FTN77®

Library Reference

Salford Software

The Compiler Specialists

IMPORTANT NOTICE

Salford Software Ltd. gives no warranty that all errors have been eliminated from this manual or from the software or programs to which it relates and neither the Company nor any of its employees, contractors or agents nor the authors of this manual give any warranty or representation as to the fitness of such software or any such program for any particular purpose or use or shall be liable for direct, indirect or consequential losses, damages, costs, expenses, claims or fee of any nature or kind resulting from any deficiency defect or error in this manual or such software or programs.

Further, the user of such software and this manual is expected to satisfy himself/herself that he/she is familiar with and has mastered each step described in this manual before the user progresses further.

The information in this document is subject to change without notice.

May 1998

Preface

FTN77 provides a number of useful subroutines and functions in addition to those specified in the ANSI Standard. Some of the functions that have been provided are defined as intrinsic functions and are described in chapter 11 of the *FTN77 User's Guide*. The remaining functions, and all of the subroutines are described in this library reference, together with examples of their use where appropriate. Most of this information is also available in the on-line help systems. Some of these routines will be widely applicable, while others were written primarily for use within the compiler system (FTN77 is itself written in Fortran) but may be of use to some users in specialised circumstances and are therefore documented here. Note that all functions obey the implicit typing rules in effect, so it will be necessary to declare some of the functions before they can be used.

If you are using a routine which is not described in this manual, but which has been suggested by Salford Software as a solution to some specific problem, there is no need to alter your code. Any routine in DBOS.LIB which has been used by one of our users is guaranteed to remain in the library in all subsequent versions of the software. If it is not documented it is probably because a better routine has been made available. The routines are arranged in chapters as functional groups. Within the chapters the routines are arranged in alphabetical order.

The following symbols are used to denote the availability of each routine on the various platforms:

no symbol	Function is available on all platforms, DOS, Win16 and Win32.
❶	At the time of going to press, function is only available under DOS .
❷	Function is only available under DOS.
❸	Function is only meaningful under DOS. Under Win16 and Win32 the function either has no operation or is not relevant. This category is for DOS programs and programs that are being ported from DOS to Windows.
❹	Function is available under DOS and also in ClearWin+ but with slightly different functionality. See this guide for the DOS function and the Clearwin+ documentation (the manual or an information file on the release disk) for information on the ClearWin+ variant.
❺	Function is only available under Win32.

On the next page you will find a list of chapter headings in this guide. A full table of contents appears after the acknowledgements.

Chapter headings in this manual:

Acknowledgements

* * *

FTN77 is a registered trademarks of Salford Software Ltd.

DBOS, Salford C++, SLINK and ClearWin+ are trademarks of Salford Software Ltd.

FTN90 is a joint trademark of Salford Software Ltd and the Numerical Algorithms Group Ltd.

MS-DOS, Windows, Windows 95 and Windows NT are trademarks of Microsoft Corporation.

BRIEF is a trademark of Borland International Inc.

Intel is a registered trademark of Intel Corporation.

AUTOMAKE is a trademark of Polyhedron Software Ltd.

Table of Contents

1.

Bit-handling

The routines in this chapter provide for bit packed logical arrays. The routines are compiled in-line using the Intel bit manipulation instructions. The result executes about as fast as a reference to a logical array, but the data is stored 8 or 16 or 32 times more efficiently. The bit array may be held in an array (or even a simple variable) of any type (usually INTEGER*2). These routines do not check that their arguments are in range, even in CHECK mode. Bits are numbered from 0. Bit 0 is the least significant bit of the first word of the array.

CLEAR_BIT@

Purpose To clear the N'th bit of an array.

Syntax
```
SUBROUTINE CLEAR_BIT@(IA,N)
INTEGER*2 IA(*),N
```

Description Clears the N'th bit of the array IA. N can be INTEGER*1, INTEGER*2 or INTEGER*4 and IA can be of any datatype.

SET_BIT@

Purpose To set the N'th bit of an array.

Syntax
```
SUBROUTINE SET_BIT@(IA,N)
INTEGER*2 IA(*),N
```

Description Sets the N'th bit of the array IA. N can be INTEGER*1, INTEGER*2 or INTEGER*4 and IA can be of any datatype.

TEST_BIT@

Purpose To test if the N'th bit of an array is set.

Syntax

```
INTEGER*2 FUNCTION TEST_BIT@(IA,N)
INTEGER*2 IA(*),N
```

Description TEST_BIT@ may be declared as LOGICAL or INTEGER.

Return value TEST_BIT@ returns 1 or 0 (.TRUE. or .FALSE.) according to whether the N'th bit of IA is set or not.

Example

```
C   routine to maintain list of bad records in a
C   file - bit-packing allows for 160000 records
      SUBROUTINE BADREC(N)
      LOGICAL*2 TEST_BIT@
      INTEGER*2 INFO(10000)
      SAVE INFO
      DATA INFO/10000*0/
      CALL SET_BIT@(INFO,N)
      RETURN
      ENTRY CHECKREC(N)
      IF(TEST_BIT@(INFO,N))THEN
       PRINT *,'Attempt to use bad record of file!'
       STOP
      ENDIF
      END
```

2.

Character-handling

The routines in this chapter provide various facilities for manipulating objects of Fortran 77 CHARACTER type.

ALLOCSTR@ ❺

Purpose To allocate dynamic storage and copy a string.

Syntax
```
INTEGER*4 FUNCTION ALLOCSTR@(STRING)
CHARACTER*(*) STRING
```

Description ALLOCSTR@ copies STRING with trailing spaces removed and terminated by a null (i.e. a C-format string), into a dynamic storage space which it allocates. The string can be retrieved using the routine GETSTR@.

Return value The return value of the function is the address of the storage used.

Example See GETSTR@.

APPEND_STRING@

Purpose To add a string to the end of a line.

Syntax
```
SUBROUTINE APPEND_STRING@(LINE,ADDITION)
CHARACTER*(*) LINE,ADDITION
```

Description This routine adds the string ADDITION to the end of string LINE after removing trailing spaces from LINE. This can be used to build up complex strings without the need to do many substring calculations.

Example

```
CHARACTER*80 LINE
CHARACTER*20 SAYINT
INTEGER*4 NO_GREEN_BOTTLES
LINE='THERE ARE'
READ *,NO_GREEN_BOTTLES
IF(NO_GREEN_BOTTLES.EQ.0)THEN
 CALL APPEND_STRING@(LINE,' NO')
ELSE
 CALL APPEND_STRING@(LINE,' '//SAYINT(NO_GREEN_BOTTLES))
ENDIF
CALL APPEND_STRING@(LINE,' STANDING ON A WALL')
CALL SOU@(LINE)
END
```

CENTRE@

Purpose To position a string in the centre of a field.

Syntax

```
CHARACTER*(*) FUNCTION CENTRE@(STRING,IW)
CHARACTER*(*) STRING
INTEGER*2 IW
```

Return value CENTRE@ returns STRING after padding with blanks on the left so that the non-blank part is centred in a field of IW characters. This is very useful for titles.

Example

```
CHARACTER*80 CENTRE@
PRINT *,CENTRE@('FINAL RESULTS',80)
```

CHAR_FILL@ ❺

Purpose To fill a string with a particular character.

Syntax

```
SUBROUTINE CHAR_FILL@(STRING,FILL)
CHARACTER*(*) STRING
CHARACTER*1 FILL
```

Description This routine fills the string in STRING with the character FILL up to the full

length of STRING.

CHSEEK@

Purpose To look for a given string in an ordered array.

Syntax

```
SUBROUTINE CHSEEK@(ITEM,LIST,N,IRES)
CHARACTER*(*) ITEM,LIST(N)
INTEGER*4 N,IRES
```

Description Seeks the string ITEM in the sorted array LIST using a binary chop. Returns the position in IRES or 0 if not found. Note that the LIST array must be sorted in ascending dictionary order.

Example

```
      OPTIONS(INTL)
      CHARACTER*10 FOODS(5)
      CHARACTER*12 MEAL
      DATA FOODS/'BUTTER','EGGS','FISH','MUTTON','SUGAR'/
1     READ (*,'(A)')MEAL
      CALL CHSEEK@(MEAL,FOODS,5,K)
      GOTO(10,20,30,40,50,60),K+1
10    PRINT *,'this is not a food I know about'
      GOTO 1
20    PRINT *,'spread it'
      GOTO 1
30    PRINT *,'boil it'
      GOTO 1
40    PRINT *,'fry it'
      GOTO 1
50    PRINT *,'stew it'
      GOTO 1
60    PRINT *,'put it in coffee'
      GOTO 1
      END
```

CNUM

Purpose To convert an integer to character form.

Syntax

```
CHARACTER*(*) FUNCTION CNUM(J)
INTEGER*4 J
```

Description Converts the INTEGER*4 number J to characters, left-justified with sign if negative.

Example

```
C       routine to open a file of name FREDnnnn
C       where nnnn is a 4-digit integer
        SUBROUTINE FREDOPEN(K)
        INTEGER*4 K
        CHARACTER*8 FRED,CNUM
C       note trick to get leading zeros
        FRED(4:8) = CNUM(K+10000)
        FRED(1:4) = 'FRED'
        OPEN (FILE=FRED,UNIT=1)
        END
```

COMPRESS@

Purpose To compress a string by using tabs.

Syntax

```
SUBROUTINE COMPRESS@(LINE,L)
CHARACTER*(*) LINE
INTEGER*2 L
```

Description COMPRESS@ replaces multiple blanks where possible in a line with tabs to column positions which are multiples of eight. The new length of the line is returned in L. The tabbing scheme is that used by DOS, so the resulting line can be written to a DOS file.

GETSTR@ ❺

Purpose To get a string which was stored using ALLOCSTR@.

Syntax
```
CHARACTER*(*) FUNCTION GETSTR@(PTR)
INTEGER*4 PTR
```

Description This function can be used for strings allocated with the ALLOCSTR@ routine. ALLOCSTR@ and GETSTR@ provide a simple way of storing and retrieving large amounts of character information for which a maximum possible length of each element is known, but where if all trailing spaces were stored the amount of memory required would be excessive. For example, lines of text destined for screen display could be stored in this way (usually a maximum of 80 characters, but often with much trailing space).

Another application of this routine is for C string entities passed to Fortran routines (see chapter 16).

Return value GETSTR@ returns the null-terminated string at address PTR as a Fortran CHARACTER entity, truncating or blank-padding as necessary.

Example

```
INTEGER*4 ALLOCSTR@,PTR
CHARACTER*80 GETSTR@
. . .
PTR=ALLOCSTR@(LINE)
. . .
OUTLIN=GETSTR@(PTR)
PRINT '(A)', OUTLIN
```

LCASE@

Purpose To alter a character argument so that all letters become lower case.

Syntax
```
SUBROUTINE LCASE@(A)
CHARACTER*(*) A
```

Example

```
CHARACTER*10 FRED
FRED = 'ABC123'
CALL LCASE@(FRED)
IF(FRED.NE.'abc123')PRINT *,'LCASE@ routine has failed'
END
```

NONBLK

Purpose To obtain the position of the first non-blank character.

Syntax

```
INTEGER*2 FUNCTION NONBLK(A)
CHARACTER*(*) A
```

Return value NONBLK@ returns the position of the first non-blank character in the character argument A. If the argument is wholly blank, 0 is returned.

Example

```
C       routine to read line of text and return first word
        SUBROUTINE READER(ITEM)
        CHARACTER*20 ITEM
        CHARACTER*80 LINE
        READ (*,'(A)')LINE
        ITEM = LINE(1:NONBLK(LINE))
        END
```

SAYINT

Purpose To return an integer argument as text.

Syntax

```
CHARACTER*(*) FUNCTION SAYINT(I)
INTEGER*4 I
```

Description As an example, the value I=-270 would return the character value 'MINUS TWO HUNDRED AND SEVENTY'.

Example

```
        CHARACTER*80 SAYINT
        INTEGER*4 I
        DO 1 I=10,0,-1
1       PRINT *,SAYINT(I)
        PRINT *,'we have lift off!'
        END
```

TRIM@

Purpose To remove leading blanks.

Syntax

```
SUBROUTINE TRIM@(X)
CHARACTER*(*) X
```

Description TRIM@ is used to remove leading blank characters from the character argument X.

Example

```
C     read names from a file and print them left justified
      CHARACTER*10 NAME
      OPEN (FILE='NAMES',UNIT=10)
 1    READ (10,'(A)',END=2)NAME
      CALL TRIM@(NAME)
      PRINT *,NAME
      GOTO 1
 2    END
```

TRIMR@

Purpose To rotate a character string right until there are no trailing blanks.

Syntax

```
SUBROUTINE TRIMR@(X)
CHARACTER*(*) X
```

Notes If the string is blank, it is left unchanged.

Example

```
C      read names from a file and print them right justified
       CHARACTER*10 NAME
       OPEN (FILE='NAMES',UNIT=10)
 1     READ (10,'(A)',END=2)NAME
       CALL TRIMR@(NAME)
       PRINT *,NAME
       GOTO 1
 2     END
```

UPCASE@

Purpose To alter a character argument so that all letters become upper case.

Syntax

```
SUBROUTINE UPCASE@(A)
CHARACTER*(*) A
```

Example

```
CHARACTER*10 FRED
FRED = 'ABcd'
CALL UPCASE@(FRED)
IF(FRED.NE.'ABCD')PRINT *,'UPCASE@ routine has failed'
END
```

4.

Data sorting

The routines in this chapter provide facilities for sorting arrays of various types. The routines described use a quicksort algorithm, and perform well for data which is originally randomly ordered. Note, however, that these routines are not stable in the strict sense. That is, equal keys do not necessarily maintain their order relative to each other.

CHSORT@

Purpose To sort an array of characters.

Syntax

```
SUBROUTINE CHSORT@(A,CHS,N)
CHARACTER*(*) CHS(N)
INTEGER*4 A(N),N
```

Description CHSORT@ sorts the character array CHS by setting pointers from 1 to N in the array A. After sorting, A(1) contains a pointer to the "first" element of CHS, A(2) to the "second", and so on.

Example

```
      OPTIONS(INTL)
      CHARACTER*20 PUPILS(100)
      INTEGER*4 IP(100)
      DO 1 I=1,100
1     READ (5,'(A)')PUPILS(I)
      CALL CHSORT@(IP,PUPILS,100)
      PRINT *,'sorted list of pupils:-'
      DO 2 I=1,100
2     PRINT *,PUPILS(IP(I))
      END
```

ISORT@

Purpose To sort an integer array.

Syntax

```
SUBROUTINE ISORT@(A,IA,N)
INTEGER*4 IA(N)
INTEGER*4 A(N),N
```

Description ISORT@ sorts the integer array IA by setting pointers from 1 to N in the array A in the same manner as CHSORT@.

RSORT@

Purpose To sort a REAL*4 array.

Syntax

```
SUBROUTINE RSORT@(A,R,N)
REAL*4 R(N)
INTEGER*4 A(N),N
```

Description RSORT@ sorts the REAL*4 array R by setting pointers from 1 to N in the array A in the same manner as CHSORT@.

DSORT@

Purpose To sort a REAL*8 array.

Syntax

```
SUBROUTINE DSORT@(A,D,N)
REAL*8 D(N)
INTEGER*4 A(N),N
```

Description DSORT@ sorts the REAL*8 array D by setting pointers from 1 to N in the array A in the same manner as CHSORT@.

5.

Error and exception handling

The routines described in this chapter fall into two main categories:

- Those which allow interpretation of error codes returned by other routines. Routines which fall into this category include ERR77 and DOS_ERROR_MESSAGE@. Where a routine returns an error code it is of course always good practice to check it for an acceptable value (usually zero).
- Those which allow control over the action taken in the event of a software-generated exception (such as an underflow, or a DOS critical error). Some "events", such as mouse movements and button presses, can be treated as exceptions in this context, and can be dealt with by the mechanisms described in this chapter.

See also chapter 28 in the *FTN77 User's Guide*.

ACCESS_DETAILS@ ❺

Purpose Get the details of the access violation.

Syntax
```
SUBROUTINE ACCESS_DETAILS@(ADDRESS, MODE, IC)
INTEGER*4 ADDRESS
LOGICAL*4 MODE
INTEGER*2 IC
```

Description This subroutine is used after an access violation has accured to ascertain the address that was being accessed when the exception occured. If this function is successful, *address* contains the address that was being accessed. *mode* is set to TRUE if the instruction was attempting to read from the address, FALSE if the instruction was attempting to write to the address.

ic is set to 0 on success, 1 on failure.

CLEAR_FLT_UNDERFLOW@ ❺

Purpose Clear a floating point underflow exception.

Syntax
```
SUBROUTINE CLEAR_FLT_UNDERFLOW@
```

Description Decode the instruction that caused the floating point underflow and clear the floating point underflow from the machine state.

DOS_ERROR_MESSAGE@

Purpose To get a DOS error message.

Syntax
```
SUBROUTINE DOS_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
INTEGER*2 ERROR_CODE
CHARACTER*(*) MESSAGE
```

Description Returns the DOS error string corresponding to the error number ERROR_CODE. The error numbers are augmented in the same way as for DOSERR@.

Example
```
CHARACTER*80 MESSAGE
CALL OPENR@('DATA',IH,ERROR_CODE)
CALL DOS_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
PRINT *,MESSAGE
```

DOSERR@

Purpose To print a DOS error message and exit when an error occurs.

Syntax
```
SUBROUTINE DOSERR@(ERROR_CODE)
INTEGER*2 ERROR_CODE
```

Description This routine does nothing if ERROR_CODE is zero, otherwise it prints the DOS error message corresponding to ERROR_CODE and exits from the program. It is typically used after system calls that use DOS and are normally

expected to succeed.

Example:

```
CALL OPENR@('DATA',IH,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
```

ERR77

Purpose To print a DOS error message and terminate a program when an error occurs.

Syntax

```
SUBROUTINE ERR77(MESSAGE,ERROR_CODE)
CHARACTER*(*) MESSAGE
INTEGER*2 ERROR_CODE
```

Description This routine has a null effect if ERROR_CODE is zero. Otherwise the string MESSAGE is printed, followed by the text of the DOS error indicated by ERROR_CODE. The program then terminates abnormally. This routine is normally used to test for DOS error following another system call, as in the example.

Example

```
CHARACTER*40 FILE
READ(*,'(A)')FILE
CALL OPENR@(FILE,HANDLE,ERROR_CODE)
CALL ERR77(FILE,ERROR_CODE)
. . .
```

ERROR@

Purpose To print a user defined error message and terminate a program.

Syntax

```
SUBROUTINE ERROR@(ERROR_MESSAGE)
CHARACTER*(*) ERROR_MESSAGE
```

Description This routine generates a user defined error condition. If the program is running under the debugger, then the message will be displayed in the error window. Otherwise, the error is printed out and the EXIT routine is called with a code of 1 (exit to DOS).

Example

```
CALL ERROR@('Too many data points for PLOT program')
```

EXCEPTION_ADDRESS@ ❺

Purpose To find the address of the intruction which generated the exception

Syntax
```
INTEGER*4 EXCEPTION_ADDRESS@
```

Description EXCEPTION_ADDRESS@ returns the address of the instruction that generated the exception event.

Return value Address of the instruction that generated the exception.

FORTRAN_ERROR_MESSAGE@

Purpose To get a Fortran error message.

Syntax
```
SUBROUTINE FORTRAN_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
INTEGER*2 ERROR_CODE
CHARACTER*(*) MESSAGE
```

Description Returns the error message corresponding to the Fortran run time error number ERROR_CODE. Some error messages are rather vague (e.g. Inconsistent call to routine) but when these errors occur the system produces a more informative error message.

Example
```
CHARACTER*80 MESSAGE
INTEGER*2 ERROR_CODE
OPEN(FILE='FRED',UNIT=6,IOSTAT=ERROR_CODE)
CALL FORTRAN_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
PRINT *,MESSAGE
```

GET_VIRTUAL_COMMON_INFO@ ❺

Purpose Get virtual common block details.

Syntax
```
 SUBROUTINE GET_VIRTUAL_COMMON_INFO@(NAME,BASE,
+     SIZE,COMMIT,AMOUT_COMMITTED)
 CHARACTER*(*) NAME
 INTEGER*4 BASE,SIZE,COMMIT,AMOUT_COMMITTED
```

Description When a program is linked using the virtual common (VC) option, all uninitialised

data (i.e. the BSS section) is removed from the executable and placed into virtual paged memory with default base address 0x20000000. Pages of memory are committed when necessary. GET_VIRTUAL_COMMON_INFO@ allows a user to determine how much memory is used and also provides other information.

The subroutine returns BASE, SIZE, COMMIT (amount of memory the system automatically commits) and AMOUNT_COMITTED (amount of memory already committed) for the allocated virtual common block.

Example

```
PROGRAM TVC1
INTEGER*4 BASE, SIZE, COMMIT, AMT_COMMIT
CALL GET_VIRTUAL_COMMON_INFO@('TVC1.EXE',BASE, SIZE, COMMIT,AMT_COMMIT)
PRINT*,' BASE = ', BASE
PRINT*,' SIZE = ', SIZE
PRINT*,' COMMIT = ', COMMIT
PRINT*,' AMT_COMMIT = ', AMT_COMMIT
END
```

JUMP@

Purpose To execute a non-local jump.

