%SDA(3)+1$806'DAY'
DA()'-'%SDA(1)$808'-'DA(2)@W>$
804 RETURN
806 $,MON,TUES,WEDNES,THURS,FRI,SATUR,SUN
808 $,JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
```

This generator requires EP()>1. Line 800 takes about 5 milliseconds to execute. If you are using RX and not RN, the line may be reduced to

```
800 LET RX=(RA*RX+RC)&RM
```

giving faster execution. If you need to call the generator from several points in the program use "DO 800" since this is faster than making line 800 into a subroutine and using a GOSUB.

The random numbers returned by this routine are considerably more random than those obtained by playing with TI() or JS(4) or any other system data.

Square Roots

Square roots may be calculated by an iterative technique:

```
800 LET T=N*100 I=1 X=T/2 :UNLESS X :PRINT "N LESS THAN 2" :STOP
802 LET Y=(X+T/X)/2 :UNLESS X=Y :LET X=Y I=I+1 :LOOP
804 RETURN
```

The routine should be called with the number in N. It returns ten times the square root in X and also in Y. The factor of ten arises because the routine actually finds the square root of 100*N due to the *100 in line 800. Greater accuracy may be achieved by changing line 800 to multiply N by a larger even power of two.

The variable I counts the number of iterations actually performed. I is not part of the algorithm and references to I may be omitted if speed is important.

Inputting Octal Numbers

This can be done by inputting the number as if it were in decimal and then using string manipulation to strip off the digits one by one. The following routine removes the first number from the string in $1 and returns its octal value in the variable C:

```
800 LET C=0 T=0 :PUT $1=T>$1
802 PUT T@W>$2 :IF $2_"8"!$2_"9" :PRINT "ILLEGAL NUMBER" :STOP
804 PUT $2>$$G1>$2 :LET N()<$#1 :LET C=8*C+(N()&127)-48 :LOOP
806 RETURN
808 ARRAY N 0
```

The routine uses the array N to convert each digit to its corresponding ASCII code.

20. EXECUTION SPEED AND MEMORY OCCUPANCY

Since AIMS is interpretive, every character in a program line is scanned whenever that line is executed. When a line is coded it is compacted as far as possible by

1) Storing the line number as a 16-bit binary word
2) Removing the colon and all spaces before and after each command word
3) Storing each command word, whether abbreviated or typed in full, as a single special character.
4) Converting all numerical constants (eg: 123) to binary.
5) Converting all variable and function references to 2-byte internal forms.

All program lines are chained together in a forwards-linked list with the lowest line number at the head. Normal progression through a program involves stepping to the next line on this list. Any out-of-sequence reference to a line, such as a GOTO, GOSUB, DO, ARRAY or string reference, involves a search from the head of the list till the required line is found. Consequently lower numbered lines are found quicker.

When programming for speed:-

a) Use low line numbers for frequently used dollar-lines and arrays.
b) Avoid unnecessary spaces in program lines.
c) Avoid multiple references to the same array element when this could be assigned to a simple variable.
d) Do not work out the same expression several times when it could be done once and assigned to a variable.
e) Try and get small loops onto one line and use the LOOP command. This is quicker than a GOTO.
f) Use multiple commands on one line, rather than many commands on separate lines.
g) Avoid manipulating strings where an equivalent effect could be obtained by operating on numbers in arrays.
h) Do not use the PUT command where the IF command would do.
i) Use the smallest possible value of EP().

If the string expression on the lefthand side of a PUT command is anything other than a single dollar-line, a temporary copy of the value-string will be made whenever the command is executed. Thus the command

```
PUT "!"$1>$1
```

needs enough space for three copies of $1. PUT commands using long strings thus take a lot of time and space.

Every program line requires about 2+(N+1)/2 words of memory, where N is the number of characters in the line, counting command words as one character, and ignoring the line number. (This estimate is approximate due to the operations descibed at (4) and (5) above) Thus it takes less memory to put several commands on one line rather than giving them each a line to themselves.

Execution times

As a rough guide it may be estimated that AIMS arithmetic runs about 500 times slower than the best machine code. AIMS string handling and I/O operations take about the same time as they would if coded in machine code.

Taking the mean execution time for a mixture of typical arithmetic operations like LET X=X+1, LET X=Y*Z, LET X=Y/Z, the following operation-times are obtained:

AIMS version 2 on 11/20 cpu:	3.18 milliseconds
AIMS version 3 on 11/20 cpu:	1.95
AIMS version 2 on 11/45 cpu:	1.71
AIMS version 3 on 11/45 cpu:	1.08

GARBAGE COLLECTION

Whenever a line is altered it is recreated and the old line remains in memory as garbage. A garbage collection routine is provided which scans the whole program area, removes the unwanted lines, and creates a compact line-structure. This process can be invoked explicitly by the "GARB" command, but it is not normally necessary to do so. Whenever the interpretor advances to a new program line it checks the amount of free space that is available (as given by QS). If it is below a threshold given by the value of the system variable QG, then a garbage collection is performed. QG is initially set to 350 characters, which is sufficient for the PUT command operating on a string of normal length.

If AIMS runs out of space in the middle of a program line, it abandons execution of that line, does a garbage collection, and gives error S. The error is fatal in the sense that execution is abandoned at an indeterminate point in the line, and the result may be different if the line is re-executed from the beginning. Therefore, if you need to manipulate long strings of length N characters, the garbage threshold QG should be set to at least 3*(4+N). Alternatively, QS can be checked explicitly at an appropriate point in the program and a GARB command executed if it is too low.

Garbage collection is a very slow process which should be avoided if possible. Running with a large value of QG will cause frequent collections and slow the program down.

The VGARB command is used to delete unwanted user-defined simple variables. This deletes all user-defined variables that are not mentioned in the current program and recovers the space used. This command is useful for deleting unwanted variables that have collected during passage through a sequence of overlays. It has the advantage over RUN that it does not make all variables undefined.

CONTROL OF MEMORY USAGE

In a multi-user environment there is a central pool of free space and each user is allocated memory as he requires it. The allocation is performed in pages of 32 words (MON) or blocks of 128 words (DOS). At any given moment each user owns a particular number of blocks, as given by the system function UC(). This space is occupied as follows:-

a) About 150 words are used as workspace by the interpreter.
b) About 20 words for the system variables.
c) P+1 words for each P-word precision simple variable defined by the user.
d) About 2+(N+1)/2 words for each N-character program line
e) D+6 words for each array of dimension D
f) Some free space private to the job as given by QS.

We have already mentioned that a garbage collection is performed automatically if QS falls below QG. After such a collection QS is checked again, and if it is still below QG+100, an attempt is made to obtain more memory from the monitor. Thus the user's area is expanded automatically as necessary, and a working free space margin of at least QG characters is maintained. This process will proceed until all the available memory has been allocated to users. If a user's memory requirements continue to expand beyond this point he will be operating with QS<QG causing repeated garbage collections. In this extreme case the collection rate can be reduced by setting QG to a smaller value, but this will run the risk of a no-space-left error condition.

When an EXIT or LOAD command is executed, the users memory area is automatically reduced to an optimum size such that QS is about QG+100. Thus the system reclaims memory that is no longer needed by a user.

A special command is provided to allow explicit control of the user's memory area:-

CORE [ne]

This command adjusts the user's memory area such that QS is at least [ne] and not more than [ne]+256. The command will fail if there is insufficient memory. The command always succeeds if [ne] specifies a reduction in the user's area.

This command is useful in two cases:-

a) CORE QG returns surplus free space to the common pool.

b) CORE [ne] allows a user to grab space for future use.

Note: memory adjustment is an expensive operation that holds up other jobs on the system since it involves shuffling user memory areas. It is possible to minimise job memory areas by including lines like

GARB :CORE

in programs, but this should only be done after a careful analysis of the likely effects. A line of the above sort might cause two memory shuffles and these have to be weighed against the reduction in swapping that the smaller job size will give. Memory shuffles also hold up input/output operations. Note that the CLEAR command performs a GARB implicitly, so if you do a GARB after a CLEAR this will waste a lot of time.

It is neither necessary nor desirable to give CORE, CLEAR, or GARB commands before or after a LOAD.

DOS: the system function FC() gives the amount of memory left in the common pool. A negative value of FC() indicates that the pool is empty and that the system is running below its safety margin. This situation arises when FC()=0 and a number of users then perform operations that cause DOS to claim storage.

ERRORS DUE TO LACK OF MEMORY SPACE

Some AIMS commands may cause an error due to lack of memory space. There are two different errors that can happen: ?S and ?Y.

The ?S error means that the command cannot be performed for lack of free space within the user area (ie. QS is too small). It may still be possible to perform the command successfully by securing a larger value of QS. You may be able to achieve this without increasing the size of your user area, by doing a garbage collection. This can be done either with an explicit GARB command or by setting QG to a suitably large value. (Remember that the automatic garbage collection tries to keep QS>QG.) An explicit GARB is not recommended because the value of QS immediately afterwards depends upon the amount of garbage that happened to be around. This is usually indeterminate and there is thus no guarantee that the GARB will always release sufficient space. A suitably large value of QS may also be obtained by increasing the total size of the user area with a CORE command, thereby increasing both UC() and QS. Note that however large the area is made, garbage will still accumulate until QS gets near QG. So the command may succeed at first and then fail later when sufficient garbage has accumulated. You must either place the CORE command immediately before the command that needs the space, or you must set QG appropriately. If QG is not set appropriately, a GARB is desirable before the CORE.

The ?Y error means that the command cannot be completed for lack of memory in the entire system. It arises in three ways only: (1) attempt to CODE an array with insufficient QS and no scope for UC() expansion, (2) attempt to convert a number to a string of digits with the width setting (@W) too big, and (3) DOS: insufficient monitor buffer space when performing file operations. Errors (2) and (3) cannot be removed by changing QS or QG or by doing GARBs or COREs.

MEMORY REQUIREMENTS FOR FILE OPERATIONS

Every channel that is INITed requires 16 (MON) or 32 (DOS) words. This is reclaimed only when the channel is RELEASEd. DOS: Whenever a file is OPENed there is a transient requirement for at least 768 words. Once the file is open this requirement falls to zero if open in modes 4 through 6, or to the device buffer size if open in modes 0 or 1.

Device	Buffer size
DF	64 words
DP	512
others	256

This buffer space is reclaimed when the channel is CLOSEd.

DOS: DELETE and RENAME have transient requirements for 768 words. INITing a non-disk device requires 256 words for the device driver. This is not reclaimed until the channel is RELEASEd.

The READ and WRITE commands have transient requirements for data buffers if the transfer runs across device block boundaries.

21. COMMUNICATION BETWEEN DIFFERENT USERS

GLOBAL VARIABLES

There is a standard system function, GV(N), which accesses an array of global variables. These variables may be read by all users, and may be set by certain privileged users (see JS() function). There are at least 8 variables numbered from 0 to 7, of which 0 to 3 are reserved for use by AIMS executive programs. Multi-word references like GV(3,N) may be used as with arrays, see page 18.

These variables are useful for inter-job communication, since a value may be set into GV() by one job and later read by another job. GV() may also be used as a shared array to save space. For example if a number of jobs all require access to a large database via an index, it may be efficient to keep the top level of the index in GV(). This would require perhaps a 100 or 200 word global vector, which can be arranged when the AIMS system is configured.

LOCKS

Where a number of jobs require access to some shared facility, such as a common database, it is often necessary to control the accesses so that only one job is able to modify the data at a time. Otherwise unexpected results may be obtained when several jobs simultaneously update the same part of the database.

A global variable may be used to effect the necessary control if each job contains an appropriate sequence of commands. Suppose we set GV(n)=0 when the shared facility is idle, and we set it to 1 whenever a job is accessing the facility. Each job should then contain a procedure like:

1) Wait till GV(n)=0.
2) Set GV(n)=1.
3) Access the facility as required.
4) Set GV(n)=0.

This procedure ensures that only one job has access to the facility at a time, provided steps (1) and (2) are carried out as a single operation that cannot be interrupted by the execution of any other job. Steps (1) and (2) may be done as follows:

```
100 LET T=GV(N) GV(N)=1 :IF T=1 :WAIT 10 :LOOP
```

Because the examination and setting of GV(n) is done using a single AIMS command (the LET command above), the two operations cannot be interrupted. Control will remain on line 100 until some other job sets GV(n) to zero. GV(n) is then set back to 1 and control goes to the line following 100.

Rather than setting GV(n) to 1, it may be more informative to set it to the number of the job that is currently using the

facility. This may be done as follows:

```
100 LET GV(N)=GV(N)-J*(GV(N)=0) :UNLESS GV(N)=J :WAIT 10 :LOOP
```

In this case control will remain on line 100 until someone sets GV(N) to zero. Control then resumes on the next line with GV(N) set to the job number J. J may be obtained from JS(3), see section 22. Notice the use of a conditional expression (GV(N)=0) to test and set the lock in one command.

SIMULTANEOUS FILE UPDATES

Where several jobs are updating the same file it is essential to use a lock to control the accesses as described above. A further complication is caused by the use of the READ and WRITE commands. As noted in section 14, these commands allow any number of words to be transferred between any point in the file and an array. In fact, however, most real devices like disks and DECtape can only transfer data in blocks of 256 words. In addition, the transfers must begin at a device address that is a multiple of the block size. AIMS handles this problem automatically. If you give a READ or WRITE command with an array size that is not a multiple of the device block size, or with a device address (ie. PT() value) that is not such a multiple, AIMS will buffer the transfer. It does this by setting up a temporary memory buffer, reading a complete device block into the buffer, extracting (for a READ) or modifying (for a WRITE) the data in the buffer, and then writing the complete block back to the device (for a WRITE).

This means that a single READ command in your program may actually involve two separate reads from the device, and a single WRITE command may actually involve two reads and three writes to the device. These multiple transfers may be interleaved with computation or I/O done by other jobs.

These considerations should be borne in mind when designing the system of locks that controls access to a database. For example, in some cases it may be sufficient to have a lock that simply prevents the simultaneous update of the same record, so that one job can update record A whilst another job is updating record B. But if the two records happen to be stored in the same block on the device, the multiple reads and writes of the whole block may cause problems.

SENDING MESSAGES BETWEEN TERMINALS

A terminal is regarded as an input device and an output device. The input side of each terminal is identified by a console number, C, which is always even. The corresponding output device is identified by the number C+1, which is thus always odd. For

example, the first two terminals are denoted as

```
KB 0    input side of console 0
PT 1    output side of console 0
KB 2    input side of console 2
PT 3    output side of console 2
```

Users may send messages to one another by means of the PRINT command:-

```
>INIT #5 'PT 7' :PRINT #5 'HELLO TERMINAL 6' :PRINT 'OK
OK
>
```

Sends the message 'HELLO TERMINAL 6' to console 6.

Only privileged jobs may INIT terminals belonging to other users.

Short messages may be sent from one terminal to another by means of the EXEC TELL command, see section 18. This command is implemented in terms of INIT and PRINT commands as described above.

Extended WAIT command

There is an optional extension to the WAIT command which enables certain events to be detected at the earliest possible moment. The syntax is:

```
WAIT [time ne] [wake mask ne]
```

where [wake mask ne] is a bit mask specifying a set of events which are to wake the job. IE. the WAIT will be prematurely terminated if one of these events occurs.

```
 1  Controlling console activity
 2  WAKE command from other job
 4  reserved
 8  Completion of autodialling sequence
16  Slave job (on connected PC) requires service
```

Note that if [wake mask ne] is omitted, it will evaluate to zero and the WAIT command will function as usual.

Take note:

Events are remembered from one execution of a WAIT command to the next. A WAIT command will cause no delay if the wake mask matches an event which has occurred since the last WAIT command. Note that this happens even if the program has processed the event in the meantime.

Unwanted events that have been remembered may be cleared by executing a WAIT command with a zero wait time and an appropriate mask. Use a wait time of -1 if you want to wait indefinitely for an event.

When developing a program that uses the wake mask, remember that every carriage-return that you type will count as controlling console activity.

WAKE command

This command allows one job to wake another sleeping job. The syntax is:

WAKE [se]

The first six characters of [se] are assumed to be the name of a program. The WAKE command scans all jobs on the system and wakes those which are executing a program of that name. QA is set to the number of jobs that are woken.

The program name that is used in this context is that printed by the EXEC JOBS command. It is settable by the SETNAM command:

SETNAM [se]

Sets the program name of your job to the first 6 characters of [se].

Normally two jobs wishing to communicate using the WAKE command will agree on unique program names which they will publish using the SETNAM command.

Whilst job A is sleeping, job B may create a data file or alter a GV() value, and then WAKE job A. This avoids the need for job A to repeatedly look for events which may not have happened.

Note that the WAKE command simply sets a status bit associated with the named job(s). If such a job is WAITing with a [wake mask] containing 2, the setting of this bit terminates the waiting condition. If the job is not waiting the bit is still set but it has no immediate effect. When the job next executes a WAIT with mask 2 it will cause no delay because the bit is already set.

22. JOB STATUS INFORMATION

Associated with every job is a job status vector, accessed by the system function JS(N) as follows:-

JS(0) Right byte: Job privileges
JS(1) DOS department/user numbers for this job
JS(2) Argument set by LOGIN command (ignored by AIMS)
JS(3) Right byte: Number of console controlling this job
Left byte: 2*Job number
JS(2,4) Cumulative runtime for this job (1/50'ths sec)
JS(6) User's connect time in tens of seconds

*** entries above JS(6) are liable to change as the system ***
*** is developed. Although some executive programs refer to ***
*** these entries, ordinary programmers should not do so. ***

All these may be read by the user, but only JS(0) may be set.

There are several levels of privilege at which an AIMS job may run. JS(0) is a status word whose bits indicate different privileges:-

BIT	DEC	MEANING
0-1		Intrinsic job priority (00=highest, 11=lowest)
2		Reserved for future expansion
3	8	1 if job is logged in
4	16	0 if job allowed to set GV(n)
5	32	1 if STOP command and control-C are allowed
6	64	1 if control-O abort character is to be recognised
7	128	0 if program is an AIMS-executive (EXEC)
8-15		Reserved for system use

This status word is initially set to zero before a user logs onto the system. The user may set any bit of JS(0) by means of the LET command. For example

```
100 LET JS(0)=16
```

will set bit 4. However, the user cannot clear any bits of JS(0). This convention allows the user to reduce his privileges, but prevents him from increasing them. Mon: an unprivileged user can reduce his priority within the limits specified in JS(2).

Whenever the AIMS executive program EXEC is recalled, either as a result of an EXIT command, or of the control-O key, bits 0,1 and 4-7 of JS(0) are cleared, giving the program executive privileges.

EXEC PRIVILEGES

a) Unrestricted access to disk directory [16,16], which is used for system administration and accounting.

b) Capability of writing into the system status vector SS(N).

c) Allowed to set the right byte of JS(0) unconditionally.

d) Allowed to communicate with terminals owned by other users.

e) Allowed to execute the LOGIN and LOGOUT commands.

f) Allowed to examine and deposit (DOS) in real memory by means of the PK() function.

g) Allowed to bypass the file structure on directory devices and thus gain direct access to the storage medium.

h) Allowed to INIT any pseudo-console.

INTRINSIC JOB PRIORITY

Each job competes for central processor time with a priority that depends on the job's behaviour. For example, jobs coming out of keyboard wait-states are more likely to be run than jobs that are compute-bound. Apart from this dynamically changing job priority, every job has an 'intrinsic priority' which is determined by bits 0 and 1 of JS(0). These are set by the user or by EXEC to define four broad classes of job:-

0 Top-priority job pre-empts all others of lower priority.
1 Normal interactive job. (control-O and EXIT force this priority)
2 For low-priority interactive jobs, or fast through-put batch.
3 Lowest priority for background jobs. Will not run unless all higher-priority jobs are blocked.

AIMS executive programs run at priority 0 to ensure fast response. EXEC reduces its priority to 1 as soon as it runs. Users may further reduce their priority for background processing. Privileged users may raise their priority to 0 for fast overall response at the expense of others.

Note: be careful when changing these bits of JS(0) to preserve the remaining bits which determine your privileges. The priority bits should be set using the inclusive-OR operator (!) and cleared using the AND (&) or exclusive-OR (\) operators.

23. SYSTEM ADMINISTRATION

Some of the facilities described in sections 23 and 24 are not an integral part of the AIMS language and are subject to change. They are provided by programs written in the AIMS language, and these programs are normally tailored to meet the needs of each installation.

The AIMS system essentially provides the capability of running a number of AIMS programs at the same time. Each such program is associated with and controlled by a particular console. The AIMS system itself is not concerned with the identity or legality of the user, or with the demands he makes upon system resources. User identification, access restriction, and accounting are functions performed by privileged programs written in AIMS. These are referred to as AIMS executive programs, and they may be modified by the system administrator to implement any desired resource management policy.

There are three executive programs that are essential to the operation of the system:-

LOGIN This is run automatically when a user connects to the system. It performs the screening functions associated with 'logging in'.

EXEC Provides the AIMS 'monitor' as seen by the user once he is logged in. Control goes to EXEC when the control-O key is pressed, or if the EXIT command is executed.

LOGOUT This is run when the BYE command is given. It performs the function of 'logging off' a user. This may involve updating an accounting file and perhaps ensuring that the user is not occupying more than his allotted disk space.

These programs are privileged (see section 22) and cannot be interfered with by the user.

When LOGIN is run the only clue to the user's identity is his console number, which gives the position of his terminal. Some terminals may be situated in highly secure areas, in which case no further screening need be done. In most cases however the user should be asked to identify himself by giving a secret password. LOGIN can validate this by looking it up in a file of permitted users. This file may also contain information about the resources and privileges allowed to the user. LOGIN may then set the job status word JS(0) appropriately before passing control to the user.

Once a user has been admitted to the system by LOGIN, subsequent system behaviour may be made dependent on the user's identity. For example, a particular applications program may be run automatically for a class of users.

LOGIN AND LOGOUT COMMANDS

These commands are only available to EXEC-privileged programs.

LOGIN [ne1] [ne2]

Sets JS(1) to the value of [ne2] which must be a valid DOS department/user number. Also notifies DOS that the job is now running under department/user number [ne2]. Sets JS(2) to the value of [ne1]. JS(2) is ignored by AIMS and may thus be used for any purpose. The present executive programs use JS(2) to hold a unique internal user code that is set by LOGIN and referenced by LOGOUT. Control resumes on the same program line.

LOGOUT

Kills the job and returns its memory to the common pool. All I/O channels must be released by the program before executing a logout command. Control never returns.

BYE [ne]

Forces job number [ne] to execute the system LOGOUT program.

SYSCOM command (MON only)

The SYSCOM command enables a user program to exchange information with the monitor in the form of strings. It permits the reading and setting of certain monitor data items without requiring knowlege of the internal monitor organisation.

SYSCOM [command se]>$[reply ne]

[command se] specifies the function required, and the monitor returns a reply string in $[reply ne] if present. The following functions are implemented:

?	Returns a list of all SYSCOM functions
DEVTRA	Returns the system device translation table
STRUCTURES	Returns a list of all structures currently MOUNTed
UNITS	Returns a list of all device units that exist

Other functions may be implemented from time to time.

SYSTEM STATUS VECTOR

A 50-word system status vector accessed by the function SS(N) is provided. It can be read by all users but can only be written by executive programs. These status words are used to control the way in which the system is used:-

SS(0) Maximum amount of memory that a job may use (words/32)
SS(1) Access control word (see below)
SS(2) Interval (in 1/50'ths sec) between scheduling decisions
SS(2,3) System null time in 1/50'ths sec (2-word quantity)
SS(2,5) System lost time in 1/50'ths sec (2-word quantity)
SS(7) Address of mode 6 fast access directory, if any
SS(8) Name of default user filing disk in radix50
SS(9) System feature switch, see below
SS(10) 256*Version + Mark number of system
SS(11) Name of system disk in radix50

*** Entries above SS(11) are liable to change as the system ***
*** is developed. Although some system programs refer to ***
*** these entries, ordinary programmers should not do so. ***

SS(12) Address of job table (for use by PK() function)
SS(13) Number of job slots in job table
SS(14) Address of console table
SS(15) Number of highest console + 2
SS(16) Number of highest pseudo-console +2
SS(17) Number of terminal designated as 'system console'
SS(20) Number of priority queues in cpu scheduler.
SS(20+n) For n=1 to SS(20). Time quantum in 1/50'ths sec for jobs in run priority queue N.
SS(28) Total allocated swapping space in 1K-word blocks
SS(29) Amount of free swapping space in 1K-word blocks
SS(30) Amount of occupied swapping space in 128-word blocks
SS(32) Total number of swap transfers done, both in and out
SS(33) Instantaneous number of jobs swapped out
SS(34) Number of hardware errors during swap transfers
SS(35) Number of jobs aborted due to irrecoverable swapping hardware errors
SS(38) Number of software checksum errors on swap-in
SS(39+n) For n=1 to SS(20). Resident protect time for jobs in run priority queue n.
SS(46) Number of fair-schedules done
SS(47) Seconds until fair scheduling acts

Note: SS() words not mentioned above are reserved and may actually be in use for internal purposes. Words above SS(20) depend on options that may not be present in your configuration.

SS(0) influences the behaviour of the AIMS monitor when servicing requests for more memory. SS(1) is used by the login program, LOGIN. Depending on the value of SS(1), access to the system may be restricted to local users, or further logins may be prevented entirely. This allows the system to offer a variety of reduced services whilst it is doing resource-critical background processing.

SYSTEM FEATURE SWITCH

SS(9) is a bit mask indicating the presence of various optional features which may be included in an AIMS system when it is configured. Each bit is set to 1 to indicate presence of the feature:

8	Swapping system
16	Multi-level priority scheduler
32	Crash dump
64	Pseudo-consoles
128	reserved
256	Extended performance metering
512	DOS: Mode 6 fast access directory
1024	Fair-scheduling
2048	WAKE command
4096	System supports segments

FORMAT OF ACCESS CONTROL WORD SS(1)

SS(1) may be set by the system manager to limit access to various classes of users. It is bit-coded as follows:-

BIT	MEANING WHEN SET TO 1
1	System is being initialised, no logins allowed.
2	System is initialised. (I.E. DA() and TI() are set etc.)
4	Restrict all logins to people with department numbers < 17.
8	Allow logins from local terminals.
16	Allow logins from remote (I.E. modem) terminals.
32	Do not ask for password if job is being logged-in by a pseudo-console (I.E. if job is slaved).
64	Allow logins from pseudo-consoles.
128	Do not ask for password if user is logging-in with a department number greater than 16. This effectively removes password security from non-system departments.

When the system is started, bit 1 is zero. This causes the LOGIN program to run the system initialisation program SYSINI which converses with the operator and initialises the system. SYSINI then sets bits 1 and 2 and returns control to LOGIN. The remaining bits of SS(1) are set by SYSINI as specified by the operator. They may also be set by other privileged programs at any time. The default value for SS(1) is 132 octal, which is 64!16!8!2. SS(1) may be examined and changed by means of the executive SCHEDULE command.

PASSWORDS

The DOS file structure provides for partitioned directories accessed via the job's department/user number. It does not provide password security or accounting. These facilities are implemented by the AIMS login and logout programs using a file 'USERS.SYS' which contains the department/user number and password of every user known to the system.

DEPARTMENT/USER NUMBERS

Each new user should be allocated a department and a user number which must be entered into the master file directories of every device that the user is permitted to access. This is done using the /EN switch in PIP or the USER.DMP utility.

Two disk areas are treated specially by AIMS:-

[16 16] System administration area. This area can be accessed freely by AIMS executive programs, and cannot be accessed at all by any other program. It is used to hold files of passwords and accounting information.

[16 17] AIMS library. EXEC searches this area by default if a requested program cannot be found in the user's own file directory.

Note that all references to department/user numbers are in decimal, whereas DOS uses octal numbers.

LOGIN PROGRAM - LOGIN

When a user connects to the system a new job is created and the AIMS login program, LOGIN, is run. At this stage the job is running under the AIMS system account [16 16]. This does not imply any loss of security since LOGIN is privileged and cannot be stopped by the user.

LOGIN begins by examining the state of the system using the FC(), SS(), and PK() functions. If LOGIN decides that the user should not be allowed to use the system, it may print a message to this effect and kill the job by executing the LOGOUT command. Alternatively, if the system can accept another job, LOGIN will ask the user for his department/user number and password, and validate these by reference to the file 'USERS.SYS'. If all is well, LOGIN will log the user in by executing the LOGIN command, which also changes the department/user number to the appropriate value. LOGIN now examines words 21 to 25 of the user's record in USERS.SYS to see if they contain a filename and department/user specification. If so, LOGIN opens the specified .DMP program on channel 4, and LOADs and runs the first overlay of it. If not, LOGIN transfers control to EXEC via the EXIT command.

FORMAT OF USERS.SYS FILE

The file begins with an 8-word header as follows:-

0: file creation date
1: number of entries
2: base address of first user record
3: number of words per user record
4: base address for storing system-wide memo string
5 to 7 spare

This is followed by a table of 1-word entries giving the valid department/user numbers. Unoccupied entries contain zero. The relative position of an entry in this table is used as an internal code identifying the user.

Following the department/user table there is a table of fixed-length records, one record for each user. Each record contains password and accounting information for the user. This table begins at the file address (ie. PT() value) given in Word2 of the header, and the size of the records is given in Word3 of the header. The PT() value of the record for user number U is thus given by Word2 + U*Word3, where U is the internal code derived from the department/user table.

This scheme allows the size of the tables or records to be altered without affecting existing administrative programs. The internal code number, U, is computed by LOGIN and entered into the left half of JS(2) by the LOGIN command. Once a user is logged in his record may thus be accessed directly by reference to JS(2). In particular, the logout program uses JS(2) to update accounting information in the user's record.

Each user record is currently 32 words long and has the following format:-

0: Upto 7-character password
4: Bit-mask indicating user's privileges
5: Cumulative connect time in seconds
7: Cumulative run time in 1/50ths of a second
9: Date when user last logged off
10: Time when user last logged off in tenths of a second
12: Upto 15-character name of user
20: Number of times user has logged in
21: Upto 6-character name of program for auto-start.
25: Department/user number of auto-start program.
26: Logout quota on default disk (-1=infinity)
28: Default file protection
29 to 31 spare

Words 21 to 25 are used by the LOGIN program to cause a user-specified program to be run automatically whenever someone logs in under a particular [p,p] number. The specified file is assumed to be in DUMP format with a file extension of .DMP. If no program is specified (detected by word 25 being zero), LOGIN will run the EXEC program.

The privilege bit-mask is identical in format to JS(0). This 8-bit mask is stored in the right-half of JS(2) by the login program. EXEC inclusive-OR's the mask into JS(0) whenever it relinquishes control to the user.

Word4 of the header indicates the address of the system memo string area. The program SYSMEM.BAS allows a short string to be stored in this area. The login program prints this string if the user has not logged in since the message was stored. This is a simple way of notifying all users.

ACCOUNTING OPERATIONS

Use of the system is monitored chiefly by the login and logout programs.

The login progam sets words 9 to 11 of the user's record to the date and time of login, and increments word 20.

During the running of a job the runtime is accumulated by AIMS in JS(4) and JS(5), and the connect time in JS(6).

The logout program updates the user's record by:

a) Adding the runtime from JS(4-5) to words 7-8.
b) Adding the connect time from JS(6) to words 5-6.
c) Resetting words 9-11 to the current time and date.

LOGOUT PROGRAM - LOGOUT

The logout program, LOGOUT, is called in response to the BYE command. Apart from the accounting operations noted above it performs the following functions:

a) Releases all I/O channels
b) Kills the job by executing the LOGOUT command

If desired the logout program may perform other functions, such as writing a transaction file, or deleting temporary files from the user's disk area.

PERFORMANCE MONITORING - WATCH OPTION

There is a system data file called LASTUP.SYS which is used to store performance data relating to the current running of the AIMS system. This file is initialised by SYSINI when the system is started up. SYSINI uses this file to store the date and time when the system was started, thus enabling the SYSTAT program to compute the system uptime. There is a standard executive program called WATCH which is logged-in on a pseudo-console by SYSINI, and which normally runs until the system is taken down. WATCH wakes itself up every ten minutes and stores system performance data in LASTUP.SYS.

WATCH is also used to perform long-term time-dependent functions on behalf of other executive programs. For example, WATCH wakes up at midnight and updates the DA() and TI() functions. WATCH may also be instructed to start the BATCH controller at a specified time of day, thus providing a completely automatic batch processing facility (see section 24). Customers may make their own additions to WATCH to implement other time-dependent functions as required.

MON: WATCH is also used to log hardware errors and to deal with user/operator communication if a device goes down. For example, if the lineprinter runs out of paper WATCH will print the message

[Problem with device LPA0]

on the user's terminal and the operating console.

24. CONTROL FILES AND BATCH PROCESSING

A job is normally controlled by a user typing commands at a console, but there are some situations in which this interactive mode of working is inconvenient:-

a) Since each job requires one console, the job capacity of a system is limited to the number of consoles in the installation. For small installations this may be unduly restrictive, causing reduced utilisation of the other hardware components.

b) If the commands that are needed to run a job are known completely in advance, it would be more convenient to place them in a file, and get the system to execute this file as if the commands had been typed at a console. This is especially useful if the command sequence is at all lengthy or complex. A typical example of this is the command sequence required to CALL and DUMP a large number of program overlays into a dump-format file.

c) Some jobs may perform a large amount of input/output or computation, causing them to take several hours to complete. It is obviously tedious for an operator to remain present for the duration of the job, simply in order to type in the occasional command.

d) Routine maintenance procedures, such as file updates, disk dumps, and so on are best done by some automatic procedure so as to minimise operational errors.

To cater for these situations, AIMS provides a facility which allows one job to send commands to another job. The job which sends the commands is called the master job, and the other one is called the slave job. The slave job receives the commands exactly as if they had been typed by someone at a console, and in fact the slave job need not know whether it is being controlled by a person or by a master job.

This facility is implemented in AIMS by means of a special device called a 'pseudo-console', which is emulated by the AIMS system software. A master job may INIT a pseudo-console, and may then send messages to it by means of the PRINT command. To the slave job the pseudo-console looks like an ordinary console, and it receives the commands on channel 2 in the normal way. Similarly, if the slave job generates any printout, it is sent to the pseudo-console via channel 1 as usual. The master job may read the slave job's printout by INPUTting strings from the pseudo-console. Thus the master job is able to 'type' commands to the slaved job, and to 'read' the printout from the slaved job, just like a person sitting at a real console.

The pseudo-console facility is provided in a fairly crude form, and the mechanics of its use require a fair knowledge of the internal structure of AIMS. These details are described later. We now describe the two standard executive programs that are provided to cater for normal user requirements.

OBEY command

OBEY is one of the commands available under EXEC. The syntax is

OBEY [filename]

where the file is assumed to have an extension of .CTL.

This command causes the specified file to be executed by a slave job. The printout from the slave job is displayed on the console where the OBEY command was typed. When you give an OBEY command, your current job becomes a master job. The master job finds a free pseudo-console, logs it in, reads your control file, sends the commands one at a time to the slave job, reads the slave printout, and prints it on your console. When the end of the control file is reached, the slave job is logged out, the master job releases the pseudo-console, and you are returned to EXEC.

The OBEY command thus requires two job slots and the use of your console, so it does not save any resources. It is mainly useful for saving time and typing when it is required to execute a fixed sequence of commands online. For example, you can keep a library of standard control files that build program overlays, list all the programs in a particular suite, or copy them from disk to magnetic tape or vice versa. The OBEY command has the additional advantage that it provides a printout in a standard format, with the date and time of day, showing exactly what has been done. This is invaluable for the routine maintenance of complex application suites, since the printouts may be filed away and referenced later if some problem occurs.

Unlike most EXEC facilities, the OBEY command cannot be aborted with control-O, since this would leave the slave job still running. OBEY may be aborted by typing carriage-return which causes EXEC to logout the slave job.

BATCH processing

The batch processing option allows complete jobs to be run automatically without the use of a real console. A user may create a control file containing any sequence of EXEC, AIMS or user-level commands. He may then use the executive SUBMIT command to enter his control file into a queue of jobs waiting to be run. Later on this queue will be read by a standard executive program called BATCH. BATCH reads each queue entry in turn and executes the control files using a pseudo-console and slave job. The printout from the slave job is returned to the user's disk area as a log file which is written by BATCH.

The master job controller, called BATCH, is itself normally run on a pseudo-console, so that the whole process does not require the use of any real consoles. When compared with interactive use, the batch process has the disadvantage of requiring an extra job slot for the BATCH controller. However, since BATCH can run two slave jobs concurrently, this overhead is not too severe. With

large AIMS installations it is normal to leave the BATCH controller running all the time on a detached pseudo-console. The controller wakes up when necessary and scans the queue for newly submitted jobs. For smaller installations where memory space is at a premium it is more usual to run the BATCH controller only during off-peak hours such as overnight.

The batch processing option provides some extra EXEC commands as follows:-

.QUEUE LIST

Lists the batch queue showing what jobs are waiting for batch processing.

.SUBMIT [filename]

Enters the specified control file into the batch queue.

A number of switches may be used with the SUBMIT command to specify options:

%AFTER:hrs:mins	Do not run until after the specified time. Default is run as soon as possible.
%RUN:hrs:mins	Abort job if runtime exceeds this limit. Default is 10 minutes.

Example