Syntax

```
SUBROUTINE JUMP@(LABEL)
COMPLEX*16 LABEL
```

Description This routine is described here since its most frequent use is in conjuction with SET_TRAP@. It takes a label generated by LABEL@ and jumps to the label. The label must exist in a still active routine, but this can be any distance down the call stack.

Example Consider a program designed to process several sets of data and to carry on even after errors had been diagnosed in earlier data sets:

```
      COMPLEX*16 RECOVER
      COMMON/ERR/RECOVER
      CALL LABEL@(RECOVER,*10)
10    CALL READ_DATA
      CALL PROCESS
      GOTO 10
      END
      SUBROUTINE ERROR(MESSAGE)
      COMPLEX*16 RECOVER
      COMMON/ERR/RECOVER
```

```
CHARACTER*(*) MESSAGE
CALL COU@(MESSAGE)
CALL JUMP@(RECOVER)
END
```

Subroutine ERROR could be called from anywhere inside PROCESS (even many layers down inside subroutine calls) to return to label 10 ready to process the next data set. The use of JUMP@ obviates the need to provide an explicit error exit path back to the main program. It is the user's responsibility to ensure that the LABEL is still accessible when JUMP@ is called, i.e. that the routine which is called LABEL@ has not yet exited.

LABEL@

Purpose To set a label for a non-local jump.

Syntax

```
SUBROUTINE LABEL@(LABEL,*)
COMPLEX*16 LABEL
```

Description This routine makes a label for use with JUMP@.

Example See JUMP@.

PERMIT_UNDERFLOW@

Purpose To switch off floating point underflow checking.

Syntax

```
SUBROUTINE PERMIT_UNDERFLOW@(PERMISSION)
LOGICAL*2 PERMISSION
```

Description If PERMISSION is .TRUE. then this routine forces subsequent floating point underflows to return zero. If PERMISSION is .FALSE. then subsequent underflows will force a program fault.

PRERR@

Purpose To print the error message associated with a given error code.

Syntax
```
SUBROUTINE PRERR@(ERROR_CODE,STRING)
INTEGER*2 ERROR_CODE
CHARACTER*(*) STRING
```

Description This routine does nothing if ERROR_CODE is zero, otherwise it prints the user-supplied string STRING followed by the system error message corresponding to ERROR_CODE. The routine returns normally and program execution continues. ERROR_CODE will normally be a value returned by an earlier call to a routine that could generate a system error condition.

QUIT_CLEANUP@

Purpose To print a message and exit from a program with Control-break

Syntax
```
SUBROUTINE QUIT_CLEANUP@(MESSAGE)
CHARACTER*(*) MESSAGE
```

Description This routine uses SET_TRAP@ to trap Control-break. The system responds to Control-break by printing the given message and returning to DOS. This routine provides a simple way to enable programs to be terminated early in a graceful fashion.

Example
```
CALL QUIT_CLEANUP@('Quit pressed - program abandoned')
```

RESTORE_DEFAULT_HANDLER@ ❺

Purpose Remove a user defined exception handler.

Syntax
```
SUBROUTINE RESTORE_DEFAULT_HANDLER@(EXCEPTION)
INTEGER*4 EXCEPTION
```

Description Remove the default exception handler for the exception event given by EXCEPTION and re-install the default handler.

RUNERR@

Purpose To print the run-time error corresponding to a given IOSTAT value.

Syntax
```
SUBROUTINE RUNERR@(ISTAT)
INTEGER*2 ISTAT
```

Description RUNERR@ prints an error message on the screen corresponding to a given IOSTAT value ISTAT. These are the error messages listed in chapter 27.

SET_DISK_ERRORS@

Purpose To control the critical event handler.

Syntax
```
SUBROUTINE SET_DISK_ERRORS@(L)
LOGICAL*2 L
```

Description If L is .TRUE. critical errors will return an error code and *not* put up the "Abort, Retry, Ignore" message. If L is .FALSE. the message will appear.

SET_TRAP@ ❶

Purpose To trap a given event.

Syntax
```
SUBROUTINE SET_TRAP@(TRAP,OLDADDR,TYPE)
INTEGER*4 OLDADDR
INTEGER*2 TYPE
EXTERNAL TRAP
```

Description This routine assigns an address TRAP to an event number TYPE. This is the address of a routine that is to be called when the given event occurs. The former address for the event is returned in OLDADDR (for nested traps). TYPE can currently take one of the numbers in the table below.

TYPE	Event
0	Trap CONTROL BREAK
1	Trap Floating Point faults
2	Trap on every key press or release
3	Trap on alarm clock interrupt
4	Trap on mouse event
5	Trap on reaching page reserve
6	Trap on user-defined event
7	Trap on general protection exception
8	Trap on invalid opcode

The routine which handles the trap must save the register set, and so requires some assembler programming, see page 206 in the *FTN77 User's Guide*.

TRAP_EXCEPTION@

Purpose Install a user defined exception handler

Syntax
```
INTEGER*4 FUNCTION TRAP_EXCEPTION@(EXCEPTION,ROUTINE)
INTEGER*4 EXCEPTION, ROUTINE
```

Description The subroutine ROUTINE is installed as the default method of handling the event specified by EXCEPTION.

Return value If EXCEPTION is a valid exception event, the location of the previous handler is returned. 0 is returned if EXCEPTION is an invalid exception code

Example

```
SUBROUTINE UNDERFLOW_HANDLER
       :
END

OLD = TRAP_EXCEPTION@(UNDERFLOW, UNDERFLOW_HANDLER)
```

UNDERFLOW_COUNT@

Purpose To get the number of floating point underflows.

Syntax

```
SUBROUTINE UNDERFLOW_COUNT@(COUNT)
INTEGER*4 COUNT
```

Description COUNT is returned as the number of underflows that have occurred since the start of the program.

6.

File-manipulation

FTN77 offers a wide variety of file manipulation routines. If at all possible these should be used in preference to attempting explicit system calls with SVC/3 for example. *Routines which take operating system file handles as arguments must only be used with file handles obtained with one of the file opening routines detailed in this chapter*. The reading and writing routines use a buffer to eliminate unnecessarily frequent switches between real and protected mode, and to improve the performance in general. These buffers are cleared when a file is closed or when a program terminates and returns to the operating system. As a result of this, a file which has been open for writing may give an error (e.g. full disk) as it is closed with CLOSEF@. In all cases where a routine returns an error code, this may be interpreted by calling a routine such as DOSERR@.

ATTACH@

Purpose To set the current directory.

Syntax

```
SUBROUTINE ATTACH@(PATH,ERROR_CODE)
CHARACTER*(*) PATH
INTEGER*2 ERROR_CODE
```

Description PATH should be the pathname of a directory (e.g. C:\PROJECT). ATTACH@ makes this the current directory, switching disks if necessary. ERROR_CODE is returned with a non-zero system error code if it fails.

Example

```
CHARACTER*50 PATH
CALL COUA@('Where do you want to be? ')
READ(*,'(A)')PATH
CALL ATTACH@(PATH,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
```

```
CALL COU@('OK - that is where you are! ')
. . .
```

CLOSEF@

Purpose To close a file.

Syntax
```
SUBROUTINE CLOSEF@(HANDLE,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
```

Description CLOSEF@ closes a file opened by OPENR@, OPENW@ or OPENRW@. ERROR_CODE is returned with a non-zero system error code if it fails.

Example See OPRNRW@

CLOSEFD@

Purpose To close and delete a file.

Syntax
```
SUBROUTINE CLOSEFD@(HANDLE,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
```

Description CLOSEFD@ is the same as CLOSEF@ but CLOSEFD@ also deletes the file from the disc. This is useful for temporary files.

CLOSEV@ ❷

Purpose Closes a file opened with OPENV@

Syntax
```
SUBROUTINE CLOSEV@(SELECTOR,ERROR_CODE)
INTEGER*2 ERROR_CODE,SELECTOR
```

Description CLOSEV@ closes a file and removes the corresponding memory segment. It is important to ensure that no segment register still holds the selector before calling this routine. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

CURDIR@

Purpose To get the current directory.

Syntax
```
CHARACTER*(*) FUNCTION CURDIR@()
```

Return value CURDIR@ returns the fully qualified pathname of the current directory.

Example
```
CHARACTER*50 CURDIR@
PRINT *,'You are currently in ',CURDIR@()
```

CURRENT_DIR@ ❺

Purpose To get the current directory.

Syntax
```
SUBROUTINE CURRENT_DIR@(DIR,ERROR_CODE)
CHARACTER*(*) DIR
INTEGER*2 ERROR_CODE
```

Description This routine is obsolete. Use CURDIR@ instead.

Return value Returns the name of the current working directory in DIR, or a non-zero error code ERROR_CODE if failed. ERROR_CODE is returned as ERANGE if the variable DIR is not of sufficient length. (ERANGE is defined in the include file *errno.ins*.)

DIRENT@

Purpose To obtain directory information.

Syntax
```
 SUBROUTINE DIRENT@(PAT,ATTRIBUTE,RESULT,RESULT_ATTRIBUTE,
+  RESULT_DATE,RESULT_TIME,FILE_SIZE,ERROR_CODE)
 CHARACTER*(*) PAT,RESULT
 INTEGER*2 ATTRIBUTE,RESULT_ATTRIBUTE,RESULT_DATE,
+  RESULT_TIME,ERROR_CODE
 INTEGER*4 FILE_SIZE
```

Description DIRENT@ returns directory information for files selected by PAT (e.g. A:*.FOR). That is, each call of the routine searches for a single file in the directory and with the extension implied by PAT. The attribute of the first file

returned is selected by setting ATTRIBUTE to one of the following values:

0	Return a normal file
2	Return a hidden file
4	Return a system file
6	Return a volume name
16	Return a subdirectory

The name of the file that has been found is returned in RESULT. Other file information is returned in RESULT_ATTRIBUTE, RESULT_DATE, RESULT_TIME and FILE_SIZE. The file attributes are returned in DOS coded form using bits 0 to 5 of the result. The date and time are returned in DOS compressed format.

After the first call of the routine, ATTRIBUTE should be set to -1 in order to continue the search for another file with the same attribute as before. When no more files can be found, ERROR_CODE is returned with the corresponding system error code.

Notes A sequence of calls to DIRENT@ with a given PAT must not be interrupted by a call to DIRENT@ with a different PAT.

The FILES@ routine has a simpler interface and is usually preferred to this routine.

Example

```
      CHARACTER*20 PAT,FILE
      INTEGER*2 ATTR,DATE,TIME,EC
      INTEGER*4 SIZE
      CALL COUA@('Input directory pattern:')
      READ 100,PAT
      EC=0
      WHILE (EC.EQ.0) DO
        CALL DIRENT@(PAT,0,FILE,ATTR,DATE,TIME,SIZE,EC)
        IF (EC.EQ.0) PRINT 101,FILE,ATTR,DATE,TIME,SIZE
C       Do not call DIRENT@(PAT2,...) from here
      ENDWHILE
 100  FORMAT(A)
 101  FORMAT(A,I6,I6,I6,I6)
      END
```

EMPTY@

Purpose To clear a file for writing.

Syntax
```
SUBROUTINE EMPTY@(HANDLE,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
```

Description EMPTY@ clears the file open with file handle HANDLE (which must not be open for reading only). ERROR_CODE is returned as zero for success or with a system error code if it fails.

ERASE@

Purpose To delete a file.

Syntax
```
SUBROUTINE ERASE@(FILE,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*2 ERROR_CODE
```

Description ERASE@ deletes a file. The file name may be a local name, for example: FOO.TXT, or a fully qualified pathname, for example, C:\PROJECT\JUNK.TXT. ERROR_CODE is returned as zero for success or with the system error code.

Example
```
CALL ERASE@('USELESS.DAT',ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
```

FEXISTS@ ❺

Purpose To search for a file with a given path name or wildcard.

Syntax
```
LOGICAL*4 FUNCTION FEXISTS@(PATH, ERROR_CODE)
CHARACTER*(*) PATH
INTEGER*4 ERROR_CODE
```

Return value FEXISTS@ returns a logical value which is .TRUE. if the name supplied in PATH is that of a file which does exist, or is a wildcard which matches one file only. It returns .FALSE. if such a file does not exist, or if an error occurs in which case ERROR_CODE returns a non-zero system error code.

FILE_EXISTS@ ❺

Purpose To search for a file with a given path name or wildcard.

Syntax
```
LOGICAL*4 FUNCTION FILE_EXISTS@(PATH)
CHARACTER*(*) PATH
```

Description This function is obsolete. Use FEXISTS@ instead.

FILE_EXISTS@ returns a logical value which is .TRUE. if the name supplied in PATH is that of a file which does exist, or is a wildcard which matches one file only. It returns .FALSE. if such a file does not exist, or if any kind of error occurs.

FILE_SIZE@

Purpose To get the size of FILE in bytes.

Syntax
```
SUBROUTINE FILE_SIZE@(FILE,SIZE,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*4 SIZE
INTEGER*2 ERROR_CODE
```

FILE_TRUNCATE@

Purpose To truncate an open file at its current position.

Syntax
```
SUBROUTINE FILE_TRUNCATE@(HANDLE,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
```

Description This routine uses the handle of a file that has already been opened by OPENW@ or OPENRW@ and truncates the file at the current writing position. ERROR_CODE is returned as zero if the process is carried out successfully, otherwise ERROR_CODE returns a system error code.

FILEINFO@

Purpose To get information about a specified file.

Syntax

```
 SUBROUTINE FILEINFO@(PATH,MODE,DEV,RDEV,
+ NLINK,SIZE,ATIME,MTIME,CTIME,ERROR_CODE)
 CHARACTER*(*) PATH
 INTEGER*2 MODE,DEV,RDEV,NLINK,ERROR_CODE
 INTEGER*4 SIZE,ATIME,MTIME,CTIME
```

Description Returns information about the file specified by PATH. This routine can be used to return the size of a file in SIZE and the date and time that the file was last accessed in ATIME. The returned value of ATIME can then be supplied to TOTIME@, TOEDATE@ etc.. ERROR_CODE is returned as zero if the process is carried out successfully, otherwise ERROR_CODE returns a system error code.

Note Arguments that do not appear in the above description are redundant in this operating system environment.

FILES@

Purpose To obtain directory information.

Syntax

```
 SUBROUTINE FILES@(PAT,N,NMAX,FILES,ATTR,DATE,TIME,
+  FILE_SIZE)
 CHARACTER*(*) PAT,FILES(NMAX)
 INTEGER*2 N,NMAX
 INTEGER*2 ATTR(NMAX),DATE(NMAX),TIME(NMAX)
 INTEGER*4 FILE_SIZE(NMAX)
```

Description Returns directory information for files selected by PAT (e.g. `A:\*.FOR`). N is returned as the number of file names returned. If N is equal to NMAX there may be more matches which could not be returned. The remainder of the arrays return information about the files. Hidden files and directories are returned by this routine together with ordinary files. Such files may be distinguished by using the DOS file attribute returned in the ATTR array. The date and time are returned in the DOS compressed format.

Example See SET_FILE_ATTRIBUTE@.

FPOS@

Purpose To reposition a file.

Syntax
```
SUBROUTINE FPOS@(HANDLE,POSITION,NEW_POSITION,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
INTEGER*4 POSITION,NEW_POSITION
```

Description FPOS@ attempts to reposition an open file with the given HANDLE to the given POSITION. NEW_POSITION is returned as either the requested POSITION or as the position of end-of-file, whichever is less. ERROR_CODE is returned as zero or with a system error code if it fails.

Notes If the input value of POSITION is supplied as a constant, it is usually necessary to force its length to 4 bytes (e.g. 0L for the beginning of the file).

FPOS_EOF@ ❺

Purpose To move the file pointer to end-of-file

Syntax
```
SUBROUTINE FPOS_EOF@(DESC,NEW_POSITION,ERROR_CODE)
INTEGER*2 DESC,ERROR_CODE
INTEGER*4 NEW_POSITION
```

Description Move the file pointer associated with the file open on DESC to end-of-file. NEW_POSITION is the new value of the file pointer.

GET_FILE_DATE_TIME_STAMP@ ❷

Purpose To get the DOS date and time stamp for a particular file.

Syntax
```
SUBROUTINE GET_FILE_DATE_TIME_STAMP@(FILE,DATE,TIME)
CHARACTER*(*) FILE
INTEGER*2 DATE,TIME
```

Description This routine gets the date and time stamp for FILE. DATE and TIME will be returned with the value -1 if the file does not exist.

GET_FILES@ ❺

Purpose To get a list of files in the current working directory.

Syntax

```
 SUBROUTINE GET_FILES@(WILDCARD,FILES,
+  MAXFILES,NFILES,ERROR_CODE)
 CHARACTER*(*) WILDCARD,FILES(MAXFILES)
 INTEGER*2 MAXFILES,NFILES,ERROR_CODE
```

Description Returns a list of all the files in the current working directory which can be matched by WILDCARD.

Returns error code ERROR_CODE as ERANGE if FILES is not big enough, but the entries which are stored in FILES will be valid.

GET_PATH@

Purpose To get the fully qualified pathname.

Syntax

```
SUBROUTINE GET_PATH@(HANDLE,RESULT,ERROR_CODE)
CHARACTER*(*) RESULT
INTEGER*2 HANDLE,ERROR_CODE
```

Description GET_PATH@ returns the pathname of the file open on file handle HANDLE. This works regardless of whether a local or global name was used when the file was originally opened. ERROR_CODE is returned as zero for success or it is returned as a system error code.

Example

```
CHARACTER*100 FULL_PATH
CALL OPENR@('MYDATA',HANDLE,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
CALL GET_PATH@(HANDLE,FULL_PATH,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
PRINT *,FULL_PATH
. . .
```

GET_PATHV@ ❷

Purpose To get the fully qualified pathname.

Syntax
```
SUBROUTINE GET_PATHV@(SEGMENT,RESULT,ERROR_CODE)
CHARACTER*(*) RESULT
INTEGER*2 SEGMENT,ERROR_CODE
```

Description GET_PATHV@ returns the pathname of a file opened with OPENV@ to memory segment SEGMENT. This works regardless of whether a local or global name was used when the file was originally opened. ERROR_CODE is returned as zero or contains a system error code.

MKDIR@

Purpose To create a new system directory.

Syntax
```
SUBROUTINE MKDIR@(DIR,ERROR_CODE)
CHARACTER*(*) DIR
INTEGER*2 ERROR_CODE
```

Description The argument DIR can be either the local name of a directory, or the full path name. In either case, if the directory cannot be created for any reason, a non-zero system error code will be returned.

Example
```
CALL MKDIR@('C:\ACCOUNTS',IC)
CALL DOSERR@(IC)
```

OPENR@

Purpose To open a file for reading.

Syntax
```
SUBROUTINE OPENR@(FILE,HANDLE,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*2 HANDLE,ERROR_CODE
```

Description This routine opens the given file FILE for reading and returns the file handle HANDLE for use with other file handling routines in this chapter. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it

is returned with the relevant system error code.

Notes HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see the *User's Guide* for further details).

Example See OPENRW@

OPENRW@

Purpose To open a file for reading or writing.

Syntax

```
SUBROUTINE OPENRW@(FILE,HANDLE,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*2 HANDLE,ERROR_CODE
```

Description This routine opens a file FILE for reading or writing and returns the file handle HANDLE for use with other file handling routines. If the file does not exist it is created, however an existing file is *not* emptied and may be over-written at the current position. If the intended action depends on whether or not a given file exists, then a prior call to OPENR@ can be used to test if it does exist. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

Notes HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see the *User's Guide* for further details).

Example

```
C     Run the program and list TEST.DAT after each run.
      INTEGER*2 HANDLE,ERROR_CODE
      INTEGER*4 BYTES
      CHARACTER*80 LINE
      CALL OPENR@('TEST.DAT',HANDLE,ERROR_CODE)
      IF(ERROR_CODE.NE.0) THEN
        CALL OPENW@('TEST.DAT',HANDLE,ERROR_CODE)
        CALL WRITEFA@('Test data.....',HANDLE,ERROR_CODE)
      ELSE
        CALL CLOSEF@(HANDLE,ERROR_CODE)
        CALL OPENRW@('TEST.DAT',HANDLE,ERROR_CODE)
        LINE=' '
        CALL READFA@(LINE,HANDLE,BYTES,ERROR_CODE)
        CALL DOSERR@(ERROR_CODE)
```

```
    WRITE(*,*) LINE
    CALL WRITEFA@('More info.....',HANDLE,ERROR_CODE)
  ENDIF
  END
```

OPENV@ ❷

Purpose To open a file for reading.

Syntax

```
SUBROUTINE OPENV@(FILE,SELECTOR,NB,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*2 ERROR_CODE,SELECTOR
INTEGER*4 NB
```

Description This routine opens a file FILE for reading only to form a separate memory segment. The selector of that segment is returned in SELECTOR, and NB is returned as the size of the file in bytes. The file is read (as needed) using the paging mechanism. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

OPENW@

Purpose To open a file for writing.

Syntax

```
SUBROUTINE OPENW@(FILE,HANDLE,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*2 HANDLE,ERROR_CODE
```

Description This routine opens a file FILE for writing, by creating a file or emptying the file if it already exists. It returns the file handle HANDLE for use with other file handling routines. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

Notes HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see the *User's Guide* for further details).

Description See OPENRW@

READF@

Purpose To read binary data from a file.