```
SUBMIT FRED%A:20:30 %R:5:0
```

Submits the file called FRED.CTL, with a runtime limit of five hours, not to be run before 8:30 in the evening.

.QUEUE KILL

Deletes from the queue the last entry that you made. Used to cancel an erroneous SUBMIT command.

.QUEUE KILL #n

Deletes your entry number n from the queue.

.KILL Pn

Aborts the job running on pseudo-console number n. Used to kill a job that has already been started by BATCH.

Control files

Both the OBEY command and the batch processing option expect a control file in the following format:-

```
dep usr          [First line is department/user numbers only
......
......           [body of control file
......
B                [Last line must be a BYE command
```

The control file should have an extension of .CTL. The first line of the file must contain the department/user number under which the job is to be run. The password need not normally be given (although this is an option controlled by SS(1)). The following lines may contain anything, provided this makes sense to whatever program is being run. Normally the second line of the file should contain some EXEC command, since control goes to EXEC after login. The last line of the file should be a BYE command.

Note that the control and log files are to be found under the department/user number of the job that did the SUBMIT command. This number need not be the same as the number under which the batch job runs.

Control-O and control-C may be included in a control file by using the syntax ^O and ^C.

It is sometimes useful to be able to send commands directly to the batch controller during the running of a slave job. This is done by means of a line in the control file that begins with a percent sign. By convention any control line beginning with % is interpreted as a command to BATCH and is not sent to the slave job. The following BATCH commands are implemented:-

%NOTIME Suppresses the time-of-day, which is normally printed by BATCH at the left margin of all lines in the log file. In this mode the log file contains only what would appear on a real console.

%TIME Restores the time-of-day printout.

%NOLOG Suppresses all log file output.

%LOG Restores log file output.

%PRIORITY:n Sets the intrinsic job priority of the slave job to n (where n is between 1 and 3). Default is 1.

These batch commands may be placed anywhere in the control file after the first line. The %PRIORITY command is only effective if it is read by the batch controller at a time when the slave job is executing a user program, so it should normally be placed after a RUN or EXECUTE command.

Log files

BATCH creates a log file under the department/user number of the job that submitted the request. The file has the same name as the control file, and an extension of .LOG. Any previous file with this name will be deleted by BATCH.

EXAMPLE USE OF OBEY COMMAND

This example shows a control file for copying a set of programs from DECtape onto disk. The control file contains the following text:-

```
16 16
CO =DT:NEX.BAS/U
CO =DT:DIRECT.BAS/U
CO =DT:COPY.BAS/U
CO =DT:PIP.BAS/U
CO =DT:SYSTAT.BAS/U
CO =DT:JOBS.BAS/U
CO =DT:MAP.BAS/U
CO =DT:OBEY.BAS/U
CO =DT:SET.BAS/U
CO =DT:QUEUE.BAS/U
B
```

When this file is obeyed we get the following printout at the terminal:-

```
.OBEY GETNEX
15:10:54   PC 6 INITIALISED AS JOB 4
15:10:54   AIMS V2C
15:10:54   J4-P6
15:10:55   DEPT,USER:16 16
15:11:03
15:11:03   .CO =DT:NEX.BAS/U
15:11:35
15:11:35   .CO =DT:DIRECT.BAS/U
15:12:30
15:12:34   .CO =DT:COPY.BAS/U
15:13:37
15:13:38   .CO =DT:PIP.BAS/U
15:14:01
15:14:03   .CO =DT:SYSTAT.BAS/U
15:14:28
15:14:28   .CO =DT:JOBS.BAS/U
15:15:04
15:15:05   .CO =DT:MAP.BAS/U
15:15:26
15:15:26   .CO =DT:OBEY.BAS/U
15:16:04
15:16:09   .CO =DT:SET.BAS/U
15:16:30
15:16:31   .CO =DT:QUEUE.BAS/U
15:17:05
15:17:05   .B
```

```
15:17:08   RUN TIME 1 MINS 35.9 SECS
15:17:08   CONNECT TIME 6 MINS 10.0 SECS
15:17:08   BYE
```

Batch and Obey examples

```
AIMS V2C
J1-K2
DEPT,USER:16 16
PASSWORD:

.DIR *.CTL
DF 0:[16,16]
 23-MAR     1  FRED.CTL
TOTAL OF 1 BLOCKS IN 1 FILES.

.COPY PT3:=FRED.CTL          [This is a very simple control file
16 17
DA
J
B

.OBEY FRED                   [Which we now OBEY
14:53:15   PC 6 INITIALISED AS JOB 7
14:53:15   AIMS V2C
14:53:16   J7-P6
14:53:16   DEPT,USER:16 17
14:53:24
14:53:25   .DA
14:53:26   14 HRS 53 MINS 25.7 SECS ON THURSDAY 29-MAR-73
14:53:33
14:53:33   .J
14:53:35   JOB LINE  AREA  PROGRAM SIZE ST P   RUNTIME CONNECT TIME
14:53:46    1   K2  16,16  OBEY   1,152 SL 1        05     01:20
14:53:47    2   P0  16,16  WATCH    512 SL 1        05  71:25:40
14:54:00    3   P2  16,16  BATCH  2,048 SL 1   2:15:03  71:25:20
14:54:02    4   K4  16,16  LOGIN  1,152 TI 0        00     21:30
14:54:12    5   K0  40,41  EXEC     768 TI 1     01:06     37:40
14:54:13    6   P4  40,41  TMP1   3,328 R4 1     18:59     20:50
14:54:24    7   P6  16,17  JOBS   1,408 R3 1        02     01:00
14:54:24
14:54:33   .B
14:54:36   RUN TIME 3.0 SECS
14:54:36   CONNECT TIME 1 MINS 20.0 SECS
14:54:37   BYE
```

Looking at the JOBS printout above, you can see the master job running the OBEY program, and the slave job number 7 running on pseudo-console 6.

You can also see the batch controller, which is sleeping (job 3), and the job that it is currently running (job 6).