Syntax

```
 SUBROUTINE READF@(DATA,HANDLE,NBYTES,NBYTES_READ,
+ ERROR_CODE)
 CHARACTER*(*) DATA
 INTEGER*2 HANDLE,ERROR_CODE
 INTEGER*4 NBYTES,NBYTES_READ
```

Description This routine reads NBYTES of data from an open file with a given HANDLE. ERROR_CODE is returned as zero for success or a system error code. If end of file is reached, NBYTES_READ is returned as -1, with an ERROR_CODE of zero. Also NBYTES_READ may be returned as less than NBYTES. This routine should be used on binary data.

Notes If the input value of NBYTES is supplied as a constant, it is usually necessary to force its length to 4 bytes (append L to the decimal value).

READFA@

Purpose To read ASCII text from a file.

Syntax

```
SUBROUTINE READFA@(DATA,HANDLE,NBYTES_READ,ERROR_CODE)
CHARACTER*(*) DATA
INTEGER*2 HANDLE,ERROR_CODE
INTEGER*4 NBYTES_READ
```

Description READFA@ reads a line of text from an open file with a given HANDLE. Tabs are expanded as necessary. ERROR_CODE is returned as zero for success or with a system error code. If end of file is reached, NBYTES_READ is returned as -1, with an ERROR_CODE of zero.

Example See OPENRW@

RENAME@

Purpose To rename a file.

Syntax
```
SUBROUTINE RENAME@(FILE1,FILE2,ERROR_CODE)
CHARACTER*(*) FILE1,FILE2
INTEGER*2 ERROR_CODE
```

Description RENAME@ renames FILE1 as FILE2 in exactly the same way as the DOS RENAME command.

RFPOS@

Purpose To get the position of a file.

Syntax
```
SUBROUTINE RFPOS@(HANDLE,POSITION,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
INTEGER*4 POSITION
```

Description This routine returns the POSITION of the file open on the given HANDLE. ERROR_CODE is returned as zero or with a system error code if it fails.

SELECT_FILE@ ❷

Purpose To select from a displayed list of files.

Syntax
```
SUBROUTINE SELECT_FILE@(PATTERN,RESULT,*)
CHARACTER*(*) PATTERN,RESULT
```

Description SELECT_FILE@ takes a file pattern and displays all files that correspond to that pattern in a window. A file may be selected using the up and down cursor keys (or the mouse) and pressing Enter or a mouse button on the right one. If no files are found or the user presses Esc to indicate that he does not choose any of the files on display, then RESULT is set to spaces and the alternate return is taken. This routine makes it easy to provide an interactive startup interface to a program as illustrated in the example.

Example
```
CHARACTER*80 FILE
CALL SELECT_FILE@('C:\TESTS\*.DAT',FILE,*10)
OPEN(FILE=FILE,UNIT=5,STATUS='READONLY')
```

```
      . . .
10    PRINT *,'NO FILE SELECTED'
      END
```

SET_FILE_ATTRIBUTE@

Purpose To set a file attribute.

Syntax

```
SUBROUTINE SET_FILE_ATTRIBUTE@(FILE,IAT,ERROR_CODE)
CHARACTER*(*) FILE
INTEGER*2 IAT,ERROR_CODE
```

Description This routine sets the attributes of the file FILE to IAT. ERROR_CODE is returned as zero for success or it is the system error code. This routine is useful for performing such tasks as changing the read-only status of a file, hiding or revealing a file, reseting the backup bit etc..

The following program will read a file name (possibly including wild cards) from the terminal and make the files read-only.

Example

```
      CHARACTER*120 FILE,FILES(1000),CMNAM
      INTEGER*2 ATTR(1000),DATE(1000),TIME(1000)
      INTEGER*4 FILE_SIZE(1000)
      CALL FILES@(CMNAM(),N,1000,FILES,ATTR,DATE,TIME,FILE_SIZE)
      DO 1 I=1,N
        CALL SET_FILE_ATTRIBUTE@(FILES(I),OR(ATTR(I),1),IC)
        CALL DOSERR@(IC)
1     CONTINUE
      END
```

SET_SUFFIX@

Purpose To change the extension of a given file name.

Syntax

```
SUBROUTINE SET_SUFFIX@(FILENAME,SUFFIX,L)
CHARACTER*(*) FILENAME
CHARACTER*3 SUFFIX
LOGICAL*2 L
```

Description SET_SUFFIX@ changes the file extension of a given file with name

FILENAME. SUFFIX is the new extension required, given without the period (“.”). The value L will be set to .TRUE. if the file had an extension that was not SUFFIX. L will be given the value .FALSE. if the file had the same or no extension.

Example

```
      A='c:\ftn77.dir\file.dat'
      CALL SET_SUFFIX@(A,'ASC',L)
C  At this point A contains 'c:\ftn77.dir\file.ASC'
C  and L contains .TRUE.
```

SET_SUFFIX1@

Purpose To add an extension to a given file name.

Syntax

```
SUBROUTINE SET_SUFFIX1@(FILENAME,SUFFIX,L)
CHARACTER*(*) FILENAME
CHARACTER*3 SUFFIX
LOGICAL*2 L
```

Description SET_SUFFIX1@ will add a file-extension SUFFIX to the string FILENAME containing a filename if none is present. The filename will be left as it is if the filename already contains an extension. The extension should be given without the period (“.”). The value L will be set to .TRUE. if the file had an extension that was not SUFFIX. L will be given the value .FALSE. if the filename had the same or no extension.

Example

```
      A='c:\ftn77.dir\file.dat'
      CALL SET_SUFFIX1@(A,'ASC',L)
C  At this point A contains 'c:\ftn77.dir\file.dat'
C  and L contains .TRUE.
```

TEMP_FILE@

Purpose To provide a unique name for a file.

Syntax
```
SUBROUTINE TEMP_FILE@(FILEX,ERROR_CODE)
CHARACTER*(*) FILEX
INTEGER*2 ERROR_CODE
```

Description TEMP_FILE@ provides a name which may be used for the creation of a temporary file. This name (of the form F$dddddd.TMP where d is a digit) is different from all the file names within the current directory. It is important to note that this routine does not create or open a file.

TEMP_PATH@

Purpose To get a suitable name for a temporary file.

Syntax
```
SUBROUTINE TEMP_PATH@(PATH)
CHARACTER*(*) PATH
INTEGER*2 ERROR_CODE
```

Description This routine is obsolete. Use TEMP_FILE@ instead.

TEMP_PATH@ makes up a path name for a temporary file. The file name component is created in the same way as for TEMP_FILE@. The directory is that given by the TMPDIR environment variable (or "\TMP" if this is not set). Note that this routine does not actually open the file.

WILDCHECK@ ❺

Purpose To check for the matching of a file name with a wild card.

Syntax
```
LOGICAL*2 FUNCTION WILDCHECK@(WILDCARD,NAME)
CHARACTER*(*) WILDCARD,NAME
```

Description Returns .TRUE. if NAME can be matched with WILDCARD, .FALSE. otherwise (including when the syntax of WILDCARD or NAME is invalid).

WRITEF@

Purpose To write binary data to a file.

Syntax

```
SUBROUTINE WRITEF@(DATA,HANDLE,NBYTES,ERROR_CODE)
CHARACTER*(*) DATA
INTEGER*2 HANDLE,ERROR_CODE
INTEGER*4 NBYTES
```

Description Writes NBYTES of binary data DATA to the file with the given handle. ERROR_CODE is returned as zero for success or a system error code on failure. No data compression on insertion of control characters is performed.

Notes If the input value of NBYTES is supplied as a constant, it is usually necessary to force its length to 4 bytes (append L to the decimal value).

WRITEFA@

Purpose To write a line of data to an ASCII file.

Syntax

```
SUBROUTINE WRITEFA@(DATA,HANDLE,ERROR_CODE)
CHARACTER*(*) DATA
INTEGER*2 HANDLE,ERROR_CODE
```

Description WRITEFA@ writes DATA to an open file with a given HANDLE. A carriage return/linefeed is added to the end of the data. ERROR_CODE is returned as zero for success, otherwise it returns the system error code.

Example

```
INTEGER*2 HANDLE,ERROR_CODE
CALL OPENW@('TEST.DAT',HANDLE,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
CALL WRITEFA@('Test data.....',HANDLE,ERROR_CODE)
CALL WRITEFA@('More info.....',HANDLE,ERROR_CODE)
END
```

7.

Graphics drawing

Introduction

FTN77 supports screen, printer and HP-GL (plotter) compatible graphics. The printer, plotter and "virtual screen" are auxiliary devices (for convenience these are described separately in the next two chapters). You can only open one auxiliary device at a time. If you open another whilst one is open then the old device will be closed. All the graphics output produced by the routines described in this chapter will be directed to the auxiliary device if one is open, otherwise the output will be directed to the screen. Even when an auxiliary device is open, however, it is still possible to use the routines in this chapter which only relate to the screen (e.g. EGA@, TEXT_MODE@, CLEAR_SCREEN@).

Palette registers and 16 colour graphics.

The colour number which is used for 16 colour graphics is a Palette Register Number (PRN) in the range 0..16. Each register takes a Palette Register Value (PRV) in the range 0..63 which defines the colour.

In EGA mode the PRV specifies the colour directly, with the 6 least significant bits having the following symbolic meaning

bit	5	4	3	2	1	0
	red	green	blue	Red	Green	Blue
	one third intensity			two thirds intensity		

The default PRVs are given by:

PRN	Colour	PRV
0	Black	0
1	Blue	1
2	Green	2
3	Cyan	3
4	Red	4
5	Magenta	5
6	Brown	20
7	White	7
8	Dark Grey	56
9	Light Blue	57
10	Light Green	58
11	Light Cyan	59
12	Light Red	60
13	Light Magenta	61
14	Yellow	62
15	Intense White	63
16	Black	0

PRNs 1..15 represent available colours for pixels, lines, etc.

PRN 0 provides the default background colour.

PRN 16 specifies the screen border (overscan) colour.

PRN 7 provides the default text colour attribute.

The PRVs can be changed using SET_PALETTE@ and SET_ALL_PALETTE_REGISTERS@.

In 16 colour VGA mode the PRV specifies the colour indirectly by providing a pointer in the range 0..255 to certain DAC (digital-to-analogue converter) registers. The DAC registers provide a means of defining 256 colours (although only 16 different colours can appear on the screen at any one time). Each DAC register contains three values representing the red, green and blue intensities in the range 0..63.

The default palette register values are the same as for EGA mode and the default values for the DAC registers 0..63 emulate EGA mode. For example, palette register 7 has value 20 (brown) corresponding to two thirds intensity red with one third intensity green. The default values in DAC register 20 are (R=42; G=21; B=0).

256 colour graphics.

The colour number which is used for 256 colour graphics is a DAC register number in the range 0..255 with the same construction as for 16 colour modes. In this case all 256 colours can appear simultaneously on the screen.

The DAC values can be changed using SET_VIDEO_DAC@ and SET_VIDEO_DAC_BLOCK@.

Polygon filling

A polygon is a closed polygonal line, i.e. a line joining an ordered set of vertices. The edges of the polygon may intersect and polygons may be combined. There is no limit to either the complexity of a polygon (many thousands of intersecting edges are possible), or to the number of polygon definitions that you have currently defined, beyond the memory space available on your machine.

A polygon is filled by colouring all points in its interior. A point is on the interior of a polygon if an odd number of boundaries have to be crossed to reach the exterior of the polygon. Specifying the vertices of the polygon in a different order may, therefore, produce a different fill result.

A polygon definition is created in memory by making a call to CREATE_POLYGON@. This call will return a polygon "handle". Use this handle in all subsequent calls that affect the polygon. The position of the polygon is part of the polygon definition. However, the polygon definition may be altered by making a call to MOVE_POLYGON@ to shift the position relatively.

Some polygons contain sub-polygons. For example, an area may have several holes in it, or its boundary may be intersected by other polygons. Every sub-polygon in the polygon should be created and the definitions combined to make a new polygon with COMBINE_POLYGON@. The original definitions will remain and be available. Subsequent operations on these polygons will have no effect whatever on the combined polygon, which is now an entirely separate and distinct entity. You will be given a new polygon handle for the combined polygon.

The polygon may be filled by making a call to FILL_POLYGON@. The polygon definition will remain and still be available for later use. When the useful life of a polygon definition has expired it may be deleted by calling DELETE_POLYGON_DEFINITION@. This releases memory for future use.

Text attributes

Text written to a graphics device using DRAW_TEXT@ has attributes which can be selected by using SET_TEXT_ATTRIBUTE@. These are:

FONT: the shape of the characters.

SIZE: the replication factor from the original definition of the characters.

ROTATION: this is the direction of the character string; the string is rotated about the bottom left corner of the first character in the string.

ITALIC: this is a shear transformation applied to the character.

Both of the raster devices (i.e. screen and graphics printer) use the same fonts. The plotter has its own built-in fonts which are different from those used by the raster devices. The text attributes are available globally (i.e. when you have selected a font, size, rotation and italic, these do not have to be re-selected) and are used in every subsequent call to DRAW_TEXT@.

These attributes should be used with some caution on the raster devices when using bit mapped fonts. Whilst increasing the size merely increases the chunkiness of the text, pixel rounding effects may make the text untidy for certain rotations and italicisations. The situation might be improved in these cases by altering the size of the text.

Here are a few pertinent hints:

For rotations in multiples of 90 degrees:

Keep the size of the characters an integer.

For rotations in multiples of 45 degrees:

Multiply an integer size by $\sqrt{2}$ and use this. i.e. instead of using size 5.0 use size $5*\sqrt{2}=7$. For best effects the size should be divisible by 1.4.

For other rotations:

A certain amount of thought and experimentation is necessary. Avoid character sizes below 2.0.

Additional fonts

In addition to the fonts provided with the display adapters and the plotter, a set of proportionally spaced fonts has been made available. These fonts have characters of varying widths, unlike the fixed (or monospaced) fonts provided with the display adapters. You should also be aware that they are stroke fonts and more suitable to plotter output than to screen or printer output.

A list of these fonts is given below:

Font No.	Description	Style	Weight
101	Simplex Roman	sans serif	
102	Duplex Roman	sans serif	bold
103	Simplex Greek	sans serif	
104	Complex Roman	seriffed	
105	Complex Italic	seriffed	
106	Triplex Roman	seriffed	bold
107	Triplex Italic	seriffed	bold
108	Simplex Script		
109	Complex Script		bold
110	Complex Greek	seriffed	
111	Complex Cyrillic	seriffed	
112	Gothic English		
113	Gothic German		
114	Gothic Italian		

Coordinate systems

Every device has its own sense and range of coordinates. All devices have x=0 on the left of the screen or page but in the case of the screen and printer y=0 is on the top whereas for the plotter it is on the bottom. Due to differences between the horizontal and vertical sizes of the pixels on the raster devices, a circle in coordinate space will not appear as a true circle on the screen or page. You should take this into account when designing your software. The following table illustrates the situation for the raster devices:

Device	Position of (0,0)	Range	Resolution	Pixel Size Ratio Hor:Ver
EGA	top left	(0,0)-(639,349)	640x350	1.37:1.0
VGA	top left	(0,0)-(639,479)	640x480	1.0:1.0
printer	top left	(0,0)-(959,575)	960x576	5.0:3.0
virtual-screen	top left	(0,0)-(xx,yy)	xx+1, yy+1	

Note that the printer referred to above is an Epson 9 pin dot matrix (or compatible) printer. This is the default type.

The following figure illustrates the situation for plotters:

HP7550A:

Paper Size	Position of (0,0)	Range	Resolution
A4	bottom left	(0,0)-(10870,7600)	0.025mm
A3	bottom left	(0,0)-(15970,10870)	0.025mm
A	bottom left	(0,0)-(10170,7840)	0.025mm
B	bottom left	(0,0)-(16450,10170)	0.025mm

HP7475A:

Paper Size	Position of (0,0)	Range	Resolution
A4	bottom left	(0,0)-(11040,7721)	0.025mm
A3	bottom left	(0,0)-(16158,11040)	0.025mm
A	bottom left	(0,0)-(10365,7962)	0.025mm
B	bottom left	(0,0)-(16640,10365)	0.025mm

CLEAR_SCREEN@

Purpose To clear the screen.

Syntax
```
SUBROUTINE CLEAR_SCREEN@
```

Description CLEAR_SCREEN@ clears the screen to the default background colour in either text or graphics mode and sets the text cursor position to (0,0).

See also CLEAR_SCREEN_AREA@, FILL_RECTANGLE@.

CLEAR_SCREEN_AREA@

Purpose To clear a rectangular area of the screen.

Syntax
```
SUBROUTINE CLEAR_SCREEN_AREA@(IX1,IY1,IX2,IY2,ICOL)
INTEGER*2 IX1,IX2,IY1,IY2,ICOL
```

Description This routine clears an area of the graphics screen to colour number ICOL. (IX1,IY1) are the coordinates of the top left corner of the rectangle whilst (IX2,IY2) are the coordinates of the bottom right. If any portion of the area is

off screen or the screen is not in graphics mode then no action will be taken.

See also CLEAR_SCREEN@, FILL_RECTANGLE@.

COMBINE_POLYGONS@

Purpose To get the handle for a combination of polygons.

Syntax

```
SUBROUTINE COMBINE_POLYGONS@(HANDLE_ARRAY,N,HANDLE,
+  ERROR_CODE)
INTEGER*2 HANDLE_ARRAY(N),N,HANDLE,ERROR_CODE
```

Description This routine combines N polygons with handles HANDLE_ARRAY(1) to HANDLE_ARRAY(N), to make one complex polygon and returns the handle HANDLE for it. All of the polygons must be valid otherwise no action will be taken and the value ERROR_CODE=2 is returned. Other error codes are the same as those for CREATE_POLYGON@. All of the original polygons remain defined and available for use.

See also FILL_POLYGON@, MOVE_POLYGON@, DELETE_POLYGON_DEFINITION@.

Example

```
C To see what the effect of combining polygons is,
C take a large rectangle and combine it with a smaller
C rectangle. A hole will be created in the larger rectangle.
C It is important to realise that the result does not depend
C on the order in which the polygons are combined.
      PROGRAM BOX7
      INTEGER*2 X1(5),Y1(5),X2(5),Y2(5),K,HANDLE(3),
     + ERROR_CODE
C Data for the large rectangle
      DATA X1/50,300,300, 50,50/
      DATA Y1/50, 50,300,300,50/
C Data for the small rectangle
      DATA X2/100,200,200,100,100/
      DATA Y2/100,100,200,200,100/
C Create the boxes as polygon definitions
C There is no need to actually be in graphics mode
C or have a graphics device open
      CALL CREATE_POLYGON@(X1,Y1,5,HANDLE(1),ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 20
      CALL CREATE_POLYGON@(X2,Y2,5,HANDLE(2),ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 20
```

```
C Produce a new polygon by combining the large and small boxes
      CALL COMBINE_POLYGONS@(HANDLE,2,HANDLE(3),ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 20
C Enter graphics mode
      CALL EGA@
C Fill the new polygon with colour 4
      CALL FILL_POLYGON@(HANDLE(3),4,ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 10
C Wait for a key press
      CALL GET_KEY@(K)
C Return to text mode
      CALL TEXT_MODE@
      STOP '***** OK'
C Error handling
 10     CALL TEXT_MODE@
 20     IF(ERROR_CODE.EQ.1)THEN
          STOP '***** ERROR: Out of memory'
      ELSEIF(ERROR_CODE.EQ.2)THEN
          STOP '***** ERROR: Invalid polygon handle'
      ENDIF
      END
```

CREATE_POLYGON@

Purpose To get a handle for a specified polygon.

Syntax

```
SUBROUTINE CREATE_POLYGON@(X,Y,N,HANDLE,ERROR_CODE)
INTEGER*2 X(N),Y(N),N,HANDLE,ERROR_CODE
```

Description This routine creates a polygon with ordered set of edges (IX(1),IY(1))..(IX(N),(IY(N)) in memory and returns the value HANDLE for use in other polygon functions (see page 47).

If the polygon is not closed then the function will close it automatically. However, it is good practice to provide a closed polygon. It is also recommended that a polygon that is filled on a plotter should also be edged.

The creation of a polygon is independent of the graphics screen mode or device for which it is intended.

The polygon error codes in this and other polygon functions are:

ERROR_CODE	Description
0	operation successful
1	out of memory; the polygon is too complex to be created or filled with the available memory
2	invalid polygon handle; the polygon specified is not present

It is advisable to check the error code on every call of a polygon function.

See also COMBINE_POLYGONS@, MOVE_POLYGON@, FILL_POLYGON@, DELETE_POLYGON_DEFINITION@.

Example See FILL_POLYGON@

DELETE_POLYGON_DEFINITION@

Purpose To delete a polygon definition.

Syntax
```
SUBROUTINE DELETE_POLYGON_DEFINITION@(HANDLE,ERROR_CODE)
INTEGER*2 HANDLE,ERROR_CODE
```

Description This routine frees the memory associated with a polygon formed by using CREATE_POLYGON@ and disassociates the handle. The error codes are the same as those for CREATE_POLYGON@.

DRAW_HERSHEY@

Purpose To draw an Hershey character.