```
.QUEUE LIST
FROM 14:32 HRS 29-MAR-73, FREE=500
[ 40, 41] #1   SUBMIT TRIAL %RUN:4:0  (IN PROGRESS)
```

```
.SUBMIT FRED        [Example of queueing a batch request
#1 FRED QUEUED

.Q L
FROM 14:32 HRS 29-MAR-73, FREE=492
[ 40, 41] #1    SUBMIT TRIAL %RUN:4:0  (IN PROGRESS)
[ 16, 16] #1    SUBMIT FRED %RUN:0:10

.Q K                [Delete submitted request from queue
FRED KILLED

.WHO
J1-WHO-K2 J2-WATCH-P0 J3-BATCH-P2 J4-LOGIN-K4 J5-EXEC-K0 J6-TMP1-P4

.KILL J4            [Kill a job directly
JOB 4-LOGIN-[16,16] KILLED, K4 DETACHED.

.W
J1-WHO-K2 J2-WATCH-P0 J3-BATCH-P2 J5-EXEC-K0 J6-TMP1-P4

.W                  [Wonder if batch job has finished
J1-WHO-K2 J2-WATCH-P0 J3-BATCH-P2 J5-EXEC-P2 J6-LOGOUT-P4

.W                  [Yes, it is just logging out now
J1-WHO-K2 J2-WATCH-P0 J3-BATCH-P2 J5-EXEC-P2

.Q L                [It should have gone from queue
QUEUE EMPTY

.CO PT3:=TRIAL.CTL[40,41
40,41               [This was his control file
E TMP1
CONTRO
S'DUMP
BYE
                    [Examine his log file
.CO PT3:=TRIAL.LOG[40,41
14:33:13   29-MAR-73  AIMS V2C  BATCH V1D
14:33:14   PC 4 INITIALISED AS JOB 6
14:33:17   AIMS V2C
14:33:17   J6-P4
14:33:18   DEPT,USER:40,41
14:33:19
14:33:19   .E TMP1
14:33:21   FILE:CONTRO
15:25:01   >S'DUMP
15:25:08   >BYE
15:25:10   RUN TIME 45 MINS 37.3 SECS
15:25:10   CONNECT TIME 52 MINS 0.0 SECS
15:25:10   BYE

.B                  [Now we log out
RUN TIME 24.3 SECS
CONNECT TIME 10 MINS 20.0 SECS
BYE
```

Driving pseudo-consoles - Master jobs

A master job connects itself to a pseudo-console by INITing it on a particular I/O channel. A privileged job may INIT any pseudo-console, even if it is already in use by another master job. An unprivileged job can only INIT a pseudo-console if it is either

a) Not attached to a slave job, or

b) Attached to a slave job that is logged in under the same department/user number as the master job, provided the slave job is not potentially privileged.

These restrictions are designed to prevent unprivileged users from interfering with privileged system jobs.

Normally a master job will want to create a new slave job and will therefore need to find a pseudo-console that is free. MON: The special devicename PCX is translated by the monitor into the name of the lowest-numbered free PC. The command INIT #c 'PCX:' will connect channel c to a free PC and the number of the PC thus obtained my be found from QX(c,7), see page 65. DOS: There is no convenient way of finding a free PC. The method involves peeking at the KTAB system table, see the utility program PCDEMO.BAS.

Once the channel is INITed, the master job may PRINT and INPUT messages in the normal way. PC's are special in that input and output may be done on the same channel number.

The INPUT command will extract characters from the slave job's console output buffer until a line terminator is found or the buffer becomes empty. The string thus obtained is given to the master job. Note that this string may not be a complete line (the slave job may be compute-bound at the time), and also that the INPUT command never suspends the master job. Hence the master job should contain a routine that repeatedly INPUTs from the PC until a null string is returned. The routine can assemble the slave job's output into proper lines by scanning it for [cr/lf]. It is also necessary for this routine to detect when the slave job is waiting for console input, since this often happens when half a line has been printed (eg. the slave job's input prompt string). The routine must also notice if the slave job logs itself out, so that the master does not wait indefinitely for output that will never be generated. On the other hand the routine should be sure to collect the printout from the LOGOUT program, which may appear after the job has logged out.

For these reasons the master routine that obtains output from the slave job is somewhat complex. A suitable routine is given below:-

```
 800 REM GET LINE FROM SLAVE JOB
 805 PUT $7>$1$6$6>$7 :RETURN 4
 810 INPUT #5 $ :UNLESS $='' :PUT $7$>$7 :GOTO 805

MON:
 815  if qx(5,4)=9 :put $7>$1 :put >$7 :return 2
 820 if qx(5,4) :return 3
```

```
 825 return 1

DOS:
 815 unless pk(jt+j-1) :unless pk(kp+10) :return 1
 820 if (pk(jd+14)&255)=238 :put $7>$1 :put >$7 :return 2
 825 return 3
```

where

```
#5     is PC channel
$6     contains [cr/lf]
$7     is buffer used for building up a full line
$1     is full line returned by routine
DOS only:
JT     is address of job table
J      is the number of the slave job
JD     is address of slave job's JOBDAT block
KP     is address of PC's entry in KTAB
```

The routine has four returns:

```
1  Job has logged out, there is no more prinout.
2  Job is waiting for input, $1 is remainder of printout.
3  Job is busy, no full line available yet.
4  Full line of printout in $1
```

The master job sends characters to the slave job by PRINTing them to the PC channel. These characters are sent through CONSER just as if they originated from a real console. The master job should normally end each line with %C13, rather than [cr/lf], because CONSER will add a linefeed. The lines may or may not be echoed depending on whether or not the slave job has suppressed echoing. The master job should not send characters to the slave job unless the slave job is waiting for input, otherwise the correspondence between input and output will be upset. This correspondence is maintained automatically provided the master input routine is always called after each PRINT to the PC.

Since the PC-driving procedure is rather complex and is affected by internal changes to AIMS, it is strongly recommended that programmers should use the standard subroutine given above.

25. LINE COMMUNICATION FACILITIES

The standard AIMS system will support any mixture of asynchronous line interfaces operating at speeds upto 9600 Baud for output and 2400 Baud for input. (eg. KL, DL, DC, LC, DJ, DH, DZ). Line characteristics such as speed, parity, echoing, filling, and the interpretation of control characters, may be varied under program control to suit the needs of each individual terminal. These facilities are primarily intended for interactive terminals such as Teletypes, DECwriters, and visual displays, which communicate with people. Extra facilities are needed if a line is to be used for other purposes, such as computer-to-computer communication or the control of special-purpose teletype-compatible devices. A number of options are available as follows:-

1) Synchronous Line Option

Provides support for synchronous interfaces operating at speeds upto 2400 Baud (eg DP11, DU11).

2) Image Mode Option

Allows any line to be treated as an 8-bit wide input/output device with no character interpretation. This permits the control of special-purpose devices like filmstrip projectors and cassette recorders, where all eight bits may be used for data transmission.

3) Specialised Communication Protocols

Special facilities are available for connecting an AIMS system to some well known message switching networks, such as S.W.I.F.T. and C.H.I.P.S. Other facilities can be provided on request.

These options are described in more detail below.

Line Modes

Each communication line is capable of operating in two modes:-

1) Normal Mode

Most lines operate in this mode. Data is assumed to be ASCII, parity is stripped on input and may be generated on output. Characters like carriage-return, null, rubout, control-Y, control-C, and control-O are treated specially.

Terminals operating in normal mode may be either attached or detached. Typing carriage-return on a detached terminal creates a new job and causes the LOGIN program to be run. The terminal is then said to be attached to that job, and the characters control-O and control-C affect job execution. The terminal becomes detached when the user logs out.

2) Image Input Mode

Received bytes are assumed to be eight bits wide and no special formatting or interpretation takes place. It is normal to use image mode when a terminal is detached. If an attached terminal is placed in image mode, the terminal loses control of the job until normal mode is restored. The PRINT, INPUT and ACCEPT commands may be used to communicate with an image mode line, but the LINE command is recommended.

The LINE command

The LINE command provides a means of changing line characteristics, and of sending and receiving data in 8-bit bytes or packets. The command controls a line directly without the use of an I/O channel. The syntax is

LINE [line number ne] [function ne] [optional argument]

where

[line number ne]	is console line number (not channel number)
[function ne]	specifies the function to be performed
[argument]	depends on function.

The function codes are:

0 Input of 1 byte to QA. If the line is in image mode, the full eight bits will be obtained.
1 Image-mode output of low-order 8 bits of [argument ne]
2 Read line characteristics into QA, set line characteristics from [argument ne].
3 Set modem status from [argument ne], read latest modem status into QA.
4 Set line speed (Baud rates).
5 Reserved.

These facilities are elaborated below.

Function 0 - Input one byte

To input one byte:

```
100 LINE L 0 : [Here with byte in QA]
110 [Here if input buffer empty]
```

The line number L must be even. If there are any characters in the input buffer, the LINE command succeeds and the next received byte is returned in the system variable QA. If the line is in image input mode QA will have a value between 0 and 255 (decimal) corresponding exactly to the received data byte. If the line is in normal mode QA will be between 0 and 127 (decimal) and certain characters like Null, ^C, ^O etc. are not seen due to special interpretation. If no characters are available control goes to the next line of the program. Thus the command never suspends the job.

If you want to wait until a character is available, use the following subroutine:

```
700 LINE L :LET C=QA :RETURN
705 WAIT 30 1 :GOTO 700
```

This routine returns with a character in C. If there are no characters the routine waits for three seconds and tries again. Any characters that arrive during the wait will be buffered in the normal way and all such characters will be delivered by the

subroutine at the end of the wait period.

If you need to respond as soon as a character is received, the possible three second delay may not be acceptable. In this case you could use

```
705 WAIT 1 1 :GOTO 700
```

which will respond within 100 milliseconds. The WAIT 1 should not be used unnecessarily since it places heavy demands on system resources. If you need to respond quickly to every character typed it is best to place the line in image input mode and use a WAIT 30 1.

Function 1 - Image Output

For image output the line number L must be odd. The 8-bit argument is output to the line exactly as supplied with no parity generation, filling or CRC computations. The LINE command always succeeds, but if the line output buffer is full the job may be suspended for a while.

The following program sends an ascending 'binary count' pattern to a specified line:

```
 20 $LINE NUMBER:
100 INPUT ?20 L :LET L=L!1 :GOTO 120
110 PRINT '?' :GOTO 100
120 LET B=0
130 LINE L 1 B :LET B=B+1 :LOOP
```

Function 1 - Force

A privileged job may also use function 1 with an even line number L. In this case the character is processed by the system exactly as if it had been typed at the keyboard of the terminal connected to line L. This facility is useful in tutorial situations where an instructor can show someone what to type even if he is at a remote terminal. It is also useful occasionally if a terminal becomes faulty in the middle of an important job.

The executive FORCE command allows you to type on other keyboards in this way.

Function 2 - Line Characteristics

Function 2 of the LINE command allows the line characteristics word to be read and set under program control. If [argument] is -1, the command simply reads the line characteristics into QA. If [argument] is positive, certain bits of the characteristics word are set as specified, and the old characteristics are returned in QA.

The command