Syntax
```
SUBROUTINE DRAW_HERSHEY@(IHERSH,IH,IV,ICOL,IH_END,IV_END)
INTEGER*2 IHERSH,IH,IV,ICOL,IH_END,IV_END
```

Description This function draws the Hershey character number IHERSH at the position (IH, IV) in the colour ICOL. The position which could be used for a following character is returned as (IH_END, IV_END). In the Hershey set of occidental character digitisations, every character graphic was assigned a number in the range 1..3926, though not every value in this range was used. Here the convention of using 0 as the space character is adopted. The character is drawn using the current attributes of size, rotation and italicisation (the default values can be changed by calling SET_TEXT_ATTRIBUTE@; the font number is not relevant in this context).

Example

```
C This program uses DRAW_HERSHEY@ and HERSHEY_PRESENT@ to
C draw Hershey characters. The character base line and
C boundaries are indicated by corner sections drawn in red.
C The program requires ANSI.SYS to be loaded.
      PROGRAM HERSH1
      LOGICAL*2 YESNO
      INTEGER*2 I,ID,IH,IV
      CHARACTER POSN*26,ESC
C Use ANSI control strings to position text cursor
      ESC=CHAR(27)
      POSN=ESC//'[10;2fHershey character '
C Select character size 5, the FONT parameter will
C not be used for this program
      CALL SET_TEXT_ATTRIBUTE@(1,5.0,0.0,0.0)
C Change to VGA graphics mode
      CALL VGA@
C Length of line segment used in drawing corners
      ID=10
C Begin search of database at Hershey character 1, concluding
C at character 3926
      DO 1 I=1,3926
C See if the character is present
        CALL HERSHEY_PRESENT@(I,YESNO)
C If present, draw the character
        IF(YESNO)THEN
C Print text string using ANSI cursor controls
          PRINT '(A,I5)',POSN,I
C Draw red left corner angle
          CALL DRAW_LINE@(50-ID,300,50,300,4)
          CALL DRAW_LINE@(50,300,50,300+ID,4)
C Draw the character itself,
C the right hand boundary will be returned in IH and IV
          CALL DRAW_HERSHEY@(I,50,300,7,IH,IV)
C Draw red right corner angle
          CALL DRAW_LINE@(IH,IV,IH+ID,IV,4)
          CALL DRAW_LINE@(IH,IV,IH,IV+ID,4)
C Wait on key press
          CALL GET_KEY@(K)
C If escape was pressed exit from program
          IF(K.EQ.27)GOTO 2
C Otherwise clear the screen ready for the next character
          CALL NEW_PAGE@
        ENDIF
 1      CONTINUE
```

```
C Return to text mode
 2      CALL TEXT_MODE@
       END
```

DRAW_LINE@

Purpose To draw a straight line in graphics mode.

Syntax

```
SUBROUTINE DRAW_LINE@(IX1,IY1,IX2,IY2,ICOL)
INTEGER*2 IX1,IY1,IX2,IY2,ICOL
```

Description DRAW_LINE@ draws a line with colour number ICOL from (IX1, IY1) to (IX2, IY2). The screen must be in graphics mode and the coordinates are pixel numbers so the result depends on the graphics mode in use.

See also POLYLINE@.

Example

```
C Draw a box (50,50), (200,200) of colour 2 in EGA graphics
C mode. Note that it would be simpler to use the routine
C RECTANGLE@ in this case, but this example serves the purpose
C of illustration.
      PROGRAM BOX1
      INTEGER*2 X1,Y1,X2,Y2,K
      DATA X1,Y1,X2,Y2/50,50,200,200/
C ENTER EGA GRAPHICS MODE
      CALL EGA@
C DRAW THE 4 SIDES OF THE BOX
      CALL DRAW_LINE@(X1,Y1,X2,Y1,2)
      CALL DRAW_LINE@(X2,Y1,X2,Y2,2)
      CALL DRAW_LINE@(X2,Y2,X1,Y2,2)
      CALL DRAW_LINE@(X1,Y2,X1,Y1,2)
C WAIT FOR A KEY PRESS AND RETURN TO TEXT MODE
      CALL GET_KEY@(K)
      CALL TEXT_MODE@
      END
```

DRAW_TEXT@

Purpose To draw text in graphics mode.

Syntax
```
SUBROUTINE DRAW_TEXT@(STR,IH,IV,ICOL)
CHARACTER*(*) STR
INTEGER*2 IH,IV,ICOL
```

Description DRAW_TEXT@ draws text on an EGA or VGA screen at the point (IH, IV). STR contains the character string to be drawn. The text is positioned to the nearest pixel (unlike the corresponding BIOS routine). The screen must be in graphics mode. ICOL provides the colour number for the text which appears on the existing background. The text attributes of font, size, rotation and italicisation can be assigned using SET_TEXT_ATTRIBUTE@.

See also DRAW_HERSHEY@.

EGA@ ❹

Purpose To switch to EGA graphics mode.

Syntax
```
SUBROUTINE EGA@
```

Description EGA@ switches a console with an EGA or VGA card to graphics mode with EGA resolution (640x350,16 colours) and clears the screen.

See also VGA@, GRAPHICS_MODE_SET@, TEXT_MODE@.

ELLIPSE@

Purpose To draw an ellipse.

Syntax
```
SUBROUTINE ELLIPSE@(IXC,IYC,IA,IB,ICOL)
INTEGER*2 IXC,IYC,IA,IB,ICOL
```

Description ELLIPSE@ draws an ellipse with centre at (IXC, IYC), with horizontal semi-axis IA, vertical semi-axis IB and colour number ICOL. This routine can be used to produce a circle on the current graphics device if the axes are scaled appropriately by the values given on page 49.

See also FILL_ELLIPSE@.

FILL_ELLIPSE@

Purpose To fill an ellipse.

Syntax
```
SUBROUTINE FILL_ELLIPSE@(IXC,IYC,IA,IB,ICOL)
INTEGER*2 IXC,IYC,IA,IB,ICOL
```

Description FILL_ELLIPSE@ fills an ellipse with centre at (IXC, IYC), with horizontal semi-axis IA, vertical semi-axis IB and colour number ICOL. This routine can be used to fill a circle on the current graphics device if the axes are scaled appropriately by the values given on page 49.

See also ELLIPSE@.

FILL_POLYGON@

Purpose To fill a polygon.

Syntax
```
SUBROUTINE FILL_POLYGON@(HANDLE,ICOL,ERROR_CODE)
INTEGER*2 HANDLE,ICOL,ERROR_CODE
```

Description This routine fills the polygon which has handle HANDLE with colour number ICOL. If the target device is a plotter then it is recommended that the polygon should be edged using POLYLINE@ (assuming that the polygon has not been moved or combined). The error codes are the same as those for CREATE_POLYGON@.

See also CREATE_POLYGON@, COMBINE_POLYGONS@, MOVE_PLOYGON@.

Example

```
PROGRAM BOX3
      INTEGER*2 X(5),Y(5),K,HANDLE,ERROR_CODE
      DATA X/50,200,200, 50,50/
      DATA Y/50, 50,200,200,50/
C Create the box as a polygon definition
C There is no need to actually be in graphics mode
C or have a graphics device open
      CALL CREATE_POLYGON@(X,Y,5,HANDLE,ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 20
C Enter ega graphics mode
      CALL EGA@
C Fill the box with colour 4, it is only now that the box
```

```
C will appear at all on the screen
      CALL FILL_POLYGON@(HANDLE,4,ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 10
C Edge the box, but use a different colour
C Note the return to (50,50) to close the box
      CALL POLYLINE@(X,Y,5,7)
C Wait for a key press return to text mode
      CALL GET_KEY@(K)
      CALL TEXT_MODE@
      STOP '**** OK'
C Error handling
 10     CALL TEXT_MODE@
 20     IF(ERROR_CODE.EQ.1)THEN
        STOP '***** ERROR: Out of memory'
      ELSEIF(ERROR_CODE.EQ.2)THEN
        STOP '***** ERROR: Invalid polygon handle'
      ENDIF
      END
```

FILL_RECTANGLE@

Purpose To fill a rectangle.

Syntax

```
SUBROUTINE FILL_RECTANGLE@(IX1,IY1,IX2,IY2,ICOL)
INTEGER*2 IX1,IY1,IX2,IY2,ICOL
```

Description FILL_RECTANGLE@ fills a rectangle in colour number ICOL where (IX1, IY1) and (IX2,IY2) are opposite corners. This routine is similar to CLEAR_SCREEN_AREA@.

See also RECTANGLE@, CLEAR_SCREEN_AREA@.

GET_ALL_PALETTE_REGS@ ❹

Purpose To get all palette registers for colour graphics.

Syntax

```
SUBROUTINE GET_ALL_PALETTE_REGS@(CREGS)
CHARACTER*17 CREGS
```

Description This routine gets the contents of all of the 17 palette registers. CREGS is an array of 17 bytes containing the palette register values (see page 45).

See also GET_VIDEO_DAC_BLOCK@.

GET_DEVICE_PIXEL@ ❹

Purpose To get the pixel colour for a virtual screen or printer.

Syntax
```
SUBROUTINE GET_DEVICE_PIXEL@(IX,IY,ICOL)
INTEGER*2 IX,IY,ICOL
```

Description Returns the colour of the pixel at (IX, IY) for the current graphics device in ICOL. This routine differs from GET_PIXEL@ in that GET_PIXEL@ always relates to the screen, whereas GET_DEVICE_PIXEL@ relates to a currently open virtual screen or printer (not a plotter).

See also SET_DEVICE_PIXEL@, GET_PIXEL@

GET_GRAPHICS_MODES@ ❸

Purpose To get details of all the graphics modes.

Syntax
```
 SUBROUTINE GET_GRAPHICS_MODES@(XRES,YRES,COLOURS,MODE,
+  BANKED)
 INTEGER*2 XRES(*),YRES(*),MODE(*)
 INTEGER*4 COLOURS(*)
 LOGICAL*2 BANKED(*)
```

Description This routine returns arrays containing the horizontal and vertical resolutions, the number of colours, the corresponding mode and a logical array (0 or 1) indicating if the mode uses “banked” memory. An array size of 20 should be sufficient for all the modes. This function has the side effect of configuring the library to your graphics board.

See also GRAPHICS_MODE_SET@, SCREEN_TYPE@.

GET_GRAPHICS_RESOLUTION@ ❸

Purpose To get details of the high resolution graphics mode.

Syntax
```
SUBROUTINE GET_GRAPHICS_RESOLUTION@(HSIZE,VSIZE,NCOLOURS)
```

```
INTEGER*2 HSIZE,VSIZE,NCOLOURS
```

Description This routine gets details of a predefined high resolution graphics mode that has been assigned using the CONFIGDB utility (or has a default value, see page 309 of the *FTN77 User's Guide*).

The function yields the horizontal and vertical resolutions HSIZE and VSIZE and the number of colours NCOLOURS.

Notes In most situations it is better to use GET_GRAPHICS_MODES@ instead.

GET_PIXEL@ ❹

Purpose To get a pixel colour.

Syntax
```
SUBROUTINE GET_PIXEL@(IH,IV,ICOL)
INTEGER*2 IH,IV,ICOL
```

Description GET_PIXEL@ gets the colour number ICOL of the pixel at (IH, IV) on the screen. Higher order bits of ICOL may contain unwanted bit planes. For example, with 16 colours, **and** with Z'F' to mask off higher order bits.

See also SET_PIXEL@, GET_DEVICE_PIXEL@, SET_DEVICE_PIXEL@.

GET_TEXT_MODES@ ❷

Purpose To get information about the available text modes.

Syntax
```
 SUBROUTINE GET_TEXT_MODES@(COLUMNS,ROWS,MODE,CELL_HEIGHT,
+  CELL_WIDTH)
 INTEGER*2 COLUMNS(*),ROWS(*),MODE(*),CELL_HEIGHT(*),
+  CELL_WIDTH(*)
```

Description This routine returns the text modes supported by your video card in a set of arrays. The arrays are filled with the number of columns and rows, the mode number and the size of the character cell for each supported mode. Please ensure that your arrays are large enough to hold all the returned data. An array size of 30 will suffice for all graphics cards currently supported. On return the last MODE array element will be followed by the value -1.

This routine has the side-effect of configuring DBOS to your graphics board.

See also TEXT_MODE_SET@

GET_TEXT_SCREEN_SIZE@ ❷

Purpose To get the resolution of the current text mode.

Syntax
```
SUBROUTINE GET_TEXT_SCREEN_SIZE@(HSIZE,VSIZE)
INTEGER*2 HSIZE,VSIZE
```

Description Reads the text screen size from the BIOS data area (this data may be incorrect if your graphics board does not set this area up correctly). Returns the number of character columns in HSIZE and the number of character rows in VSIZE.

See also TEXT_MODE_SET@, GET_TEXT_MODES@

GET_VIDEO_DAC_BLOCK@ ❹

Purpose To get a block of VGA DAC registers.

Syntax
```
SUBROUTINE GET_VIDEO_DAC_BLOCK@(IFIRST,NREGS,IRGB)
INTEGER*2 IFIRST,NREGS
INTEGER*1 IRGB(3,NREGS)
```

Description This routine gets a block of video DAC registers. It passes information in the same manner as SET_VIDEO_DAC_BLOCK@.

See also GET_DACS_FROM_SCREEN_BLOCK@.

GRAPHICS_MODE_SET@ ❹

Purpose To set the graphics mode to a given resolution.

Syntax
```
 SUBROUTINE GRAPHICS_MODE_SET@(IXRES,IYRES,NCOLOURS,
+  ERROR_CODE)
 INTEGER*2 IXRES,IYRES,ERROR_CODE
 INTEGER*4 NCOLOURS
```

Description This routine finds a graphics mode with IXRES horizontal resolution, IYRES vertical resolution and the number of colours NCOLOURS. If no suitable mode exists on your graphics board or the specified mode could not be entered then ERROR_CODE is returned as a non-zero value. It is important to always check the value of ERROR_CODE after calling this routine. Reasons for failure to enter a mode might be:

- insufficient memory on the board for the mode; some modes require 1Mb of graphics memory
- you have an early version of the board that does not support the mode,
- you have an incorrect monitor type; some boards detect the monitor type and will not enter a mode that needs a monitor different from the one which is attached.
- the compiler does not support the mode; only modes returned by GET_GRAPHICS_MODES@ are supported.

WARNING:

Entering a mode for which the monitor is unsuitable is likely to damage the monitor and graphics board. It is THE USER'S responsibility to check the board and monitor for suitability.

See also EGA@, TEXT_MODE@, VGA@, HIGH_RESOLUTION_GRAPHICS_MODE@, USE_VESA_INTERFACE@.

GRAPHICS_WRITE_MODE@

Purpose To select replace/XOR mode before writing to the screen, virtual screen or printer.

Syntax

```
SUBROUTINE GRAPHICS_WRITE_MODE@(MODE)
INTEGER*2 MODE
```

Description This routine sets the graphics write mode for the screen depending on the value of MODE. It also sets the mode for a virtual screen or printer if one of these is open. Values of 0, 1 or 2 will force all subsequent graphics output to replace existing pixels. A value of 3 will cause the output to be XORed with previous pixels.

HERSHEY_PRESENT@

Purpose To test if a character number has a Hershey representation.

Syntax
```
SUBROUTINE HERSHEY_PRESENT@(IHERSH,YES)
INTEGER*2 IHERSH
LOGICAL*2 YES
```

Description This routine tests if the character number IHERSH is present as a digitised character in the database. YES is returned as .TRUE. if the Hershey character IHERSH is present in the database, otherwise YES=.FALSE.

Example See DRAW_HERSHEY@

HIGH_RESOLUTION_GRAPHICS_MODE@ ❹

Purpose To switch to high resolution graphics mode.

Syntax
```
SUBROUTINE HIGH_RESOLUTION_GRAPHICS_MODE@
```

Description This routine switches to a predefined high resolution graphics mode that has been assigned using the CONFIGDB utility (or has a default value). The CONFIGDB utility edits a file called DBOS.CFG which automatically configures DBOS when it is loaded. The routine also clears the screen.

Notes In most situations it is better to use GRAPHICS_MODE_SET@ instead.

IS_TEXT_MODE@ ❷

Purpose To test if the screen is in text or graphics mode.

Syntax
```
SUBROUTINE IS_TEXT_MODE@(STATE)
LOGICAL*2 STATE
```

Description STATE is returned as .TRUE. if the screen is in text mode and is returned as .FALSE. if the screen is in graphics mode. In the case of a VGA board, this routine interrogates the CRTC (cathode ray tube controller). Otherwise the information is obtained from the video BIOS data area.

LOAD_STANDARD_COLOURS@ ❹

Purpose To load the standard colours for 256 colour mode.

Syntax
```
SUBROUTINE LOAD_STANDARD_COLOURS@
```

Description This routine loads the video DACs with the standard colour values of the 320x200x256 VGA mode. It should only be used in 256 colour modes since the video DAC values for the 16 colour modes are different (see page 47).

Video DAC registers 0..15 give the standard EGA colours 0..15. Registers 16..31 are a grey scale of increasing intensity. There follow three 72 colour structures of decreasing intensity. Each 72 structure itself is three 24 colour structures of decreasing saturation. The 24 colour structures can be regarded as a colour wheel going from blue to red to green and back to blue again with all the intermediate hues.

The effect is like this:

	high saturation:	medium saturation:	low saturation:
high intensity:	32-55	56-79	80-103
medium intensity:	104-127	128-151	152-175
low intensity:	176-199	200-223	224-247

Registers 248..255 give black.

MOVE_POLYGON@

Purpose To move the position of a polygon.

Syntax
```
SUBROUTINE MOVE_POLYGON@(HANDLE,IDX,IDY,ERROR_CODE)
INTEGER*2 HANDLE,IDX,IDY,ERROR_CODE
```

Description This routine redefines the polygon with handle HANDLE by shifting the former polygon an amount IDX horizontally and IDY vertically. The error codes are the same as those for CREATE_POLYGON@.

See also CREATE_POLYGON@, FILL_POLYGON@, COMBINE_POLYGONS@.

POLYLINE@

Purpose To draw a number of connected straight lines.

Syntax
```
SUBROUTINE POLYLINE@(IX,IY,N,ICOL)
INTEGER*2 IX(N),IY(N),N,ICOL
```

Description POLYLINE@ draws a straight line from (IX(1), IY(1)) to (IX(2), IY(2)), and continues until (IX(N), IY(N)). That is, it joins N points with straight lines. ICOL specifies the colour number.

For a plotter, POLYLINE@ can be used to draw a continuous line without lifting the pen from the paper.

For a closed polygon, simply set the last pair of coordinates equal to the first pair. For a plotter it is recommended that a filled polygon should be edged using POLYLINE@.

See also DRAW_LINE@

Example

```
      PROGRAM BOX2
      INTEGER*2 X(5),Y(5),K
      DATA X/50,200,200, 50,50/
      DATA Y/50, 50,200,200,50/
C Enter EGA graphics mode
      CALL EGA@
C Draw the 4 sides of the box,
C note the return to (50,50) to close the box
      CALL POLYLINE@(X,Y,5,2)
C Wait for a key press
      CALL GET_KEY@(K)
C Return to text mode
      CALL TEXT_MODE@
      END
```

RECTANGLE@

Purpose To draw a rectangle.

Syntax
```
SUBROUTINE RECTANGLE@(IX1,IY1,IX2,IY2,ICOL)
INTEGER*2 IX1,IY1,IX2,IY2,ICOL
```

Description RECTANGLE@ draws a rectangle in colour ICOL where (IX1, IY1) and

(IX2, IY2) are opposite corners.

RESTORE_GRAPHICS_BANK@ ❸

Purpose To restore the graphics bank after a BIOS call.

Syntax
```
SUBROUTINE RESTORE_GRAPHICS_BANK@
```

Description If any action has been taken which directly or indirectly uses BIOS to calculate a screen address whilst in graphics mode then the current graphics 64k bank number maintained by DBOS may be invalidated. This may result in graphics drawing occurring at the wrong place on the screen.

To remedy this effect, call RESTORE_GRAPHICS_BANK@ after any such action. This routine is not needed if the current graphics mode is any of the standard VGA graphics modes ie 640x350x16 colours, 640x480x16 colours or 320x200x256 colours.

RESTORE_TEXT_SCREEN@ ❷

Purpose To restore a text screen saved with SAVE_TEXT_SCREEN@.

Syntax
```
SUBROUTINE RESTORE_TEXT_SCREEN@(IP,IERR)
INTEGER*4 IP
INTEGER*2 IERR
```

Description Restores the entire text screen saved in IP. IP must be a pointer to a valid text screen block previously obtained from a call to SAVE_TEXT_SCREEN@.

A non-zero returned value for IERR denotes one of the following error conditions:

IERR=1, IP is not a valid text screen block (IP = 0 or IP = -1),

IERR=2, text screen block is corrupt - invalid header.

SAVE_TEXT_SCREEN@ ❷

Purpose To save the whole of the text screen.

Syntax
```
SUBROUTINE SAVE_TEXT_SCREEN@(IP)
INTEGER*4 IP
```

Description This routine allocates its own memory using GET_STORAGE@ and returns its address in IP. You can free this memory by calling RETURN_STORAGE@. A returned value of IP = -1 indicates that there is insufficient heap space.

See also RESTORE_TEXT_SCREEN@.

SCREEN_TYPE@ ❷

Purpose To get the graphics screen type.

Syntax
```
SUBROUTINE SCREEN_TYPE@(TYPE)
INTEGER*2 TYPE
```

Description SCREEN_TYPE@ gets the type of the screen as follows:

TYPE	description
0	No graphics available
1	CGA screen
2	EGA screen
3	VGA screen

See also GET_GRAPHICS_MODES@.

SET_ALL_PALETTE_REGS@ ❹

Purpose To set all palette registers for colour graphics.

Syntax
```
SUBROUTINE SET_ALL_PALETTE_REGS@(CREGS)
CHARACTER*17 CREGS
```

Description This routine is like SET_PALETTE@ but sets the contents of all of the 17 palette registers. CREGS is an array of 17 bytes containing the palette register

values, all of which must be supplied. CREGS(17) is for the overscan (border) colour (see page 45).

See also SET_PALETTE@, SET_VIDEO_DAC_BLOCK@.

SET_DEVICE_PIXEL@

Purpose To set a pixel colour for a virtual screen or printer.

Syntax
```
SUBROUTINE SET_DEVICE_PIXEL@(IX,IY,ICOL)
INTEGER*2 IX,IY,ICOL
```

Description Sets a single pixel at (IX,IY) for the current graphics device to the colour ICOL. This routine differs from SET_PIXEL@ in that SET_PIXEL@ always relates to the screen, whereas SET_DEVICE_PIXEL@ relates to a currently open virtual screen or printer (not a plotter).

See also GET_DEVICE_PIXEL@, SET_PIXEL@

SET_PALETTE@ ❹

Purpose To set a palette register for colour graphics.

Syntax
```
SUBROUTINE SET_PALETTE@(IREG,IVAL)
INTEGER*2 IREG,IVAL
```

Description SET_PALETTE@ is used to change the palette register number IREG (0..15 in this function) to the value IVAL (0..63) for 16 colour EGA and VGA modes (see page 45).

See also SET_ALL_PALETTE_REGS@, SET_VIDEO_DAC_BLOCK@.

SET_PIXEL@ ❹

Purpose To set a pixel to a colour.

Syntax
```
SUBROUTINE SET_PIXEL@(IH,IV,ICOL)
INTEGER*2 IH,IV,ICOL
```

Description SET_PIXEL@ sets the pixel at (IH,IV) on the screen to colour number ICOL.

See also GET_PIXEL@, GET_DEVICE_PIXEL@, SET_DEVICE_PIXEL@.

SET_TEXT_ATTRIBUTE@

Purpose To set the current graphics text attributes.

Syntax

```
SUBROUTINE SET_TEXT_ATTRIBUTE@(FONT,SIZE,ROTATION,ITALIC)
INTEGER*2 FONT
REAL*4 SIZE,ITALIC,ROTATION
```

Description This routine selects FONT to be the current font for use with DRAW_TEXT@ and sets the text attributes as follows:

- For display adapter and printers, fonts 1..3 are the 8x14 (default), the 8x8 and the 8x16 fonts respectively.

  For the plotter, refer to the manual supplied by the manufacturer. For all devices, fonts 101..114 are the Hershey proportionally spaced stroke fonts listed on page 48.

- SIZE is the replication factor from the original which corresponds to SIZE=1.

- Text strings will be written rotated through ROTATION degrees in an anti-clockwise direction about the bottom left corner of the first character in the string (see page 48).

- Text will be sheared ITALIC degrees clockwise from the vertical.

The use of 0 for any parameter will select a default value for that parameter. In general, the screen and printer use different fonts from the plotter. The only exceptions to this are the Hershey fonts which may be used for all graphics output devices.

Notes The colour attributes are not selected with this function.

See also DRAW_TEXT@, DRAW_HERSHEY@.

SET_VIDEO_DAC@ ❹

Purpose To set a VGA DAC register.

Syntax
```
SUBROUTINE SET_VIDEO_DAC@(IREG,IR,IG,IB)
INTEGER*2 IREG,IR,IG,IB
```

Description This routine sets the values of the VGA DAC register number IREG (it is not relevant to EGA mode). IR, IG, and IB are the red, green and blue intensities in the range 0..63.

In 16 colour VGA mode, the colour is given by the palette register number (0..16) each with a value in the range 0..63, which is a DAC register number. In this case only DAC registers 0..63 can be changed using this routine (see page 45).

In 256 colour VGA mode the DAC register number is used to specify the colour in graphics functions and any of the registers 0..255 can be changed using this function (see page 47).

See also RESTORE_GRAPHICS_BANK@, SET_VIDEO_DAC_BLOCK@.

SET_VIDEO_DAC_BLOCK@ ❹

Purpose To set a block of VGA DAC registers.

Syntax
```
SUBROUTINE SET_VIDEO_DAC_BLOCK@(IFIRST,NREGS,IRGB)
INTEGER*2 IFIRST,NREGS
INTEGER*1 IRGB(3,NREGS)
```

Description This routine sets a block of video DAC registers (see pages 45 and 47). NREGS is the number of registers to be set starting at IFIRST. The registers are numbered 0..255 and each consists of a red/green/blue triple with components in the range 0..63. IRGB is a two dimensional array containing the number NREGS of triples to be set.

IRGB(1,M) is the red gun level for DAC register

IFIRST + M - 1 etc..

See also RESTORE_GRAPHICS_BANK@, SET_VIDEO_DAC@.

TEXT_MODE@ ❸

Purpose To return to text mode.

Syntax
```
SUBROUTINE TEXT_MODE@
```

Description TEXT_MODE@ clears the screen and switches to text mode, setting the text cursor position to (0,0) (i.e.the top left-hand corner).

See also EGA@, VGA@, GRAPHICS_MODE_SET@, HIGH_RESOLUTION_GRAPHICS_MODE@.

TEXT_MODE_SET@ ❹

Purpose To select the current text mode.

Syntax
```
 SUBROUTINE TEXT_MODE_SET@(COLUMNS,ROWS,CELL_HEIGHT,
+  CELL_WIDTH,ICODE)
 INTEGER*2 COLUMNS,ROWS,CELL_HEIGHT,CELL_WIDTH,ICODE
```

Description This routine looks for and selects a suitable text mode, with the given resolution COLUMNS by ROWS and the given cell size of at least CELL_HEIGHT by CELL_WIDTH (in pixels). Supplying a zero value for CELL_HEIGHT and/or CELL_WIDTH will result in a sensible default being used for the corresponding parameter(s) (in most cases it is sufficient to set both of these to zero). If a mode which satisfies these conditions is not available, then ICODE will be returned as a non-zero value.

See also GET_TEXT_MODES@

USE_VESA_INTERFACE@ ❸

Purpose To force the VESA interface to be used.

Syntax
```
SUBROUTINE USE_VESA_INTERFACE@
```

Description This routine forces use of the VESA interface for graphics mode changing and display memory banking. Most graphics boards now contain a video bios extension in the form of a TSR program on the utility disks. The bios extension TSR must be loaded before this routine is used and preferably before DBOS is invoked.

See also GET_GRAPHICS_MODES@, GRAPHICS_MODE_SET@.

VGA@ ❹

Purpose To switch to VGA graphics mode.

Syntax

```
SUBROUTINE VGA@
```

Description VGA@ switches a console with a VGA card to graphics mode with VGA resolution (640x480,16 colours) and clears the screen.

See also EGA@, GRAPHICS_MODE_SET@

9.

Graphics printer

Introduction

The routines described in this chapter provide support for printer graphics. For convenience the graphics routines are divided between this chapter and chapters 7 and 8. (Note also that chapter 13, includes the routines PRINT_CHARACTER@, INITIALISE_PRINTER@, and GET_PRINTER_STATUS@. These routines are not usually needed but may be used to drive the printer via low level BIOS calls.)

The graphics printer is one of three so-called "auxiliary" graphics devices. The other two are the screen and the plotter (see the introduction to chapter 7 for general information on auxiliary devices).

The default printer

The default printer type is an Epson 9 pin dot matrix or compatible printer with a map size of 960x576 pixels. When using the default printer, a device is firstly opened (using OPEN_GPRINT_DEVICE@ to output to a printer, or OPEN_GPRINT_FILE@ to output to a file), then drawn to (using drawing routines described in chapter 7) and finally closed (using CLOSE_GRAPHICS_PRINTER@).

In the case of the screen and plotter, output is produced simultaneously with the drawing command. However, in the case of the graphics printer, a high resolution bit map is maintained and is updated when a call to one of the graphics output routines is made. When the printer is closed, the bit map is written to the device or to the file. Printer output may also be produced by making a call to NEW_PAGE@ or PRINT_GRAPHICS_PAGE@. NEW_PAGE@ clears the graphics bit map whilst PRINT_GRAPHICS_PAGE@ does not.

If the default printer is not appropriate then the printer type should be selected by a single call to SELECT_DOT_MATRIX@ (for a dot matrix printer which is different from the default) or to SELECT_PCL_PRINTER@ (for a PCL type printer). The printer type should be selected before the printer device is opened.

PCL printers

SELECT_PCL_PRINTER@ provides access to all of the major Hewlett Packard PCL printer families. It is expected that many other PCL compatible printers will also function correctly, however, it should be noted that 100% compatibility with the relevant Hewlett Packard printer has been assumed. A number of routines are also available for configuring the PCL printer driver. These routines must be used at the appropriate point in the output process and may only be used after a call to SELECT_PCL_PRINTER@. For this reason it is convenient to arrange the configuring routines into three groups as follows:

1) Routines that may only be called before the printer is opened:

SET_PCL_BITPLANES@
SET_PCL_LANDSCAPE@.

2) Routines that may only be called after the printer is opened:

LOAD_PCL_COLOURS@
GET_PCL_PALETTE@
SET_PCL_PALETTE@.

3) Routines that may be called before or after the printer is opened:

SET_PCL_GAMMA_CORRECTION@
SET_PCL_GRAPHICS_DEPLETION@
SET_PCL_GRAPHICS_SHINGLING@
SET_PCL_RENDER@.

The general process for outputting to a PCL printer may then be summarised as follows.

a) select the printer type using SELECT_PCL_PRINTER@.

b) configure the driver (optional first stage) using routines in groups (1) and (3).

c) open the printer using OPEN_GPRINT_DEVICE@ or OPEN_GPRINT_FILE@.

d) send the image to the printer buffer using one or more of the drawing routines in chapter 7.

e) configure the driver (optional second stage) using routines in groups (2) and (3).

f) print the image using NEW_PAGE@ or PRINT_GRAPHICS_PAGE@.

g) close the printer using CLOSE_GRAPHICS_PRINTER@.

Steps (d) and (e) are interchangable.

A printer driver is configured by using of one or more of the subroutines in groups (1), (2) and (3) above. A particular routine may not apply to all printers. For example, the LaserJet series do not print in colour so using a routine to set the number of bit planes is not appropriate.

Note that the whole of the configuration process is carried out after the printer has been selected (step a) and before the image is printed (step f). Certain aspects of the configuration, such as selecting the number of bit planes and the landscape or portrait orientation, must be carried out before the printer is opened (step c). Other aspects like setting the colour palette must be carried out after the printer is opened. Some aspects, like setting the gamma correction, may be carried out either before or after the printer is opened.

Where printers support colour, by default the background colour is set to white using the standard colour mapping. The default colour mappings are RGB based, even though the DeskJet 500C and 550C use CMY and CMYK planes. The conversion is carried out for you.

For 8 colour configurations, the definitions are:

Colour index	Colour drawn
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White

For 16 and 256 colour configurations, the default colour mapping is approximately the same as for VGA 16 and 256 colour display modes.

CLOSE_GRAPHICS_PRINTER@

Purpose To close the graphics printer device or file.

Syntax
```
SUBROUTINE CLOSE_GRAPHICS_PRINTER@
```

Description This routine calls NEW_PAGE@ (if the printer buffer has been changed), closes the printer and, if the screen is in graphics mode, graphics output is reverted to the screen.

Example See OPEN_GPRINT_DEVICE@.

GET_PCL_PALETTE@ ❹

Purpose To get the colour definitions for a given number of colours.

Syntax
```
SUBROUTINE GET_PCL_PALETTE@(IPAL,IFIRST,NREGS,IERR)
INTEGER*1 IPAL(3,*)
INTEGER*2 IFIRST,NREGS,IERR
```

Description GET_PCL_PALETTE@ returns the colour definitions for a given number of colours.

NREGS is the number of registers to be returned starting at IFIRST. This routine is applicable only to the PaintJet XL and XL300 printers. Each colour is specified as a set of RGB values. Each component of the RGB value taking values from 0 (zero intensity) to 255 (full intensity).

Input arguments:

IFIRST	first colour in the range
NREGS	the number of colours in the range

Output arguments:

IPAL	an array containing the colour definitions for each of the colours in the specified range.
IERR	= 0, success = 1, printer not open = 2, printer not capable of palette loading = 3, IFIRST out of range

LOAD_PCL_COLOURS@ ❹

Purpose To load the standard colour definitions.

Syntax
```
SUBROUTINE LOAD_PCL_COLOURS@
```

Description This routine loads the standard 16 and 256 colour definitions into the current printer bit map. This is carried out by default when the printer bit map is created.

OPEN_GPRINT_DEVICE@ ❹

Purpose To open a graphics printer.

Syntax
```
SUBROUTINE OPEN_GPRINT_DEVICE@(IDEV,IERR)
INTEGER*2 IDEV,IERR
```

Description Opens the graphics printer for use on IDEV where:

IDEV =1	is LPT1
IDEV =2	is LPT2
IDEV =3	is LPT3
IDEV =4	is LPT4

Opening the printer will close any other auxiliary device (i.e. the plotter or the virtual screen). When the printer is open, graphics output is directed to the printer and not to the screen. IERR = 2 denotes an invalid IDEV value otherwise a non-zero value for IERR denotes the system extended error code.

Example

```
      PROGRAM BOX4
      INTEGER*2 X(5),Y(5),K,HANDLE,ERROR_CODE
      DATA X/50,200,200, 50,50/
      DATA Y/50, 50,200,200,50/
C Create the box as a polygon definition
C There is no need to actually be in graphics mode
C or have a graphics device open
      CALL CREATE_POLYGON@(X,Y,5,HANDLE,ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 20
C Open the printer on LPT1
      CALL OPEN_GPRINT_DEVICE@(1,ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 30
```

```
C Fill the box with colour 4, the box will be filled into
C the internal bit map, no output will appear yet
      CALL FILL_POLYGON@(HANDLE,4,ERROR_CODE)
      IF(ERROR_CODE.NE.0)GOTO 10
C There is no point in edging the box, as only 2 colours
C are available and no effect will be seen
C Wait for a key press
      CALL GET_KEY@(K)
C Closing the printer will produce the output
      CALL CLOSE_GRAPHICS_PRINTER@
      STOP '***** OK'
C Error handling
 10   CALL CLOSE_GRAPHICS_PRINTER@
 20   IF(ERROR_CODE.EQ.1)THEN
        STOP '***** ERROR: Out of memory'
      ELSEIF(ERROR_CODE.EQ.2)THEN
        STOP '***** ERROR: Invalid polygon handle'
      ENDIF
 30   CALL DOSERR@(ERROR_CODE)
      END
```

OPEN_GPRINT_FILE@ ❹

Purpose To direct graphics printer output to a file.

Syntax

```
SUBROUTINE OPEN_GPRINT_FILE@(FILE,IERR)
INTEGER*2 IERR
CHARACTER*(*) FILE
```

Description OPEN_GPRINT_FILE@ is an alternative to OPEN_GPRINT_DEVICE@ and directs graphics printer output to the given file. A non-zero value for IERR denotes the system extended error code.

Example See SELECT_PCL_PRINTER@.

PRINT_GRAPHICS_PAGE@

Purpose To print a graphics page.

Syntax

```
SUBROUTINE PRINT_GRAPHICS_PAGE@
```

Description The graphics page is printed to the graphics printer destination (device or file).

Unlike the NEW_PAGE@ routine, the page is not cleared and can still be drawn to.

Notes Graphics operations are drawn to an internal high resolution bit map and do not produce their own output. The result of the drawing can only be seen by calling this routine, NEW_PAGE@ or CLOSE_GRAPHICS_PRINTER@.

Example See SELECT_PCL_PRINTER@.

SELECT_DOT_MATRIX@ ❷

Purpose To select an Epson compatible dot matrix printer

Syntax

```
SUBROUTINE SELECT_DOT_MATRIX@(ITYPE,NHORZ,NVERT)
INTEGER*2 ITYPE,NHORZ,NVERT
```

Description This routine should be called before opening a printer device (using OPEN_GPRINT_DEVICE@ or OPEN_GPRINT_FILE@) in order to select an Epson compatible dot matrix printer (other than the default) and to provide a printer map with a given resolution. If neither this routine nor SELECT_PCL_PRINTER@ is called then the default printer type (i.e. Epson compatible 9 pin dot matrix printer with a printer map size of 960x576) is used. The routine specifies the resolution of a printer map, where:

ITYPE is the printer type (set this to zero).

NHORZ is the number of pixels horizontally in the printer map.

NVERT is the number of pixels vertically in the printer map.

Note that all arguments are input arguments.

SELECT_PCL_PRINTER@ ❹

Purpose To specify attributes of a PCL printer.

Syntax

```
 SUBROUTINE SELECT_PCL_PRINTER@(ITYPE,PAPER_SIZE, IDPI,
+  NHORZ,NVERT)
 INTEGER*2 ITYPE,IDPI,NHORZ,NVERT
 CHARACTER*(*) PAPER_SIZE
```

Description This routine should be called before opening a printer device (using

OPEN_GPRINT_DEVICE@ or OPEN_GPRINT_FILE@) in order to select a PCL type printer and to specify its attributes. If neither this routine nor SELECT_DOT_MATRIX@ is called then the default printer type (i.e. Epson compatible 9 pin dot matrix printer with a printer map size of 960x576) is used.

ITYPE represents a family of printers, and is referred to below as the driver number. Current values are:

LaserJet series 200	
LaserJet 2	202
LaserJet 3	203
LaserJet 4	204
PaintJet series 300	
PaintJet	302
PaintJet XL	303
PaintJet XL300	304
DeskJet series 400	
DeskJet 500, 500+	402
DeskJet 500C	403
DeskJet 550C	404

In addition to these, the values of 0 and 2 are also supported for compatibilty with the old version of the LaserJet 2 driver.

Although, within the families, the lowest level of printer is usually compatible with the higher specification models, there is not necessarily any advantage in (say) specifying a LaserJet 2 when a LaserJet 3 is attached.

As far as possible, the user should try to match the driver number with the printer that is attached. In this way, full advantage will be taken of any driver specific features such as better image compression or better colour choice or representation. Better image compression means better transmission times which in turn means faster drawing.

The improvement in speed can be substantial. For example, if the file is saved to disc, the file size can be as small as 7% of the size of the uncompressed image.

The exception to this is the PaintJet XL300. This has a different dot pitch from previous printers in this range.

Although images generated for the PaintJet and PaintJet XL will still draw on this printer, the image will be the wrong size and may even be clipped in order to fit on the page. Images generated for one family of printers may not produce anything meaningful on the other families. One of the reasons for this is differing

palette representations.

For example, images produced for the DeskJet 500C or 550C may simply produce a solid black page on one of the PaintJet range and vice versa.

The tables below give a brief outline of the advantages of using a particular driver against using drivers for the lower specification models in the range.

LaserJet	LaserJet 3	LaserJet 4
	better image compression	600 dpi resolution

Paintjet	Paintjet XL	
	more colours	
	higher resolution colour	
	better image compression	

DeskJet 500	DeskJet 500C	DeskJet 550C
	colour	true black rather than composite black
		better image compression

PAPER_SIZE specifies the name of the paper size e.g. 'A4' or 'LETTER'. Values differ for each of the printer families. An incorrectly specifed PAPER_SIZE will default to 'A4'. The table below gives the allowable values for each paper size.

	LaserJet			PaintJet			DeskJet		
	2	3	4	-	XL	XL300	500	500C	550C
EXECUTIVE	✓	✓	✓	×	×	×	✓	✓	✓
LETTER	✓	✓	✓	✓	✓	✓	✓	✓	✓
LEGAL	✓	✓	✓	✓	✓	✓	✓	✓	✓
LEDGER	×	×	×	✓	✓	✓	×	×	×
A4	✓	✓	✓	✓	✓	✓	✓	✓	✓
B4	×	×	×	✓	✓	✓	×	×	×
A3	×	×	×	×	×	✓	×	×	×

IDPI specifies the number of pixels per inch that you want to use.

LaserJet 2,3, PaintJet XL300 and all DeskJet models:

IDPI is one of 300, 150, 100, 75

LaserJet 4:

IDPI is one of 600, 300, 200, 150, 100, 75

PaintJet and PaintJet XL:

IDPI is one of 180, 90

If IDPI is incorrectly specified, the resolution used is the next device resolution higher than that specified, if one is available, or the highest if one higher is not available. For example, suppose 150 dpi were specified with a PaintJet XL. The printer map would use 180 dpi. If 75 dpi were specified, then 90 dpi would be used. However, if 300 dpi were specified then 180 dpi would be used as this is the highest resolution available.

NHORZ and NVERT are returned values and provide the total number of horizontal and vertical pixels in the image.

Do not assume that these values will remain fixed as they may vary for compatibility with new devices.