```
LINE L 2, -1 :LET C=QA
```

will read the characteristics of line L and store them in the variable C. Note the comma which avoids 2 -1 evaluating to 1.

The line characteristic word C is a bit mask in format:

Input side (L even):

1	Suppress echo of carriage-return linefeed only.
2	Suppress echo until completion of next INPUT or ACCEPT command. (still echoes LF & CR)
4	Image mode input.
8	Simplex line, inhibit all echo.
16	Convert lower-case input to upper-case.
32	Convert input from CCITT telex code to ASCII
128	Every character is treated as a break-character.
256	Line is attached to a job.
C/512&15	Interface type code (see below)

Output side (L odd):

4	Convert lower-case output to upper-case.
8	Line output is paused by control-S.
16	Pause mode is enabled.
32	Convert output from ASCII to CCITT telex code.
64	Suppress program printing (like control-X).
128	Generate even parity on output.
512+	same as for input side.

Interface type codes:

0	KL11
1	LC11 (parallel DECwriter)
2	DC11 (programmable speed, modem control)
3	DL11 (including DL11-E)
4	Pseudo-console
5	DP11 (synchronous)
6	DH11 multiplexer
7	DJ11 multiplexer
8	DZ11 multiplexer

Bit values greater than 128 are read-only and cannot be changed by the LINE command.

The command

```
LINE L 2, -1 :LINE L 2 QA!B
```

may be used to set a particular bit B. The command

```
LINE L 2, -1 :LINE L 2 QA&-B-1
```

may be used to clear a particular bit B. The bit assignments are such that

```
LINE L 2
```

establishes the normal default conditions.

Function 3 - Modem Status

Control of modems is an AIMS option. When the option is present AIMS takes full advantage of the facilities provided by interfaces such as the DL11-E, DM11-BB and DC11, and will answer the phone, deal with carrier fail, etc. in an appropriate manner. Function 3 of the LINE command enables the user to read the state of a modem, and also to control its operation if desired.

The modem state is represented as a device-independent bit pattern as follows:

hardware state:

* 1 Modem is connected to line (ie. DATA rather than TELE)
* 2 Modem is sending out a carrier (ie. REQ-TO-SEND is on).
4 Modem is ready for sending
8 Modem is receiving a carrier (ie. CAR DET on).
16 The received carrier has changed state (ie. come up or gone down).
32 The phone is ringing (this pulses in step with the bell)

Software control bits:

* 64 Do not answer phone when it rings.
* 128 Do not drop the line, even if carrier fails.

The bits marked * may be changed by means of function 3. The other bits are determined by the state of the modem and are read-only.

The software control bits allow calls to be ignored on particular lines, and also enable a line to be held regardless of what is going on at the other end. This is useful in experimental or error-prone conditions.

When using function 3 of the LINE command, the line number must be even. If [argument ne] is -1, the command just reads the current modem status into QA. If [argument ne] is positive, the command sets the modem status as specified, and then reads the latest status into QA. The status returned in QA may differ from [argument ne] if you attempt to set an impossible condition.

Function 4 - Speed Setting

Some line interfaces, such as the DC11, DH11, and DZ11 allow program variation of the Baud rate. This is done by writing the appropriate speed code into the interface hardware register, using function 4 of the LINE command:

LINE [line ne] 4 [speed ne]

sets the speed of line number [line ne] as specified by [speed ne]. [speed ne] must evaluate to one of the permitted speed codes for the line interface. The DC11 has four possible speeds coded from 0 to 3. The actual speeds obtained depend on the particular type of DC11. For the DH11 and DZ11 the speed code specifies a definite speed as follows:

Code	DZ11	DH11 (Baud)
0	50	N/A
1	75	50
2	110	75
3	134.5	110
4	150	134.5
5	300	150
6	600	200
7	1200	300
8	1800	600
9	2000	1200
10	2400	1800
11	3600	2400
12	4800	4800
13	7200	9600
14	9600	External speed A
15	N/A	External speed B

For example the command LINE 6 4 9 would set the receive speed of DJ line KB6 to 1200 Baud. Normally the keyboard and printer will be operated at the same speed, and a second LINE command is needed to change the printer speed like LINE 7 4 9 to set PT7.

Speed changing may be done more easily by means of the executive

.SET SPEED b

command, which sets the speed of both keyboard and printer to b Baud.

AUTOMATIC DIALLING

AIMS supports the DN11 automatic dial unit which enables the system to initiate a telephone or Telex call under program control.

Initiating a Call

The command

DIAL [DN11 number ne] [phone number se]

stores the current DN11 status in QA, checks if the DN11 is free, and if so begins dialling the number specified by the string expression. Execution continues along the same line as the DIAL command.

If the DN11 is busy or is without power the DIAL command fails and simply returns the current DN11 status in QA.

Reading DN11 Status

The status of a DN11 may be read at any time by means of the command

DIAL [DN11 number ne]

which returns the current DN11 status in QA and continues execution along the line.

The value returned in QA is the contents of the DN11 hardware status register which has bits set as follows:-

Octal	Decimal	Meaning
1	1	Call request. PDP-11 is attempting to initiate a call.
40	32	Call established. A number has been dialled and the called party has answered.
010000	4096	Data line occupied. The line is already in use, you must wait until the present call terminates.
040000	16384	Abandon call. The dial attempt was unsuccessful, try again.
100000	32768	Modem power off.

Checking for Call Establishment

When you execute a command like

DIAL 2 '012833801'

this merely initiates the dialling sequence; it does not wait for the connection to be made. It is the programmer's responsibility

to check later that the call has been correctly established. This may be done by reading the DN11 status with a second DIAL command and checking to see if the CALL ESTABLISHED bit is set.

A program can use the WAIT/WAKE mechanism to suspend execution until a dialling sequence is completed by doing a WAIT command with a [wake mask ne] which includes 8.

Example program

To make a call on DN11 number 2.

```
 90 LET Z=1_16-1
100 DIAL 2 '012833801' :WAIT 200 8 :GOTO 110
102 PRINT '?CANNOT START DIALLING, DN11 STATUS=' QA&Z :STOP
110 DIAL 2 :IF QA&32 :PRINT 'OK' :GOTO ...
112 PRINT '?CANNOT ESTABLISH CALL, DN11 STATUS=' QA&Z :STOP
```

In this example we use a 20 second timeout so that the program recovers even if for some reason the DN11 fails to complete the dialling sequence.

AIMS facilities for communication with SWIFT

The facilities described in this subsection are necessarily dependent upon decisions taken by S.W.I.F.T. and these cannot be determined or predicted by Arbat.

Communication with the SWIFT concentrator is by means of a DU11 interface to a synchronous modem. The line communications protocol is similar to but different from the IBM binary synchronous contention protocol. Messages are transmitted over the line in blocks of upto 384 bytes which are CRC checked. The line traffic consists of a sequence of data blocks interspersed with control sequences which are used for block acknowledgement and line turnaround.

A special AIMS command, SWIFT, is provided for driving the DU11 line. This enables an AIMS system to support the SWIFT communications protocol in a two level manner:

1) AIMS contains a machine-code program for transmitting and receiving data blocks and control sequences. This program also converts between the internal ISO character code and the EBCDIC code that is used on the line, and it computes the CRC check bytes for each data block.

   This program does not however concern itself with the details of the communications protocol. It merely transmits sequences or blocks as directed by the second level, and notifies the second level whenever a sequence or block is received.

2) There is a special AIMS job, called the SWIFT protocol job, which receives the information from level 1 and takes the appropriate action to implement the SWIFT communications protocol. This job is responsible for making line bids, sending blocks, checking CRCs, acknowledging blocks, error recovery and so on. Message blocking and deblocking is performed by level 1 under instructions from level 2.

AIMS contains a BLOCK BUFFER capable of transmitting or receiving one data block. It also contains a MESSAGE BUFFER capable of holding one maximal length message (upto 2000 characters).

Message Reception

When a data block is received it is stored in the block buffer and the protocol job is notified. If the block is acceptable the protocol job will instruct level 1 to append the block to the message buffer, and another block can then be received. Eventually a complete message is assembled in the message buffer and the protocol job then notifies the SWIFT message handling job that a message has arrived. The message handling job can read the message into an array by means of a SWIFT command.

Message Transmission

When a message is ready for transmission the SWIFT message handling job places it in the message buffer by executing a SWIFT command. This command also notifies the protocol job that a message is awaiting transmission.

The protocol job then obtains control of the line by transmitting an appropriate set of control sequences and prepares to send the message. It instructs level 1 to begin deblocking the message by moving the first 384 bytes of it from the message buffer to the block buffer and appending the appropriate CRC bytes. The message is also converted from ISO to EBCDIC at this stage. When the block has been transmitted the protocol job will wait for an acknowledgement and will send the block again if necessary. If the block is accepted the protocol job will instruct level 1 to move the next 384 bytes of the message into the block buffer and so on.

Eventually the whole message is successfully transmitted and the protocol job will then notify the SWIFT message handling job that the message has been sent. At the same time the message buffer is marked as being free so that another message can be loaded into it if desired.

Message Handling Job

This job is responsible for transferring messages between disk and the AIMS message buffer, the actual transmission or reception of the messages being performed by the SWIFT protocol job.

Three SWIFT commands are provided for use by the message handling job:-

SWIFT [Swift line#] 0 [command se]>$[ne]

Function 0 exchanges a string with the SWIFT protocol job and enables the message handling job to communicate directly with the protocol job. [command se] is a string expression which constitutes a command to the protocol job. This string is sent to the protocol job and it returns an appropriate reply string. The reply string is stored in the dollar-line specified by [ne]. The SWIFT0 command language is described later.

SWIFT [Swift line#] 2 A(J)

Function 2 reads a SWIFT message from the message buffer into the array A(). The message always begins with SOH and ends with ETX followed by a byte containing 128 (except that the 128 byte is omitted if the message happens to completely fill the array). QA gives the length of the message in bytes, from the SOH to the ETX inclusive.

SWIFT [Swift line#] 3 A(J)

Function 3 copies a SWIFT message from the array A() to the message buffer. The message at A(J) must begin with SOH and end

with ETX. Anything following an etx is ignored. The command will fail if the message buffer is in use for the transmission of an earlier message or the reception of a message.

The message handling job causes a message to be transmitted by executing a SWIFT 3 command and then going to sleep until woken by the protocol job. The message handling job then executes a SWIFT 0 command to find out if the message has been transmitted ok. Transmission may fail due to the concentrator or line going down or due to a race condition in which a message is received from the concentrator after execution of the SWIFT 3 command but before the protocol job has time to transmit the message. The message handling job must therefore be prepared to repeat the whole message transmission procedure, either by holding the message in an array throughout or by retrieving it again from disk.

When the message handling job has been notified that a message has been successfully transmitted it may do one of two things: (a) it may initiate the transmission of another message, or (b) it may inform the protocol job that there are no further messages to be transmitted at the moment. The message handling job has 1.5 seconds in which to do this. During this interval the protocol job will keep control of the line. If another SWIFT 3 command is executed within this interval the protocol job will immediately begin transmission of the new message. If the protocol job is notified (via SWIFT 0) that there are no further transmissions, or if the interval expires without notification, the protocol job will give up line control. This is not an error condition, it simply means that any subsequent message transmission will involve a line bid. The message handling job should however make an effort to do (a) or (b) as soon as possible after it has been woken.

When the message handling job is idle it should be sleeping with the WAKE enable bit set. If a message is received from the concentrator the protocol job will WAKE the message handling job. As for message transmission, the message handling job should execute a SWIFT 0 command to find out the reason for its being woken. On finding that the reason is the arrival of a message the message handling job should read the message by executing a SWIFT 2 command. This command frees the message buffer and wakes the protocol job, and it is possible that the protocol job will immediately begin receiving another message. Consequently the SWIFT 2 command can only be executed once per received message and it is the responsibility of the message handling job to correctly store the message on disk.

When the protocol job has received the final block of a message from the concentrator, there will in general be some delay before the message handling job reads the message with a SWIFT 2 command. During this delay the protocol job will hold off the concentrator by means of WACKs for upto 30 seconds. If the message handling job fails to read the message within this time the concentrator will mark the CBT as being down. Therefore the message handling job should take care to react smartly to received messages.

SWIFT command Errors

The SWIFT command may fail for several reasons:-

QE	Func	Reason
0	All	Invalid Swift line number
1	All	No protocol job
2	2	No message available
	3	Message buffer in use
3	2	Array too small for message (QA=message length in bytes)
	3	Message too long for message buffer
4	3	SOH or ETX absent from message in array

SWIFT commands used by Protocol Job

Apart from the three SWIFT command functions described earlier, all SWIFT functions are reserved for use by the protocol job only. If any other job executes any of these functions confusion will ensue.

All SWIFT commands begin with a [Swift line ne] followed by the function code number.

SWIFT [Swift line#] 1, [ne]

Exchanges DU11 modem status word. The contents of the DU11 receiver status register is stored in QA and the register is set to the value of [ne] if this is positive. For meaning of this register see DU11 manual.

SWIFT [Swift line#] 4

Reads the next byte from the level 1 received event queue into QA. The command fails with QE=2 if the event queue is empty. The possible received events are described later.

SWIFT [Swift line#] 5 [control sequence ne] [argument se]

Instructs level 1 to transmit the control sequence specified by [control sequence ne]. The possible control sequence numbers are described later. [argument se] is a character string which is converted to EBCDIC and output immediately before the bytes that constitute the control sequence proper. This is used for outputting the SWIFT CBT identification or line address when transmitting ENQs or ACKOs on dial-up lines.

SWIFT [Swift line#] 6

Instructs level 1 to append the current contents of the block buffer to the message buffer. If the block begins with SOH the message buffer is automatically cleared before the append. Otherwise the initial STX is removed and the block is appended to any partial message already present in the message buffer. The byte that terminated the block (either ETB or ETX) is returned in QA so that the program can tell whether the message is now

complete. The command will fail with QE=3 if the message buffer is too full to hold the block.

SWIFT [Swift line#] 7

Instructs level 1 to copy the next block of bytes from the message buffer to the block buffer. The bytes are converted from ISO to EBCDIC, STX is prefixed if the first byte is not SOH, and ETB is added if the last byte is not ETX. The CRC is calculated and appended. After this operation the block buffer contains a properly formatted data block ready for transmission. The command will fail with QE=2 if the end of the message has been reached, in which case the block buffer contains nothing in particular.

SWIFT [Swift line#] 8

Computes the CRC value for the block currently in the block buffer and returns it in QA. If the block buffer contains a valid block with its correct CRC, QA will be zero. Used for validating the CRC of a received block before appending it to the message buffer.

SWIFT [Swift line#] 9

Resets level 1 so that the next function 7 command will operate on the first block of the message in the message buffer.

SWIFT [Swift line#] 10, [ne]

Exchanges message buffer status. The current message buffer status is stored in QA, and the status is then set to [ne] if this is positive. The possible message buffer status codes are:

0	FREE	Message buffer free
1	MSOP	Message being transmitted from buffer or waiting to be transmitted
2	MSAV	A received message is available in the buffer
3	MSIP	A message is being received into the buffer.

The SWIFT function 3 is only allowed when the status is 0, and function 2 is only allowed when the status is 3. Function 3 sets the status to 1 and function 2 sets it to zero. The message buffer status allows the protocol job to work out what the message handling job is doing.

SWIFT [Swift line#] 11 [mode ne]

Initialises level 1 completely. Sets up DU11 interrupt vectors, initialises DU11 hardware registers, resets all software variables to standard state, abandons any pending transmissions, clears the received event queue. [mode ne] specifies the mode of operation as 0=Leased, +1=Pstn, -1=Ignore. The Ignore mode disables the DU11 completely.

SWIFT [Swift line#] 12 $[ne]

DOS only: Reads the next SWIFT0 command string from the message handling job into $[ne] and sets QA to the job number of the message handling job. Command will fail if no job is currently

executing a SWIFT 0 function.

SWIFT [Swift line#] 13 [job# ne] [reply se]

DOS only: Returns the string [reply se] to the specified job which is presumed to be suspended executing a SWIFT 0 function. Command will fail with QE=1 if the specified job does not exist or is not awaiting a reply.

SWIFT [Swift line#] 14 A(J)

Used for testing the system without using a real DU11 interface. Enters the DU11 interrupt handler in a way which makes it think a DU11 receive interrupt has occurred. Instead of examining the real DU11 hardware registers, the interrupt handler is instructed to examine the four words located at A(J) through A(J+3). An AIMS program can set these up beforehand to simulate any desired DU11 condition. Function 14 thus allows an AIMS test program to pump bytes into the system as if they came down the line from a SWIFT concentrator. DOS: A(J) is not specified in the command. The simulated registers are always located at GV(10-13).

SWIFT [Swift line#] 15 A(J)

As function 15 except it simulates a DU11 transmit interrupt. This allows an AIMS test program to suck bytes from the system as if they were going down the line to a SWIFT concentrator. DOS: A(J) is not specified in the command. The simulated registers are always located at GV(10-13).

SWIFT [Swift line#] 16 A(J)

Copies the contents of the BLOCK BUFFER into the array. Used for diagnostic purposes to investigate CRC errors on received data blocks.

Received Event Queue

When a control sequence or a block is received by the DU11 it is recognised, converted to an internal EVENT CODE number, and placed on a RECEIVED EVENT QUEUE. This is the queue that is read by function 4 of the SWIFT command. The format of an event in the queue is:

Event code, Low time, High time, Optional args, 0

High time*256 + Low time gives the time of day when the event occurred in tenths of a second past midnight modulo 10000.

Some event codes are followed by one or more argument bytes. For example on a dial-up line the first ENQ is received in the form ID ENQ where ID is a five character name identifying the concentrator. The ID characters are treated as an argument of the basic ENQ control sequence, and level 1 therefore places an ENQ event code on the received event queue, followed by the five ID characters. All events on the queue are terminated by a single zero byte.

The procedure for reading the next event from the queue is to read the first byte (via function 4) and treat this as the event code, read and store the two time bytes, then read subsequent bytes until a zero byte is read. These bytes then constitute the argument if any.

Code	Meaning
1	A block has been received beginning with SOH
2	A block has been received beginning with STX
3	DEOT received
4	EOT received
5	ENQ received
6	RVI received
7	TTD received
8	NAK received
9	WACK received
10	ACK0 received
11	ACK1 received
12	Unrecognisable control sequence received
13	DU11 receiver error (receive overrun or framing error)
14	The modem status has changed
15	The last byte of a data block has just been transmitted

When it is indicated that a block has been received, it may be assumed that the block is stored in the block buffer. However, the block buffer is not protected from overwriting, and if another block is received before the event queue is serviced the first block will be lost.

Control Sequence Codes

Level 1 can be instructed to output control sequences or blocks by means of function 5 of the SWIFT command. The control sequence codes are identical to the event codes given above, except that code 2 and codes above 11 do not apply. Thus to send a ACK0 you execute SWIFT function 5 with a sequence code of 10. The [argument se] of function 5 is normally null but may be used, for example, if you wish to send an ENQ prefixed by the five-character CBT identifier.

Control code 1 is used to instruct level 1 to output the contents of the block buffer. This must have been previously set up by means of SWIFT function 7. Code 1 is used to output the block whether it begins with SOH or STX.

SWIFT0 Command Language

The message handling job communicates with the protocol job by means of function 0 of the SWIFT command, which passes a string to the protocol job and returns another string in reply.

The possible command strings that can be sent to the protocol job are as follows:

INIT Initialises everything without regard to current state.
RESET Sets protocol job into the normal idle state.
IGNORE Sets protocol job into the TERIGN state.
CBTID string
Informs the protocol job that the specified string (should be 5 characters) is to be used as the CBT identifier.
CONID string
Informs the protocol job that the specified string is to be used as the concentrator identifier.
STATUS Returns current state (see below).
PSN Sets PSN (ie. dial-up) mode.
LEASED Sets leased-line mode (opposite of PSN).
TRACE dev n:file.ext=options
Causes the protocol job to begin outputting a line trace to the specified device and file (normally a 2400 Baud vdu). Options is one or more letters selecting subsets of the trace as:
E trace errors
R trace all received events
T trace all transmitted control sequences
M trace interactions between protocol and message handling jobs
TRACE CL Closes trace file.
LOG string
Outputs the string to the trace channel. Used for inserting comments into the trace.
TXEND No more messages to transmit at this time.

Protocol Job Status

The status returned by the SWIFT0 status command is of the form:

S=xxxxxx B=yyyy T=zzzzz

where

xxxxxx gives the overall state of the CBT as
idle
SLAVE Concentrator is master, CBT expecting to receive a block.
STORE Waiting for message handling job to read a received message.
BIDRPY Waiting for reply to a CBT line bid
NXTOPM Waiting for message handling job to provide the next output message or indicate that there are no more.
BLKRPY Waiting for concentrator to reply to a block sent by CBT.
TERCON Terminated by concentrator (DEOT received)
TERCBT Terminated by CBT due to timeout or countout. Probably the concentrator is down.
TERIGN Terminated due to SWIFT0 IGNORE command.

yyyy gives the state of the message buffer (see SWIFT function 10).

zzzzz shows what has happened to the most recent request for message transmission:

BUSY message being transmitted
SENT message has been transmitted ok
FAIL message not sent

if T=FAIL status is returned when attempting to send a message, the attempt should be repeated unless s=terxxx; the transmission failure could have been caused by a received message overwriting the message buffer. Note that the normal state is T=SENT and this state is set by the INIT and RESET commands. The message handling job is expected to know whether or not it has requested message transmission (via SWIFT 3), and no significance should be attached to t=sent at other times.

Summary of SWIFT command functions

0	EXS	Exchange strings with protocol job
1	MOD	Exchange modem status
2	REA	Read message into array
3	WRI	Write message from array
4	EVR	Read next byte from received event queue
5	EVW	Initiate transmission of specified control sequence
6	IPB	Append block to message buffer
7	OPB	Fill block buffer from message buffer
8	CRC	Return CRC of block now in block buffer
9	FOP	Set to output first block via next OPB function
10	STA	Exchange message buffer status
11	INI	Initialise level 1
12	IPR	DOS only: Read next SWIFT0 command string
13	REP	DOS only: Send SWIFT0 reply string to specified job
14	IFR	Simulate DU11 receive interrupt
15	OFR	Simulate DU11 transmit interrupt
16	RRB	Read raw received block

Only functions 0, 2 and 3 may be used by the message handling job.

Local Testing using a Concentrator Simulator

an AIMS system for SWIFT can be tested without making a connection to a real SWIFT concentrator (this is an option selected at system generation time using F.NODU). This is done by means of functions 14 and 15 of the SWIFT command which enable an AIMS program to simulate the DU11 line interface. The real SWIFT line runs at 2400 baud or about 300 bytes per second and it is quite feasible to simulate operation at this speed, with the exception of the CRC checking which is very slow when written as an AIMS program.

The test facilities currently available are as follows:-

SWIFT0.BAS The standard SWIFT protocol job. Warning! MON and DOS versions of this program are diferent. Also a special version is needed in the U.K. to accommodate incompatabilities between G.P.O. and CCITT standards.

SWIFTM.BAS A simple message handling job for running the SWIFT Qualification tests.

QUACON.BAS A job which simulates the SWIFT concentrator and administers the Qualification tests.

QUACON is only used for local testing when it is desired to simulate the SWIFT concentrator. To do the real SWIFT Qualification Tests QUACON is replaced by another program called QUADRV.

Before doing any local testing it is essential to execute a SWIFT L 14 command to inform the system that you are going to use a simulated DU11 rather than a real one. Failure to do this will cause a system crash when the protocol job attempts to initialise the line (if there is no real DU11 in the configuration).

The protocol job should be started on a pseudo-console by the command sequence:-