Example

```
      PROGRAM PRINTER
      INTEGER*2 NHORZ,NVERT,IERR
      CALL SELECT_PCL_PRINTER@(404,'A4',150,NHORZ,NVERT)
      CALL SET_PCL_BITPLANES@(3,IERR)
      IF(IERR.NE.0) GOTO 10
      CALL SET_PCL_LANDSCAPE@(1,NHORZ,NVERT,IERR)
      IF(IERR.NE.0) GOTO 10
      CALL OPEN_GPRINT_DEVICE@(1,IERR)
      IF(IERR.NE.0) GOTO 10
C     LOAD_PCL_COLOURS@, GET_PCL_PALETTE@ & SET_PCL_PALETTE@ may
C     be used here. The next two routines may also be called
C     before the printer is opened.
      CALL SET_PCL_GRAPHICS_DEPLETION@(3,IERR)
      IF(IERR.NE.0) GOTO 10
      CALL SET_PCL_GRAPHICS_SHINGLING@(1,IERR)
      IF(IERR.NE.0) GOTO 10
      CALL ELLIPSE@(NHORZ/2,NVERT/2,NHORZ/4,NVERT/4,1)
      CALL PRINT_GRAPHICS_PAGE@()
      CALL FILL_ELLIPSE@(NHORZ/2,NVERT/2,NHORZ/4,NVERT/4,1)
      CALL CLOSE_GRAPHICS_PRINTER@()
      STOP
```

```
C  Process errors
10 .....
   END
```

SET_PCL_BITPLANES@ ❸

Purpose To set the number of colours in the image.

Syntax

```
SUBROUTINE SET_PCL_BITPLANES@(NBIT,IERR)
INTEGER*2 NBIT,IERR
```

Description The defaults are:

LaserJet series = 1
PaintJet series = 4 (16 colours)
DeskJet series = 3

At present only the PaintJet series drivers can alter the number of colours. This routine may only be used after SELECT_PCL_PRINTER@ and before the printer is opened.

Input arguments:

NBIT allowable values are:

LaserJet series	= 1
PaintJet	= 1, 3
Paintjet XL & XL300	= 1, 3, 4, 8
DeskJet 500	= 1
DeskJet 500C & 550C	= 1, 3

Number of bit planes	Number of colours
1	2
3	8
4	16
8	256

Output arguments:

IERR = 0 success
= 1 printer already open
= 2 invalid value for NBIT

Example See SELECT_PCL_PRINTER@.

SET_PCL_GAMMA_CORRECTION@ ❸

Purpose To alter the "gamma correction" for colours.

Syntax

```
SUBROUTINE SET_PCL_GAMMA_CORRECTION@(GAMMA,IERR)
INTEGER*2 IERR
REAL*8 GAMMA
```

Description This routine is applicable only to the PaintJet XL and XL300 series. Gamma correction adjusts colour intensities on the printer to match those of the human eye and is rather similar to altering the contrast of the image.

Unless there is a real need to depart from the default gamma correction, you are advised to avoid this routine. The default is GAMMA = 1.0.

Input argument:

$0.0 \leq$ GAMMA ≤ 2.0

Output argument:

IERR = 0, success
= 2, printer not capable of gamma correction
= 3, GAMMA out of range

SET_PCL_GRAPHICS_DEPLETION@ ❸

Purpose To improve the image quality.

Syntax

```
SUBROUTINE SET_PCL_GRAPHICS_DEPLETION@(IDEP,IERR)
INTEGER*2 IDEP,IERR
```

Description Reduces the amount of ink laid down, so improving the image quality and the ink saturation of the media. This routine is applicable only to the DeskJet series of printers operating in colour mode. Unless there is a real need to depart from the

default depletion, you are advised to avoid this routine.

Input argument:

IDEP = 1, no depletion - monochrome graphics default
= 2, 25% - colour graphics default
= 3, 50%

Output argument:

IERR = 0, success
= 2, printer not capable of gamma correction
= 3, IDEP out of range

Example See SELECT_PCL_PRINTER@.

SET_PCL_GRAPHICS_SHINGLING@ ❸

Purpose To make a number of print passes.

Syntax

```
SUBROUTINE SET_PCL_GRAPHICS_SHINGLING@(ISHING,IERR)
INTEGER*2 ISHING,IERR
```

Description This routine makes a setable number of print passes, each pass filling vacant pixels from previous passes. This routine is applicable only to the DeskJet series of printers. This is used to prevent liquid inks of different types from coming into contact with each other by giving them a moment to dry. This is particularly useful with the DeskJet 550C printer where the black ink is chemically different from the colour inks. Use this routine when printing on glossy paper or transparencies.

Input argument:

ISHING = 0, no shingling
= 1, two pass printing
= 2, four pass printing - default

Output argument:

IERR = 0, success
= 2, printer not capable of shingling
= 3, ISHING out of range

Example See SELECT_PCL_PRINTER@.

SET_PCL_LANDSCAPE@ ❸

Purpose To set LANDSCAPE or PORTRAIT orientation.

Syntax
```
SUBROUTINE SET_PCL_LANDSCAPE@(IVAL,NHORZ,NVERT,IERR)
INTEGER*2 IVAL,NHORZ,NVERT,IERR
```

Description PCL printers do not usually rotate graphics images so the rotation is carried out internally by the driver.

This routine may only be used after SELECT_PCL_PRINTER@ and before the printer is opened.

Input argument:

IVAL = 0 sets PORTRAIT orientation
= 1 sets LANDSCAPE orientation

Output arguments

NHORZ	returns the number of pixels horizontally
NVERT	returns the number of pixels vertically
IERR	returns 1 if the printer is open.

Example See SELECT_PCL_PRINTER@.

SET_PCL_PALETTE@ ❸

Purpose To load the colour definitions.

Syntax
```
SUBROUTINE SET_PCL_PALETTE@(IPAL,IFIRST,NREGS,IERR)
INTEGER*1 IPAL(3,*)
INTEGER*2 IFIRST,NREGS,IERR
```

Description SET_PCL_PALETTE@ loads the colour definitions for a given number of colours. NREGS is the number of registers to be set starting at IFIRST. This routine is applicable only to the PaintJet XL and XL300 printers.

Each colour is specified as a set of RGB values. Each component of the RGB value taking values from 0 (zero intensity) to 255 (full intensity).

Input arguments:

IPAL	an array containing the colour definitions for each of the colours in the specified range
IFIRST	first colour in the range

NREGS the number of colours in the range

Output argument:

IERR = 0, success
= 1, printer not open
= 2, printer not capable of palette loading
= 3, IFIRST out of range

SET_PCL_RENDER@

Purpose To set the "rendering algorithm".

Syntax

```
SUBROUTINE SET_PCL_RENDER@(IREND,IERR)
INTEGER*2 IREND,IERR
```

Description Sets the "rendering algorithm" for the way that colours are rendered by the printer.

This routine is applicable only to the PaintJet series. Unless there is a real need to depart from the default rendering, the user is advised to avoid this routine.

Input argument:

IREND = 0 no algorithm
= 1 snap to primaries
= 2 snap black to white, all other colours black
= 3 ordered dither (default)
= 4 error diffusion
= 5 monochrome ordered dither
= 6 monochrome error diffusion
= 7 clustered ordered dither
= 8 monochrome clustered ordered dither.

Output argument:

IERR = 0, success
= 2, printer not capable of rendering
= 3, IREND out of range

10.

Hot key (DOS)

One of the attractive features of MS-DOS is the way in which programs can be made to stay resident and become active on pressing a "hot key". However, most such programs are written in assembler and are hard to code. Furthermore, because such programs stay resident over the top of DOS they use up valuable memory space. FTN77 offers a mechanism to write "hot key" programs which do not consume memory when not in use, and which in any case utilise the memory above 640K, which is usually plentiful. All the complexities of hot-key software are catered for by a small (5K bytes) TSR program called HOTKEY77. This program should be run after DBOS has been loaded, and may be incorporated in your AUTOEXEC.BAT file.

Once the HOTKEY77 program has been executed it is possible to use one hot key immediately by executing HELP77. This program (which you may also usefully include in AUTOEXEC.BAT) defines the key sequence Control-Alt-H, which is normally inactive, to cause the display of FTN77 help information (starting at the index). Here is the source of HELP77:

```
EXTERNAL FTN77_HELP@
CALL DEFINE_HOT_KEY@('CONTROL-ALT-H',FTN77_HELP@, IC)
CALL DOSERR@(IC)
CALL COU@('Control-Alt-H will now give FTN77 help screens')
END
```

The routine FTN77_HELP@ is present in the system library, and DEFINE_HOT_KEY@ is defined below. HELP77 uses up no extra memory, it simply sets up information inside HOTKEY77. Since the hot key program is not limited in size by memory considerations, there would be no problem defining a hot key to simulate an oil refinery (say) if this were thought useful! Users of HOTKEY77 should note the following:

- An additional benefit of HOTKEY77 is that the standard keyboard buffer of 16 keystrokes is increased to 512. This has been done to facilitate hot key applications which feed data back to the interrupted program (e.g. a spelling checker which

corrects what it finds on an editor screen), but it also means that you can type much further ahead while the PC is performing a task.

- Hot key applications should be written as dynamic link libraries (and included in the LIBRARIES.DIR file like any other such library).
- The 'main program' of such an application should be written as an INTERRUPT SUBROUTINE. For example the system routine FTN77_HELP@ is written as an interrupt subroutine.
- If you define a hot key whose handler is not in a dynamic link library *or if you alter the library after the hot key has been defined* you will almost certainly crash DOS.
- After a hot key has been pressed, it will take effect as soon as the program attempts to read the keyboard. Normally a program will be awaiting keyboard input when you press a hot key, so the effect will be immediate.
- Hot keys may be used from within FTN77 programs (or utilities like LINK77). In this case they act like trap routines (see SET_TRAP@) and are subject to the same restrictions. In particular, if a hot key program is likely to be invoked while an FTN77 program is performing a READ statement, then it must not itself perform Fortran I/O.
- Usually a hot key application would operate within a window which it would create as it starts and remove before it exits. In this way the underlying screen is not disturbed.
- Hot keys must not be invoked from within the FTN77 debugger.
- If you use other hot key programs not involving HOTKEY77 they should be loaded after DBOS and HOTKEY77.
- Hot key programs should be written so that they always finish cleanly by returning from the top level INTERRUPT SUBROUTINE.

  Termination as a result of an error, or as a result of executing STOP etc. will leave DOS in an ill-defined state.

DEFINE_HOT_KEY@ ❷

Purpose To associate a hotkey routine with a given key.

Syntax

```
SUBROUTINE DEFINE_HOT_KEY@(KEY,ROUTINE,ERROR_CODE)
INTEGER*2 ERROR_CODE
CHARACTER*(*) KEY
EXTERNAL ROUTINE
```

Description DEFINE_HOT_KEY@ associates routine ROUTINE with the key named KEY. The names of keys are case insensitive and are best illustrated by example:

CONTROL-ALT-DEL
: the key which normally reboots the machine;

ctrl-alt-del
: this is the same key under a different name;

Shift-Alt-Delete
: this refers to the Delete key as opposed to the key on the numeric keypad;

alt-9
: this is not the numeric keypad;

FOUR
: this is the numeric keypad left arrow.

Hot keys are unaffected by the state of the shift lock, numlock or scroll lock toggles.

Thus it is possible to define hot keys using key combinations which have no normal effect - for example Ctrl-Alt-L. It would even be possible to redefine Ctrl-Alt-Del so that it did not reboot the machine. The hot key will work whether or not an FTN77 program is active at the time it is pressed. If you already have 20 active hot keys, or if you have not loaded HOTKEY77, then ERROR_CODE will come back with a non-zero error code which can be interpreted by DOSERR@.

REMOVE_HOT_KEY@ ❷

Purpose To disassociate a hotkey routine from a given key.

Syntax
```
SUBROUTINE REMOVE_HOT_KEY@(KEY)
CHARACTER*(*) KEY
```

Description This routine removes the key KEY from the hot key table. A key with special meaning such as Ctrl-Alt-Del recovers that meaning. This routine never returns an error - if the HOTKEY77 program has not been loaded, or if the key in question has never been defined, then no action is taken.

FEED_KEYBOARD@ ❷

Purpose To push a keycode into the keyboard buffer.

Syntax
```
SUBROUTINE FEED_KEYBOARD@(DATA,ERROR_CODE)
INTEGER*2 DATA,ERROR_CODE
```

Description This routine takes the scan code/ASCII pair in DATA (scan code in high byte) and pushes it into the keyboard buffer. Although this routine may be used without HOTKEY77, it is then limited by the 16 keystroke buffer. With HOTKEY77 the buffer is increased to 512 keystrokes. This routine is usually used to “return a result” from a hot key program.

Example
```
        EXTERNAL GIVE_DATE
C DEFINE Control-Alt-D TO RETURN THE DATE
        CALL DEFINE_HOT_KEY@('CONTROL-ALT-D',6,GIVE_DATE,IC)
        CALL DOSERR@(IC)
        END
C THIS CODE MUST RESIDE IN A DYNAMIC LINK LIBRARY
        INTERRUPT SUBROUTINE GIVE_DATE
        CHARACTER*28 FDATE@,DATE
        DATE=FDATE@()
        DO 1 I=1,LENG(DATE)
C NOTE THAT THE SCAN-CODE PORTION OF THE KEY IS NOT USED
C BY MOST PROGRAMS FOR ASCII KEYS AND HAS BEEN LEFT ZERO
 1      CALL FEED_KEYBOARD@(ICHAR(DATE(I:I)))
        END
```

11.

In-line

The routines in this chapter (except for SET_IO_PERMISSION@) are converted to in-line code, rather than procedure calls, and are therefore extremely efficient. They can often be used as a convenient alternative to resorting to assembler. Those routines described here which are functions should be explicitly declared to be of the right type.

FILL@

Purpose To set an array of N bytes to a particular value.

Syntax
```
SUBROUTINE FILL@(A,N,B)
INTEGER*4 A,B,N
```

Description This routine fills A (which may be of any type and is usually an array) with N bytes of value B. Thus if A is of type INTEGER*4 and N=4, each of the 4 bytes of A will be assigned to the value of B. B may be of any type but only the lowest byte is used.

IN@ ❷

Purpose To input one byte from an I/O port.

Syntax
```
INTEGER*2 FUNCTION IN@(PORT)
INTEGER*2 PORT
```

Description To read and write I/O ports you must have set the I/O permission level (IOPL) to 3 (see SET_IO_PERMISSION@).

MATCH@

Purpose To compare two arrays of N bytes.

Syntax
```
LOGICAL*2 FUNCTION MATCH@(A,B,N)
INTEGER*2 A,B,N
```

Description This function compares N bytes of data for equality. The first two arguments can be of any type and are usually arrays. N can be INTEGER*1, INTEGER*2 or INTEGER*4.

Return value MATCH@ returns .TRUE. if the two arrays are identical, otherwise .FALSE..

MOVE@

Purpose To copy an array of N bytes.

Syntax
```
SUBROUTINE MOVE@(FROM,TO,N)
```

Description This routine copies N bytes of data from FROM to TO. No data conversion is performed and *no* checks are made to ensure that the source and destination are large enough (even in /CHECK mode). Arguments FROM and TO may be of any type, N must be INTEGER*1, INTEGER*2 or INTEGER*4.

OUT@ ❷

Purpose To output one byte of data to an I/O port.

Syntax
```
SUBROUTINE OUT@(PORT,VALUE)
INTEGER*2 PORT,VALUE
```

Description To read and write I/O ports you must have set the I/O permission level (IOPL) to 3 (see SET_IO_PERMISSION@).

POP@

Purpose To pop a value off the system stack.

Syntax
```
SUBROUTINE POP@(A)
INTEGER*4 A
```

Description This routine is the opposite of PUSH@.

PUSH@

Purpose To push a value on the system stack.

Syntax
```
SUBROUTINE PUSH@(A)
INTEGER*4 A
```

Description The argument A is pushed on to the system stack. Two bytes will be pushed for INTEGER*2, four for INTEGER*4 etc. Values saved in this way can be restored again with POP@. The corresponding POP@ call must have an argument of the same type (otherwise the wrong number of bytes would be popped). A routine may return with data still pushed on the stack - such data is then lost.

SET_IO_PERMISSION@ ❷

Purpose To set the I/O permission level to 3 or 0.

Syntax
```
SUBROUTINE SET_IO_PERMISSION@(OPTION)
LOGICAL*2 OPTION
```

Description This routine sets the I/O permission level (IOPL) to 3 if OPTION is .TRUE.. Typically this routine is called before one or more calls to IN@ or OUT@.

Each call to SET_IO_PERMISSION@ with OPTION=.TRUE. pushes the value 3 on to a stack. The top level of the stack gives the current permission level. The bottom level gives the initial default level (zero). Each call with OPTION =.FALSE. pops a value off the stack. It is good practice to return IOPL to zero when it is no longer needed, as this helps to protect the program from corrupting the system.

12.

Mouse

A number of routines are provided by FTN77 to support Microsoft-compatible mouse drivers. These routines are simply bindings to built-in functions of the mouse driver. The exact way in which some of these work may vary from supplier to supplier: some may not work at all with your mouse. You are advised to check the functionality of your mouse with these routines before using them with confidence.

A mouse can be used in either graphics mode (using EGA@ for example) or text mode. Programs must initialise the mouse before use, and if a mouse interrupt mask is set then this must be cleared before exiting from the program. Some care is needed when using these routines. In particular, the mouse should be turned off during any screen operations which may alter the screen data underneath the mouse cursor, otherwise the area will not be properly repaired when the mouse is moved.

Some of these routines take screen units (pixels) as arguments whilst others take physical mouse movements (mickeys). A mickey is approximately 0.5 mm. It is possible to alter the ratio of mickeys to pixels so that the display cursor may be made more or less sensitive to mouse movements.

There are routines to reposition the mouse cursor and to constrain the cursor movement to a box. These may be used to position the cursor on the first item of a menu, for example, and to prevent mouse movements outside of a selected range so that only certain menu items may be chosen. Similarly the mouse cursor may be moved to a safe area of the screen whilst another area is being updated. The most usual use, however, will be to make the whole screen area available to the mouse and this should be done as soon as the appropriate screen mode is entered and the mouse initialised.

In order to use a mouse together with (say) the keyboard, it is usual to construct an "event handler" of some kind. One possible approach is to use the GET_MOUSE_BUTTON_PRESS_COUNT@ routine (with GET_KEY1@ say). Mouse movements and button presses may be trapped using SET_TRAP@ and SET_MOUSE_INTERRUPT_MASK@. However, since you are effectively providing an interrupt handler for these events there are strict rules governing your

allowed actions. In particular, no system routines may be called, the register set must be saved, and interrupts must not be re-enabled.

It should be noted that mouse drivers usually do not recognise non-standard VGA screen modes. In other words, the mouse driver cannot tell that you are in a particular graphics mode if that mode is not standard. This means that you should not attempt to use a mouse driver routine to display the mouse cursor in 800x600 graphics mode for example. If the mouse driver returns the mouse coordinates in 8 pixel (rather than single pixel) increments then this indicates that it has not recognised the current graphics mode.

DISPLAY_MOUSE_CURSOR@ ❸

Purpose To show the mouse cursor on the screen.

Syntax
```
SUBROUTINE DISPLAY_MOUSE_CURSOR@
```

Description This routine causes the mouse cursor to appear on the screen. A change of mode between text and graphics will hide the cursor. Similarly moving into a region defined by MOUSE_CONDITIONAL_OFF@ will also hide the cursor.

If HIDE_MOUSE_CURSOR@ is called twice then DISPLAY_MOUSE_CURSOR@ must also be called twice before the cursor reappears and so on.

GET_MOUSE_BUTTON_PRESS_COUNT@

Purpose To get the number of times a button has been pressed.

Syntax
```
SUBROUTINE GET_MOUSE_BUTTON_PRESS_COUNT@(IB,IC)
INTEGER*2 IB,IC
```

Description This routine gets the count of the number of times a button has been pressed. IB is set to 0 for the left button and 1 for the right button. After each call to this routine, the count IC for the specified button is reset to zero.

GET_MOUSE_EVENT_MASK@

Purpose To get the mask for the most recent mouse interrupt.

Syntax
```
SUBROUTINE GET_MOUSE_EVENT_MASK@(MASK)
INTEGER*2 MASK
```

Description This routine returns the mask for the most recent mouse interrupt. The bits in the mask are used as in SET_MOUSE_INTERRUPT_MASK@ and are set if that event has occurred. This routine should only be called after a mouse interrupt has occurred and is normally called from within the interrupt routine that handles the interrupt.

GET_MOUSE_PHYSICAL_MOVEMENT@ ❸

Purpose To get the mouse pad distance from the last call.

Syntax
```
SUBROUTINE GET_MOUSE_PHYSICAL_MOVEMENT@(DX,DY)
INTEGER*2 DX,DY
```

Description This routine gets the relative position of the mouse on the mouse pad since the last call. The displacement is measured in mickeys (see SET_MOUSE_SENSITIVITY@ and GET_MOUSE_SENSITIVITY@). The mouse cursor on the screen is confined to a rectangle (the full screen perhaps), so the values given will not represent the screen displacement.

GET_MOUSE_POSITION@

Purpose To get the present state of the mouse cursor.