```
.ZE SWIFT0
UP

THE JOB SHOULD IMMEDIATELY GO TO SLEEP.
```

THE QUALIFICATION MESSAGE HANDLING JOB SHOULD BE STARTED ON ANOTHER PSEUDO-CONSOLE AS:-

```
.ZE SWIFTM
LOG DEVICE:PT nn:

The job should immediately go to sleep.
```

Finally the concentrator simulator should be started on a fast visual display:-

```
.ZE QUACON
EI.TXT READ
SINGLE-BLOCK MESSAGES CREATED
MULTI-BLOCK MESSAGE CREATED
TEST:
```

Two commands are now available:-

/xxxxxx Sends xxxxxx to the protocol job as a SWIFT0 command and prints reply. Allows direct interaction with protocol job if necessary.

nn Starts qualification test number nn, the test numbers being those in the qualification specification OPS.10.

For local test purposes it is recommended that the logging device for both the SWIFT0 and SWIFTM jobs be the same fast visual display as is used to control QUACON. This ensures that all

logging output appears in an easily discernable order. If the visual display is slower than 2400 baud it may be necessary to use several. On no account should trace output ever be sent directly to a lineprinter. There is a parameter DF in SWIFT0.BAS which is a scale factor determining the length of all protocol job timeouts. For local test purposes this may be increased if necessary to accommodate delays due to QUACON or slow logging devices.

Explanation of QUACON

Both QUACON and SWIFTM contain a block of dollar-lines that specify the action required for each qualification test. In the case of SWIFTM this merely specifies that certain messages are to be sent to the concentrator in the appropriate order. No details are given of messages to be received from the concentrator since SWIFTM is always ready to receive. For QUACON the test specification is more complicated: it includes the exact sequence of line control signals that are to be sent to the CBT, and the sequence expected to be received from the CBT. QUACON checks the replies it actually receives and stops if there is any discrepancy.

The specification of a particular test is a sequence of items in a dollar-line. The possible items are:-

nn Send the control sequence in $nn
R=nn Expect a reply control sequence matching $nn
T=xx Send the message block in array xx.
Wtt Wait for tt tenths of a second.
RBfl Expect to receive a message block of the form specified by fl, where f specifies the first byte of the block and L the last. F and L are single digits coded as: 0=ETB, 1=SOH, 2=STX, 3=ETX. So RB13 means read a block beginning with SOH and ending with ETX.
REPnn (...) Repeat the sequence of items enclosed in ( ) nn times.

An example of a test specification is:

3000 $01 35 R=40 T=S1 R=41 34

This is test number 01. It may be read as: Send an ENQ ($35), read an ACK0 reply ($40), send a single-block message (array S1), read an ACK1 reply ($41), and send an EOT ($34).

(Note: in fact the test specifications contain other items such as SEND2000, RM, WWS, and TXEND. These are obsolete and are ignored by QUACON)

Dollar-lines 33 through 42 define the possible line control sequences (eg: NAK, TTD, etc) in an octal representation of EBCDIC. arrays EI and IE are used to convert between ISO and EBCDIC. Arrays S1, S2 and S3 contain three numbered single-block messages. Arrays M1, M2 and M3 contain the three blocks of a single multi-block message.

Before running any tests it is necessary to set the TRACE control parameters for the protocol job, otherwise it will not generate any logging output. The QUACON command

TEST:/T PT N:=TRME

sends a TRACE command to the protocol job and enables Transmit, Receive, Message, and Error traces.

QUACON communicates with the message handler via GV(5-7). When you give the command

TEST:34

to start test number 34, QUACON LETs the string 34 into GV(5) to tell SWIFTM that test 34 is about to begin (strings are used because some tests have names like AA). The message handling job may be reset at any time by setting GV(5)=-1, waking SWIFTM, and waiting till GV(5)=0. SWIFTM is then ready to receive the next test number.

SWIFTM itself only communicates with the protocol job via functions 0, 2 and 3 of the SWIFT command. SWIFTM contains the four messages that the qualification procedure requires the CBT to send. The only output from SWIFTM is the logging output.

SWIFT Qualification Tests

Both SWIFTO and SWIFTM may be used without modification for doing the qualification tests with a real SWIFT concentrator. The set up procedure is the same, except that no SWIFT 14 or 15 commands should be given. If any local testing has been done the system should be restarted to restore the use of the real DU11 interface.

For doing the tests with a real SWIFT concentrator you use QUADRV.BAS rather than QUACON, since the concentrator simulator is not required. QUADRV merely informs SWIFTM (via GV(5)) which test is to be done next.

Note that for the qualification tests all these programs run at priority 0 and no other jobs should be on the system.

Special action should be taken if AIMS fails any SWIFT test. The test should be tried at least three times; the SWIFT concentrator test program sometimes gets into peculiar states and it can happen that a test will succeed on the second go. The line trace should be examined whilst the test is going on to see whether the problem appears to be a CBT or a concentrator fault. If the behaviour shown on the line trace appears to conform to the SWIFT specifications laid down in the OPS.10 document, reference should be made to the latest available information about the state of the concentrator test program. Some versions of this program will fail certain tests although there is in fact no error in the CBT. The concentrator operator should have a list of these faulty tests. Finally, if the concentrator operator will not accept that the failure is a SWIFT problem, the test should be run again with the line trace output to a disk file. This will give a permanent copy of the trace for further investigation.

Explanation of Line Trace

The trace contains one line of text for each control sequence that is transmitted or received. The general format is:

time state dir event

where

- time is the time of day in tenths of a second past midnight, modulo 10000.
- state is the overall state of the CBT (see below).
- event identifies the control sequence or block that was sent or received.
- dir is R if the event was received by the CBT, and T if it was transmitted by the CBT.

An exmple should make this clear:

```
1807 IDLE   T ENQ
1808 BIDRPY R AK0
```

This shows an initially idle CBT making a line bid. It transmits an ENQ at time 1807, and receives an ACK0 at time 1808. When it receives the ACK0 the CBT state is BIDRPY, which means Awaiting Reply to a Bid. The interval between the two events may be found by subtracting the two daytimes: 1808-1807=1, indicating an interval of 0.1 seconds.

The event codes are mainly self explanatory, such as ENQ, RVI, DEOT and WACK. ACK0 appears as AK0 and ACK1 as AK1. A received block will be traced as R SOH if it begins with SOH, or R STX if it begins with STX. An unrecognisable sequence is logged as R JUNK.

Blocks transmitted by the CBT are always traced as T SOH. This is so even if the block begins with STX. The T SOH is generated when the CBT begins to transmit the block. When the last byte of the block has been transmitted an R BTX event is logged. For example

```
1809 BLKRPY T SOH
1811 BLKRPY R BTX
1812 BLKRPY R AK1
```

Here the CBT begins transmitting a block at time 1809. It finishes block transmission 0.2 seconds later at time 1811. An ACK1 is then received from the concentrator.

CBT timeouts give rise to R TIM events in the trace. For example, if the CBT were to send a block to the concentrator and receive no reply, the trace would appear as follows:

```
7229 BLKRPY T SOH          [begin transmitting block
7242 BLKRPY R BTX          [finish transmitting block
7273 BLKRPY R TIM          [timeout occurs 7273-7242= 3.1 secs later
7273 BLKRPY T ENQ
7275 BLKRPY R AK0
```

26. THE ASCII CHARACTER CODE

CHARACTER	%C	OCTAL	CHARACTER	%C	OCTAL
Null	0	0	@	64	100
	1	1	A	65	101
	2	2	B	66	102
control-C	3	3	C	67	103
	4	4	D	68	104
	5	5	E	69	105
	6	6	F	70	106
	7	7	G	71	107
	8	10	H	72	110
Tab	9	11	I	73	111
Linefeed	10	12	J	74	112
	11	13	K	75	113
Formfeed	12	14	L	76	114
Return	13	15	M	77	115
	14	16	N	78	116
control-O	15	17	O	79	117
	16	20	P	80	120
control-Q	17	21	Q	81	121
	18	22	R	82	122
control-S	19	23	S	83	123
	20	24	T	84	124
	21	25	U	85	125
	22	26	V	86	126
	23	27	W	87	127
control-X	24	30	X	88	130
control-Y	25	31	Y	89	131
	26	32	Z	90	132
Escape	27	33	[	91	133
	28	34	\	92	134
	29	35	]	93	135
	30	36	^	94	136
	31	37	_	95	137
Space	32	40	`	96	140
!	33	41	a	97	141
"	34	42	b	98	142
#	35	43	c	99	143
$	36	44	d	100	144
%	37	45	e	101	145
&	38	46	f	102	146
'	39	47	g	103	147
(	40	50	h	104	150
)	41	51	i	105	151
*	42	52	j	106	152
+	43	53	k	107	153
,	44	54	l	108	154
-	45	55	m	109	155
.	46	56	n	110	156
/	47	57	o	111	157
0	48	60	p	112	160
1	49	61	q	113	161
2	50	62	r	114	162
3	51	63	s	115	163
4	52	64	t	116	164
5	53	65	u	117	165

6	54	66	v	118	166
7	55	67	w	119	167
8	56	70	x	120	170
9	57	71	y	121	171
:	58	72	z	122	172
;	59	73	{	123	173
<	60	74	|	124	174
=	61	75	}	125	175
>	62	76	~	126	176
?	63	77	Rubout	127	177

27. INDEX