Syntax
```
SUBROUTINE GET_MOUSE_POSITION@(IH,IV,BUTTON_STATUS)
INTEGER*2 IH,IV,BUTTON_STATUS
```

Description This routine returns the position of the mouse in pixel coordinates (IH, IV) from the top-left of the screen. It also returns the state of the mouse buttons as either depressed or not. The least significant bit of BUTTON_STATUS, (bit 0) = 1 if the left-hand button is depressed. Bit 1 = 1 if the right-hand button is depressed. Bit 2 = 1 if a middle button is depressed. Any combination of values is possible.

Example See SET_MOUSE_INTERRUPT_MASK@

GET_MOUSE_SENSITIVITY@

Purpose To get the values of the physical movement ratios and the double speed threshold.

Syntax
```
SUBROUTINE GET_MOUSE_SENSITIVITY@(DX,DY,SPEED)
INTEGER*2 DX,DY,SPEED
```

Description This routine gets the values described in SET_MOUSE_MOVEMENT_RATIO@ and SET_MOUSE_SPEED_THRESHOLD@.

HIDE_MOUSE_CURSOR@

Purpose To hide the mouse cursor on the screen.

Syntax
```
SUBROUTINE HIDE_MOUSE_CURSOR@
```

Description This routine causes the mouse cursor to disappear from the screen. Unlike the DISPLAY_MOUSE_CURSOR@ routine, it need only be called once no matter how many times the other routine has been called.

INITIALISE_MOUSE@

Purpose To initialise the mouse driver.

Syntax
```
SUBROUTINE INITIALISE_MOUSE@
```

Description This routine initialises the mouse driver and resets the mouse. It should be called before the first use of the mouse in order to obtain a reproducible state and so that both DISPLAY_MOUSE_CURSOR@ and HIDE_MOUSE_CURSOR@ work reliably. It does not cause the mouse cursor to appear on the screen.

MOUSE@ ❸

Purpose To perform a mouse interrupt.

Syntax
```
SUBROUTINE MOUSE@(IA,IB,IC,ID)
INTEGER*2 IA,IB,IC,ID
```

Description This routine performs a mouse interrupt with the registers loaded with IA, IB, IC and ID. The results are also returned in these variables. This routine should rarely be needed. Usually the services provided by the built-in mouse routines will suffice.

This routine should not be used with other routines which cause interrupt activity such as GET_MOUSE_PHYSICAL_MOVEMENT@.

The SET_MOUSE_INTERRUPT_MASK@ and SET_TRAP@ routines should be used to cause mouse interrupt activity.

MOUSE_CONDITIONAL_OFF@ ❸

Purpose To switch off the cursor when it enters a specified rectangle.

Syntax
```
SUBROUTINE MOUSE_CONDITIONAL_OFF@(LX,LY,HX,HY)
INTEGER*2 LX,LY,HX,HY
```

Description This routine switches off the mouse cursor when it enters the region defined with (LX, LY) and (HX, HY) at opposite corners. (0,0) is the top left and values are in pixels.

The routine is a conditional form of HIDE_MOUSE_CURSOR@, so the cursor is restored by a call to DISPLAY_MOUSE_CURSOR@. The cursor is switched off when any part of its 16x16 pixel form enters the rectangle.

MOUSE_LIGHT_PEN_EMULATION@ ❸

Purpose To use the mouse as a light-pen.

Syntax
```
SUBROUTINE MOUSE_LIGHT_PEN_EMULATION@(SET)
LOGICAL*2 SET
```

Description This routine enables/disables the use of the mouse in place of a light pen. SET=1 enables the emulation, SET=0 disables it. When a mouse button is

depressed, the ROM-BIOS video service 4 reports that the light-pen has been triggered and returns the coordinates of the mouse.

MOUSE_SOFT_RESET@

Purpose To initialise the mouse software.

Syntax
```
SUBROUTINE MOUSE_SOFT_RESET@(INSTALL)
LOGICAL*2 INSTALL
```

Description This routine resets the mouse software, but not the mouse. It is identical to INITIALISE_MOUSE@ except that the mouse is not reset. INSTALL is returned as .TRUE. if the mouse driver is installed, and .FALSE. if it is not.

QUERY_MOUSE_SAVE_SIZE@ ❸

Purpose To get the buffer size for the mouse state.

Syntax
```
SUBROUTINE QUERY_MOUSE_SAVE_SIZE@(SIZE)
INTEGER*2 SIZE
```

Description In preparation for a call to SAVE_MOUSE_DRIVER_STATE@, this routine is used to determine the buffer size in bytes that is required to store the mouse state. Drivers vary in the amount of storage they require, but typically about 500 bytes are needed.

RESTORE_MOUSE_DRIVER_STATE@ ❸

Purpose To restore a former state of the mouse driver.

Syntax
```
SUBROUTINE RESTORE_MOUSE_DRIVER_STATE@(BUFFER,NBYTES)
INTEGER*2 BUFFER,NBYTES
```

Description This routine restores the state of the mouse driver corresponding to an earlier call to SAVE_MOUSE_DRIVER_STATE@. The contents of the buffer are determined by the call to SAVE_MOUSE_DRIVER_STATE@. NBYTES is given by a call to QUERY_MOUSE_SAVE_SIZE@. The mouse cursor is not automatically redrawn.

SAVE_MOUSE_DRIVER_STATE@ ❸

Purpose To save the current state of the mouse driver.

Syntax
```
SUBROUTINE SAVE_MOUSE_DRIVER_STATE@(BUFFER,NBYTES)
INTEGER*2 BUFFER,NBYTES
```

Description This routine stores the current mouse driver state in the array pointed to by BUFFER. NBYTES is the size of the buffer in bytes obtained by a call to QUERY_MOUSE_SAVE_SIZE@.

SET_MOUSE_BOUNDS@ ❸

Purpose To restrict mouse movements to a specified rectangle.

Syntax
```
SUBROUTINE SET_MOUSE_BOUNDS@(LX,LY,HX,HY)
INTEGER*2 LX,LY,HX,HY
```

Description This routine defines a rectangle, with (LX, LY) and (HX, HY) at opposite corners, within which the mouse cursor will be confined. If, in the default state, the mouse cursor is too restricted then this routine can be used to extend the bounds.

SET_MOUSE_GRAPHICS_CURSOR@ ❸

Purpose To specify the shape of the mouse cursor for graphics mode.

Syntax
```
 SUBROUTINE SET_MOUSE_GRAPHICS_CURSOR@(HOT_X,HOT_Y,
+  CURSOR_DEF)
 INTEGER*2 HOT_X,HOT_Y,CURSOR_DEF(32)
```

Description This routine allows the user to specify the shape of cursor in graphics mode and how it reacts with the screen data underneath. The position of the hot-spot can also be set.

The hot-spot is the point (relative to the top left (0,0) of a 16x16 array of pixels) to which the cursor points. For example, if the default cursor is an arrow, then the hot-spot would be the tip of this arrow. For a cross, the hot-spot would naturally be the point where the bars intersect.

CURSOR_DEF points to an array of 32 elements. Elements 1..16 define the

"data mask" whilst elements 17..32 define the "cursor mask". Each element of the array represents a bit-mapped row of the cursor. On a bit-by-bit basis, the screen under the cursor is **and**ed with the data mask and then **xor**ed with the cursor mask in order to produce the visible effect of the cursor.

The default cursor can be restored by reinitialising the mouse.

Example

```
C The following will define a simple cross to be xor'ed
C against the screen data
      INTEGER*2 CURSOR_DEF(32)
C The screen mask ensures that all the screen data is
C preserved for Xor'ing with the cursor mask
      CURSOR_DEF(1)= B'1111111111111111'
      .....
      CURSOR_DEF(16)=B'1111111111111111'
C Elements 17 to 32 are the cursor mask and define
C the shape of the cursor
      CURSOR_DEF(17)=B'0000000000000000'
      CURSOR_DEF(18)=B'0000000010000000'
      .....
      CURSOR_DEF(24)=B'0000000010000000'
      CURSOR_DEF(25)=B'0111111111111110'
      CURSOR_DEF(26)=B'0000000010000000'
      .....
      CURSOR_DEF(31)=B'0000000010000000'
      CURSOR_DEF(32)=B'0000000000000000'
      C Enter VGA mode
      CALL VGA@
C Set the cursor shape with hot-spot at (8,8)
      CALL SET_MOUSE_GRAPHICS_CURSOR@(8,8,CURSOR_DEF)
C Display cursor
      CALL DISPLAY_MOUSE_CURSOR@
      .....
```

SET_MOUSE_INTERRUPT_MASK@

Purpose To enable mouse actions to produce interrupts.

Syntax

```
SUBROUTINE SET_MOUSE_INTERRUPT_MASK@(MASK)
INTEGER*2 MASK
```

Description This routine causes certain mouse actions to produce interrupts. The

SET_TRAP@ routine or its equivalent must first be called with a trap code of 4 to trap mouse events. Each bit in the mask corresponds to an event that may be trapped. The least significant bit is bit 0.

bit	interrupt
0	interrupt on cursor position change
1	interrupt on left button press
2	interrupt on left button release
3	interrupt on right button press
4	interrupt on right button release

Thus MASK = 2+8 = 10 gives an interrupt on left and right button presses.

Example

```
C Use of interrupts on cursor movement and button presses
      EXTERNAL MOUSE_TRAP
      INTEGER*4 Q
      INTEGER*2 CURSOR_H,CURSOR_V,BUTTON_STATUS
      COMMON CURSOR_H,CURSOR_V,BUTTON_STATUS
C Set up a trap for mouse events (code 4 specifies mouse)
      CALL SET_TRAP@(MOUSE_TRAP,Q,4)
C Say we want to interrupt on cursor movement and button
C press
      CALL SET_MOUSE_INTERRUPT_MASK@(11)
C Perform some process which uses the
C Cursor coordinates from time to time
      .
      .
      INTERRUPT SUBROUTINE MOUSE_TRAP
      INTEGER*2 CURSOR_H,CURSOR_V,BUTTON_STATUS
      COMMON CURSOR_H,CURSOR_V,BUTTON_STATUS
      CALL GET_MOUSE_POSITION@(CURSOR_H,CURSOR_V,BUTTON_STATUS)
      END
```

SET_MOUSE_MOVEMENT_RATIO@ ❸

Purpose To set the mouse cursor sensitivity.

Syntax

```
SUBROUTINE SET_MOUSE_MOVEMENT_RATIO@(IH,IV)
INTEGER*2 IH,IV
```

Description This routine sets the cursor sensitivity to horizontal and vertical changes in the mouse position. It may also be used to adjust the mouse movements to changes in the horizontal and vertical pixel sizes so that, for example, a circular movement of the mouse causes circular movement of the mouse cursor. The larger the values of IH and IV, the less sensitive is the mouse cursor to physical mouse movements. IH=8*m* and IV=8*n* where *m* is the horizontal and *n* the vertical number of mickeys per pixel. The minimum value for IH and for IV is 1. The default values are likely to be IH=8 and IV=16.

SET_MOUSE_POSITION@

Purpose To move the mouse cursor to a particular position.

Syntax
```
SUBROUTINE SET_MOUSE_POSITION@(IH,IV)
INTEGER*2 IH,IV
```

Description This routine sets the hot-spot of the mouse cursor to the position (IH, IV) on the screen. The coordinates are in pixels from the top left of the screen.

SET_MOUSE_SENSITIVITY@ ❸

Purpose To set the mouse cursor sensitivity and the threshold for the double speed.

Syntax
```
SUBROUTINE SET_MOUSE_SENSITIVITY@(DX,DY,SPEED)
INTEGER*2 DX,DY,SPEED
```

Description This routine combines the actions of the routines SET_MOUSE_MOVEMENT_RATIO@ and SET_MOUSE_SPEED_THRESHOLD@. Please refer to these routines for further details.

SET_MOUSE_SPEED_THRESHOLD@ ❸

Purpose To set the threshold for double speed.

Syntax
```
SUBROUTINE SET_MOUSE_SPEED_THRESHOLD@(SPEED)
INTEGER*2 SPEED
```

Description This routine sets the mouse double speed threshold. When the mouse is moved at a speed above SPEED mickeys per second the mouse cursor will move across

the screen at double speed.

SET_MOUSE_TEXT_CURSOR@ ❸

Purpose To specify details of the mouse cursor for text mode.

Syntax

```
SUBROUTINE SET_MOUSE_TEXT_CURSOR@(SELECT,IA,IB)
INTEGER*2 SELECT,IA,IB
```

Description This routine allows the user to specify details of the cursor for text mode. There are two possible types of cursor, the "hardware" cursor and the "attribute cursor".

By default, the hardware cursor is usually the familiar flashing under-score character which otherwise relates to keyboard input. By setting SELECT=1, this hardware cursor is assigned to the mouse position. In this mode the value assigned to IA is the start line of the rectangle and IB is the end line (IA=6, IB=7 is often the default) and these values have the same effect as parameters in SET_CURSOR_TYPE@. The effect will endure after the program terminates unless the default is reset. In this mode there will normally be only one visible cursor since the mouse takes over the hardware cursor.

The attribute cursor is more like the associated graphics cursor and can co-exist with the hardware cursor. By setting SELECT=0, this attribute cursor is assigned to the mouse position. In this mode the value assigned to IA is **and**ed with the underlying screen character and its attribute. The result is then **xor**ed with the value assigned to IB.

The screen character/attribute takes a two byte form with the low byte giving the ASCII code for the character and the high byte giving its attributes in the bit pattern fbbbtttt where f is set for a flashing character, bbb is the palette register for the background colour (0..7) and tttt is the palette register for the text colour (0..15).

Thus (0, Z'FFFF', Z'7700') first preserves the underlying character/attribute and then inverts the colour. Similarly (0, Z'FFFF', Z'F700') will be the same but will also invert the flashing mode. Likewise (0, Z'FF00', Z'7720') will remove the underlying character, replace it with a space (hex 20) and invert the colour.

Particular care needs to be taken with the attribute cursor since, although it is more flexible than the hardware cursor, it is susceptible to screen changes. For example, if the screen scrolls, the original attributes will not be restored at the old

cursor position, unless the cursor is first switched off.

13.

Printer (DOS)

The following routines drive the printer via BIOS calls. The printer number can be 1, 2, or 3 (LPT1, LPT2 or LPT3).

PRINT_CHARACTER@ ❷

Purpose To send one character to the printer.

Syntax
```
SUBROUTINE PRINT_CHARACTER@(CHAR,P)
CHARACTER CHAR
INTEGER*2 P
```

Description The printer number P should be in the range 1 to 3.

INITIALISE_PRINTER@ ❷

Purpose To initialise the printer.

Syntax
```
SUBROUTINE INITIALISE_PRINTER@(P)
INTEGER*2 P
```

Description The printer number P should be in the range 1 to 3.

GET_PRINTER_STATUS@ ❷

Purpose To obtain status information for the printer.

Syntax

```
SUBROUTINE GET_PRINTER_STATUS@(P,S)
INTEGER*2 P,S
```

Description S is returned with the status information for printer P. The printer number P should be in the range 1 to 3.

The bits of S have the following significance:

bit value	significance
1	Time-out
2	Reserved
4	Reserved
8	I/O error
16	Selected
32	Out of paper
64	Acknowledge
128	Not busy

14.

Process control

CISSUE

Purpose To issue a system command.

Syntax

```
SUBROUTINE CISSUE(A,IFAIL)
CHARACTER*(*) A
INTEGER*2 IFAIL
```

Description Issues the command stored as a character string in A. IFAIL is returned as one of the following:

IFAIL	Meaning
0	Successful invocation of a command processor to execute the command
1	A command processor could not be invoked

Notes The value of IFAIL refers to the success or failure of invoking the MS-DOS command processor COMMAND.COM. Unfortunately, MS-DOS does not provide a mechanism whereby the success or failure of the invocation of the particular command can be reported back to the caller. So, for example, if you get a system error such as "Not found" for the command, IFAIL will be returned as zero.

It is not possible to issue a command which itself uses DBOS (e.g. run another FTN77 program). Also the use of CISSUE to start TSR programs should be avoided since this can fragment memory.

Example

```
CALL COU@('the contents of this Directory are:-')
CALL CISSUE('DIR',K)
IF(K.NE.0)CALL COU@('DIR failed for some reason')
. . .
```

EXIT

Purpose To terminate a program.

Syntax

```
SUBROUTINE EXIT(ERROR_CODE)
INTEGER*2 ERROR_CODE
```

Description This routine terminates the program and returns to the operating system. If the error code is non zero the termination will be abnormal. Abnormal termination will interrupt the flow of a .BAT file (as if control break had been pressed).

EXIT@

Purpose To terminate a program.

Syntax

```
SUBROUTINE EXIT@(ERROR_CODE)
INTEGER*2 ERROR_CODE
```

Description EXIT@ is a synonym for EXIT.

GET_KEY_OR_YIELD@

Purpose To get the next keycode.

Syntax

```
SUBROUTINE GET_KEY_OR_YIELD@(KEY)
INTEGER*2 KEY
```

Description This routine returns a key typed on the keyboard in exactly the same way as GET_KEY@ except that it will yield control to other tasks (if any) when no keypress is pending. This routine is used in READ_EDITED_LINE@ and WREAD_EDITED_LINE@. Care should be taken to ensure that only one process is performing keyboard input with these routines at any one time - otherwise the characters will be shared randomly across several tasks!

See also SPAWN@, YIELD@.

SLEEP@

Purpose To suspend program execution for a specified time interval.

Syntax
```
SUBROUTINE SLEEP@(TIME)
REAL*4 TIME
```

Description The time is given in seconds and is accurate to within one tick (18.2 ticks per second).

SPAWN@ ❷

Purpose To initiate a concurrent subtask.

Syntax
```
SUBROUTINE SPAWN@(SUBTASK,STACK,STACKSIZE,HANDLE)
INTEGER*2 HANDLE
EXTERNAL SUBTASK
INTEGER*4 STACKSIZE
INTEGER*1 STACK(STACKSIZE)
```

Description SPAWN@ creates a subtask which executes routine SUBTASK (no arguments) concurrently with the rest of the program. Up to nine such subtasks may be created, though one will suffice for most purposes. The STACK array will hold the stack for the new task. It should be big enough to ensure that stack overflow cannot occur in the subtask. It is suggested that the array be made very large (say 10 Megabytes) and put in an uninitialised common block so that DBOS will only allocate memory for it as it is actually used. HANDLE is returned as an integer which defines the task.

See also YIELD@, GET_KEY_OR_YIELD@.

START_PROGRAM@ ❷

Purpose To start another Salford (DBOS) program.

Syntax
```
SUBROUTINE START_PROGRAM@(FILE,FLAGS)
```

```
CHARACTER*(*) FILE
INTEGER*4 FLAGS
```

Description This routine offers a way to start another Salford program (operating under DBOS) from within an FTN77 program (for example, another FTN77 program). Control will never return to the caller of START_PROGRAM@. The flags parameter is bit significant with the following meaning:

bit value	meaning
1	set if you want the program to behave as if /BREAK had been specified on the command line
2	set if underflows are to be treated (by default) as errors
4	set if you want /HARDFAIL

All other bits in the flags are reserved and must be set to 0.

Notes This routine can only be used to execute programs which have been compiled with FTN77 or one of its sister compilers.

The full pathname must be provided when the file is not in the current directory.

SET_COMMAND_LINE@ can be used to provide command line arguments for the call.

Example

```
CALL START_PROGRAM@('PROG.EXE',0)
END
```

YIELD@

❷

Purpose To yield control to a subtask.

Syntax

```
SUBROUTINE YIELD@(UNCONDITIONAL)
LOGICAL*2 UNCONDITIONAL
```

Description If the logical UNCONDITIONAL is set to .FALSE., YIELD@ will yield control to another task (initialised by SPAWN@) if one exists and the present task has run for about 2 clock ticks. If YIELD@ is called and does not actually yield control it consumes very little time, so it should be called in one of the loops of a task which is going to perform a long calculation.

If UNCONDITIONAL is set to .TRUE., YIELD@ always yields control if another task exists. The routine is called in this way from routines such as GET_KEY_OR_YIELD@ when no key is available. YIELD@ will just return if SPAWN@ has never been called or if all subtasks have been completed.

15.

Random numbers

RANDOM

Purpose To return a pseudo-random double precision value.

Syntax

```
DOUBLE PRECISION FUNCTION RANDOM()
```

Description This routine sets its seed automatically and produces the same sequence every time the program is run.

Alternatively, you may use DATE_TIME_SEED@ or SET_SEED@ which are described below.

Return value RANDOM returns a uniformly distributed random number x such that $0.0D0 < x \leq 1.0D0$.

Example

```
      DOUBLE PRECISION RANDOM,RANVEC(100)
      DO 1 I=1,100
    1 RANVEC(I)=RANDOM()
      . . .
```

DATE_TIME_SEED@

Purpose To select a new "seed" for the pseudo-random number generator function RANDOM.

Syntax

```
SUBROUTINE DATE_TIME_SEED@
```

Description This routine sets the seed for the random number generator to a value based on the current DATE/TIME. This routine is used to obtain a non repeatable sequence of pseudo-random numbers.

SET_SEED@

Purpose To enter a new "seed" for the pseudo-random number generator function RANDOM.

Syntax

```
SUBROUTINE SET_SEED@(SEED)
REAL*8 SEED
```

Description This routine sets the seed for the random number generator to a value based on SEED. SEED may take any value. Each value produces a repeatable sequence of pseudo-random numbers.

16.

Real mode interface (DOS)

The following routines are used to interface FTN77 software with code executing in real mode. They should be used in conjunction with chapter 26 of the *FTN77 User's Guide*.

It is not possible simply to call real mode functions from a protected mode FTN77 program. A particular real mode memory address always corresponds to the same byte in physical memory, whereas because of the virtual memory scheme used by FTN77, a particular virtual memory address can even refer to different addresses in physical memory in the course of one run of a single program, as the page which it belongs to gets paged out and paged in again (see chapter 23 of the *FTN77 User's Guide*). In addition to this, a particular virtual address may, at a given time, correspond to a physical address in extended memory, and thus not be directly accessible from a real mode program.

These factors mean that there is no simple way to provide a mapping for data held in memory in protected mode to that in real mode. That is not to say it is impossible to call real mode software - in fact, FTN77 has itself to call DOS and BIOS functions extensively, and these are real mode functions. A number of options exist whereby a user can invoke real mode code. These are outlined below:

The simplest method is to make the real mode code into a free-standing application, and invoke it with the CISSUE routine. Any data that needs to be passed to and from the real mode application can be written to a file.

If the code to be invoked is an interrupt handler, and the required data can be passed in registers, it is simply a matter of loading up the required registers and generating the appropriate interrupt. This can be done in a CODE/EDOC sequence, or by using the routine REAL_MODE_INTERRUPT@. Many DOS and BIOS functions can be invoked directly in this way.

Also, the user can write a TSR program which hooks an interrupt and is invoked from a FTN77 program by this mechanism. If the code to be invoked is an interrupt handler, but requires more information than can be returned in the registers, then a

mechanism exists using what is termed the DOSCOM buffer. This mechanism was designed primarily to allow those DOS interrupts which perform some data transfer (and therefore require a data buffer) to be invoked, but it can be used for other applications.

If the code to be invoked is not in the form of an interrupt handler, then a number of functions are provided to set up and perform the real mode call. This makes it possible to write a general purpose binding to a real mode library which will make it appear as if it had a simple call interface, and this is done in several available bindings for popular libraries.

Much of the material which follows assumes some knowledge of Intel 32-bit and MS-DOS architecture. For those not aquainted with these subjects several good references are available.

A real mode address of the form SELECTOR:ADDRESS is equivalent to an absolute address of LS(SELECTOR,4)+ADDRESS.

Programs compiled with real mode compilers and accessed from within an FTN77 program must use either large, compact, or huge models. Failure to do this leads to an interface failure and often requires a machine reboot.

ALLOCATE_REAL_MODE_MEMORY@ ❷

Purpose To allocate real mode memory.

Syntax

```
SUBROUTINE ALLOCATE_REAL_MODE_MEMORY@(POINTER,NBYTES,ICODE)
INTEGER*4 POINTER,NBYTES
INTEGER*2 ICODE
```

Description This routine causes DOS to allocate NBYTES of real mode memory and returns its address in POINTER. This is a real-mode (20-bit) address which is returned by the routine. ICODE is returned as zero if the call is successful.

Notes

- ALLOCATE_REAL_MODE_MEMORY@ is normally called before copying to and from real mode memory unless copying takes place to and from existing BIOS memory locations.
- The FTN77 program can reference any or all of the allocated real mode memory by using POINTER as a base, to which suitable offsets may be added in calls of COPY_FROM_REAL_MODE1@ and COPY_TO_REAL_MODE1@

COPY_FROM_REAL_MODE@ ❷

Purpose To copy data from a real mode program.

Syntax

```
SUBROUTINE COPY_FROM_REAL_MODE@(ANY_VARIABLE,NBYTES)
INTEGER*4 NBYTES
```

Description This routine copies NBYTES bytes from the data area previously specified in the real mode program by calling the real mode routine FTN77WT, to the data area in the FTN77 protected mode program specified by the argument ANY_VARIABLE. In practice, it is convenient to make ANY_VARIABLE the first word of a common block, thus ensuring a contiguous area of data, and to use ANY_VARIABLE as a flag to communicate actions to the real mode program.

Notes

- NBYTES *must* be of type INTEGER*4.
- ANY_VARIABLE can be a variable of any type.
- A call to COPY_FROM_REAL_MODE@ must not be made until LOAD_REAL_MODE_LIBRARY@ has been called from the FTN77 program, and FTN77WT has been called to return from the real mode program.

COPY_FROM_REAL_MODE1@ ❷

Purpose To copy data from a real mode program.

Syntax

```
 SUBROUTINE COPY_FROM_REAL_MODE1@(ANY_VARIABLE,NBYTES,
+  ADDRESS)
 INTEGER*4 NBYTES,ADDRESS
```

Description Copies NBYTES of information from the absolute real mode address space at the given address into ANY_VARIABLE. The real mode address must be less that 1 Megabyte.

COPY_FROM_SEGMENT@ ❷

Purpose To copy data from another segment.

Syntax
```
SUBROUTINE COPY_FROM_SEGMENT@(DATA,SELECTOR,OFFSET,NBYTES)
INTEGER*2 SELECTOR
INTEGER*4 OFFSET,NBYTES
```

Description Copies data to the variable or array DATA (which may be of any type) from a separate segment (e.g. the screen segment). OFFSET is the position within the segment from which the data is obtained.

COPY_TO_REAL_MODE@ ❷

Purpose To copy data to a real mode program.

Syntax
```
SUBROUTINE COPY_TO_REAL_MODE@(ANY_VARIABLE,NBYTES)
INTEGER*4 NBYTES
```

Description This routine copies NBYTES bytes to the data area previously specified in the real mode program by calling the real mode routine FTN77WT, from the data area in the FTN77 protected mode program specified by the argument ANY_VARIABLE.

In practice, it is convenient to make ANY_VARIABLE the first word of a common block, thus ensuring a contiguous area of data, and to use ANY_VARIABLE as a flag to communicate actions to the real mode program.

Notes

- NBYTES *must* be of type INTEGER*4.
- ANY_VARIABLE can be a variable of any type.
- A call to COPY_TO_REAL_MODE@ must not be made until LOAD_REAL_MODE_LIBRARY@ has been called from the FTN77 program, and FTN77WT has been called from the real mode program.

COPY_TO_REAL_MODE1@ ❷

Purpose To copy data to a real mode program.

Syntax
```
SUBROUTINE COPY_TO_REAL_MODE1@(ANY_VARIABLE,NBYTES,ADDRESS)
INTEGER*4 NBYTES,ADDRESS
```

Description Copies NBYTES of information to the absolute real mode address space at the given address from ANY_VARIABLE. The real mode address must be less that 1 Megabyte. This routine must be used with great care, as it is possible to corrupt DOS or DBOS if an unsuitable address is specified.

COPY_TO_SEGMENT@ ❷

Purpose To copy data to another segment.

Syntax
```
SUBROUTINE COPY_TO_SEGMENT@(DATA,SELECTOR,OFFSET,NBYTES)
INTEGER*2 SELECTOR
INTEGER*4 OFFSET,NBYTES
```

Description Copies data from the variable or array DATA (which may be of any type) to a separate segment (e.g. the screen segment). OFFSET is the position within the segment at which the data should be put.

DEALLOCATE_REAL_MODE_MEMORY@ ❷

Purpose To free real mode memory.

Syntax
```
SUBROUTINE DEALLOCATE_REAL_MODE_MEMORY@(POINTER,ICODE)
INTEGER*4 POINTER
INTEGER*2 ICODE
```

Description Deallocates real mode memory that was previously obtained with ALLOCATE_REAL_MODE_MEMORY@. POINTER must be a pointer returned by ALLOCATE_REAL_ MODE_MEMORY@.

DOSCOM@ ❷

Purpose To obtain a segment selector for the DOSCOM buffer.

Description This routine returns a segment selector in FS for the 1K DOSCOM buffer which is arranged so as to overlap the real mode space. This allows system interrupts to be used which require a buffer to be passed (e.g. I/O transfer operations). When an SVC/3 call is issued from a CODE/EDOC (in-line assembler) sequence, the DS and ES registers are set up to point at the DOSCOM buffer.

By its nature this routine is only useful from within a CODE/EDOC sequence. See page 316 of the *FTN77 User's Guide* for an example of its use.

FTN77WT etc. ❷

Purpose Used within a real mode program to receive control from and return control to a FTN77 program.

Syntax
```
SUBROUTINE FTN77WT(ANY_VARIABLE)
```

Description FTN77WT is one of a number of routines that can be used within a real mode program. Each routine is appropriate to a particular real mode compiler. These routines are used both to indicate where execution of the real mode program should start when REAL_MODE@ is called from the FTN77 program and to transfer control back to the FTN77 program. The routine is called each time this transfer is required.

It is usually convenient to use the single argument, ANY_VARIABLE, as a flag to indicate an action to be taken by the calling FTN77 program. In this case the argument should be declared as INTEGER*4, INTEGER*2 or INTEGER*1 in both the real mode and protected mode programs.

Notes

- The FTN77 program acts as a driver for the "slave" real mode program which calls FTN77WT.
- The source code for these routines is located in the DBOS.DIR directory, ready for compilation using your real mode assembler. You should select the one that is appropriate to your particular compiler. They are also supplied pre-assembled, ready for inclusion in a real mode object library.
- FTN77WT has been used successfully with IBM Professional Fortran. As the method of argument passing may differ amongst real-mode compilers, it

may not function correctly when called, for example, from a program compiled with the Lahey F77L compiler.

LINEAR_ONE_MEG_SEG@ ❷

Purpose To obtain the real mode address 0.

Syntax
```
SUBROUTINE LINEAR_ONE_MEG_SEG@
```

Description Returns with the FS segment selector pointing to real mode address 0. The segment is a one megabyte segment so the whole of real mode memory can be accessed off the segment. This is really only of use for machine code programmers.

It is advised that the routines COPY_FROM_REAL_MODE1@ and COPY_TO_REAL_MODE1@ be used in preference.

LOAD_REAL_MODE_LIBRARY@ ❷

Purpose To load and execute a real mode program.

Syntax
```
SUBROUTINE LOAD_REAL_MODE_LIBRARY@(REAL_MODE_EXE_FILE)
CHARACTER *(*) REAL_MODE_EXE_FILE
```

Description This routine loads, and starts to execute, a previously compiled and linked real mode program so that it can be called from a protected mode program. The argument, REAL_MODE_EXE_FILE, is a filename or pathname of a suitable MS-DOS .EXE file.

Notes

- LOAD_REAL_MODE_LIBRARY@ must be the first routine called by the FTN77 program which wishes to communicate with a real mode program.
- This routine both loads and starts to execute the real mode program. Execution of the real mode program starts with the first statement. Real mode routine FTN77WT must be called by the real mode program in order to return to the FTN77 program.
- Programs compiled with real mode compilers and accessed from within an FTN77 program must use either large, compact, or huge models.

MODIFY_REAL_MODE_MEMORY@ ❷

Purpose To change the size of a block of real mode memory.

Syntax
```
SUBROUTINE MODIFY_REAL_MODE_MEMORY@(POINTER,NBYTES,ICODE)
INTEGER*4 POINTER,NBYTES
INTEGER*2 ICODE
```

Description This attempts to change the size of a block of real mode memory previously allocated with ALLOCATE_REAL_ MODE_MEMORY@. POINTER must be a pointer returned by ALLOCATE_REAL_MODE_MEMORY@. NBYTES should be set to the size in bytes of the new block. ICODE will be returned as zero if the new size is acceptable. It is much easier to shrink a block than to enlarge it.

REAL_MODE@ ❷

Purpose To transfer control from a FTN77 to a real mode program.

Syntax
```
SUBROUTINE REAL_MODE@
```

Description This routine is used to transfer control from a FTN77 program to a real mode program which has been loaded by LOAD_REAL_MODE_LIBRARY@. Each time REAL_MODE@ is called, control is transferred to the statement which immediately follows the call of FTN77WT in the real mode program.

Notes

- A call to REAL_MODE@ must not be made until LOAD_REAL_MODE_LIBRARY@ has been called from the FTN77 program, and FTN77WT has been called to return the real mode program.
- Real mode routine FTN77WT must be called by the real mode program in order to return to the FTN77 program.

REAL_MODE_ADDRESS_OF_DOSCOM@ ❷

Purpose To obtain the address of the DOSCOM buffer.

Syntax
```
SUBROUTINE REAL_MODE_ADDRESS_OF_DOSCOM@(ADDRESS,SELECTOR)
INTEGER*4 ADDRESS
INTEGER*2 SELECTOR
```

Description Returns the absolute address of the 1K byte DOSCOM buffer and a protected mode segment selector. Note that this is *not* a real-mode selector:address pair.

A real mode address of the form SELECTOR:ADDRESS is equivalent to an absolute address of LS(SELECTOR,4)+ADDRESS.

REAL_MODE_INTERRUPT@ ❷

Purpose To cause a real mode interrupt from an FTN77 program.

Syntax

```
SUBROUTINE REAL_MODE_INTERRUPT@(REGISTERS,INTERRUPT)
INTEGER*2 REGISTERS(10),INTERRUPT
```

Description INTERRUPT contains an interrupt number. The elements of REGISTERS have the correspondence with the real mode registers as shown in the table below.

Notes

- Element 10 of REGISTERS (FLAGS) is returned. Its value before the call has no significance.
- This subroutine may only be called from an FTN77 program.

REGISTER ELEMENT	REAL MODE REGISTER
1	AX
2	BX
3	CX
4	DX
5	SI
6	DI
7	BP
8	DS
9	ES
10	FLAGS

SCREENSEG@ ❷

Purpose To obtain the segment selector for the graphics area.

Syntax
```
SUBROUTINE SCREENSEG@
```

Description Returns with the FS segment selector pointing to real mode address Z'A0000', the graphics area. This is really only of use for machine code programmers.

17.

Serial communications

GETTERMINATECOMMCHAR@

Purpose To get the character that terminated the last call to READCOMMDEVICE@.

Syntax
```
CHARACTER GETTERMINATECOMMCHAR@(PORTNUM)
INTEGER*4 PORTNUM
```

Return value GETTERMINATECOMMCHAR@ returns the character that terminated the last READCOMMDEVICE@ call. This will be one of the characters set using SETCOMMTERMINATECHAR@. If the port is not open the function returns -1. If READCOMMDEVICE@ has not been called the default return is zero.

OPENCOMMDEVICE@

Purpose To open a serial port for I/O.

Syntax
```
INTEGER*4 OPENCOMMDEVICE@(PORTNUM,COMSPEC,RSIZE,TSIZE)
INTEGER*4 PORTNUM, RSIZE,TSIZE
CHARACTER*(*) COMSPEC
```

Description To initiate serial communications between the computer and external devices a communications port must be selected and opened. On a standard PC there is a maximum of four serial ports, although it is common for only two to be installed. PORTNUM can therefore be either 1, 2, 3 or 4. Port 1 is commonly used to connect the mouse and so may not be available.

COMSPEC is a string that specifies the baud rate, parity, data and stop bit information (e.g. '9600, n, 8, 1'). Possible values are:

baud rate:	300,600,1200,2400,4800,9600,19200,38400,57600,115200
parity:	n (none), o (odd), e (even)

data bits: 7 or 8
stop bits: 0 or 1

Under Windows 3.1(1) it is necessary to specify a size for input and output buffers. The size of the input buffer RSIZE and the size of the output buffer TSIZE should be set at about 1024. On slower systems with high data rates it may be advisable to specify larger values.

Return value OPENCOMMDEVICE@ returns a positive value when successful otherwise it returns -1.

READCOMMDEVICE@

Purpose To read data from an open serial port.

Syntax
```
INTEGER*4 READCOMMDEVICE@(PORTNUM,STRING,NREAD)
INTEGER*4 PORTNUM,NREAD
CHARACTER*(*) STRING
```

Description Use this function to read data from a serial port that has been opened by a call to OPENCOMMDEVICE@. PORTNUM must be a valid port number in the range 1 to 4 and NREAD is set as the maximum number of characters to be read. After the call STRING will contain the data held in the serial port up to NREAD characters. Fewer than NREAD characters will be read if one of the termination characters (set using SETCOMMTERMINATECHAR@) is encountered.

Return value READCOMMDEVICE@ returns the number of characters read or -1 if an error occurred.

SETCOMMTERMINATECHAR@

Purpose To set the characters that may be used to terminate a call to READCOMMDEVICE@ .

Syntax
```
INTEGER*4 SETCOMMTERMINATECHAR@(PORTNUM,STRING,LEN)
INTEGER*4 PORTNUM,LEN
CHARACTER*(*) STRING
```

Description PORTNUM must be a valid serial port number in the range 1 to 4. STRING contains a string of length LEN. This string is used to provide a list of characters

that may be used to terminate a call to READCOMMDEVICE@. The termination character is discarded when READCOMMDEVICE@ is called. For example `char(0)//char(12)//char(15)` provides for the data to be terminated by a null character, a line feed or a carriage return.

Return value SETCOMMTERMINATECHAR@ returns a positive value if successful or -1 if an error has occurred.

SETECHOONREADCOMM@

Purpose To set the communication port to echo back to the sending device.

Syntax

```
INTEGER*4 SETECHOONREADCOMM@(PORTNUM,STATE)
INTEGER*4 PORTNUM,STATE
```

Description When communicating with a serial device such as a terminal, it is often necessary to return the data to the sender. In the case of a terminal the data will be transmitted from its keyboard to the host computer. SETECHOONREADCOMM@ is used to enable (STATE = 1) or disable (STATE = 0) the echoing of data. Data that is echoed back to a terminal will be displayed on its VDU. PORTNUM is a valid port number in the range 1 to 4.

Return value SETECHOONREADCOMM@ returns a positive value if successful or -1 if an error has occurred.

WRITECOMMDEVICE@

Purpose To write a string to a serial port.

Syntax

```
INTEGER*4 WRITECOMMDEVICE@(PORTNUM,STRING)
INTEGER*4 PORTNUM
CHARACTER*(*) STRING
```

Description Use this function to write a string to a serial port that has been opened by a call to OPENCOMMDEVICE@. PORTNUM must be a valid port number in the range 1 to 4.

Return value The function returns the number of characters written or -1 if an error has occurred.

18.

Sound

BEEP@ ❷

Purpose To output an audible beep.

Syntax

```
SUBROUTINE BEEP@
```

SOUND@ ❷

Purpose To make an audible sound at the console.

Syntax

```
SUBROUTINE SOUND@(FREQUENCY,TIME)
INTEGER*2 FREQUENCY,TIME
```

Description Produces a tone of FREQUENCY hertz for a time measured in ticks. There are approximately 18 ticks per second.

19.

Storage management

The routines described in this chapter fall into three main categories:

- Provision of a virtual storage heap.
- Control over and information about the virtual memory environment provided by the DBOS DOS extender.
- A facility to make Fortran scratch files “memory resident”.

The virtual memory heap is located in memory above the stack, and has an initial size of 100 Megabytes. Since these routines work with addresses, the storage acquired by these routines must be manipulated by the ‘core’ intrinsics. As with any storage heap, it is important to avoid excessive fragmentation. This can be achieved by a variety of strategies, such as allocating blocks of fixed size, or deallocating all allocated storage at once, so that no ‘holes’ are created. The first fit algorithm is used by the routines. Block sizes are rounded up to multiples of 4 bytes in size and carry a maximum of a 16-byte overhead. This overhead is reduced when many blocks are allocated and can be reduced to as little as 4 bytes.

It is not necessary to return allocated storage before a program terminates - this is done automatically.

For Win32 there are two separate heaps, both 100Mb.

a) A heap used by GET_STORAGE@, RETURN_STORAGE@ and SHRINK_STORAGE@. This is fully virtual. Pages are provided by the runtime system in order to fill program page demands. The program may fail if it uses too much of the address space allocated, if the physical resources n the system are not sufficient to satisfy the program’s demands. This is the same as under DBOS.

b) A C/C++ heap used by **malloc**, **new** etc. in order to provide physical (committed) pages. Memory allocated from this heap is guaranteed to be available.

For both DBOS and Win32, memory allocated by GET_STORAGE@ should not be returned using **free** or **delete**. Similarly, memory allocated using **malloc** or **new** should not be returned using RETURN_STORAGE@.

FREE_SPACE_AVAILABLE@ ❷

Purpose To obtain the amount of free memory in the system.

Syntax
```
INTEGER*4 FUNCTION FREE_SPACE_AVAILABLE@()
```

Return value The value returned is the number of bytes available in free memory pages.

FREE_VIRTUAL_PAGES@ ❷

Purpose To free memory for reuse.

Syntax
```
SUBROUTINE FREE_VIRTUAL_PAGES@(ADDR,N)
INTEGER*4 ADDR,N
```

Description This routine provides the capability to reuse memory that is no longer needed. You may, for example, have a large array that is used in a calculation and is then not needed for the rest of the program. This function allows the memory taken by the array to be freed and used for some other purpose. The routine should be used with care, the freed array is still accessible but it will contain random values. ADDR is the address of the memory to free, N is the size in bytes.

It is not necessary to use this routine with memory allocated with GET_STORAGE@, the memory taken will be passed back to the system with the corresponding RETURN_STORAGE@ call.

Example

```
      SUBROUTINE WASTE_MEMORY
      INTEGER*4 A(100000)
      SAVE A
      ...
C COULD USE A(1), WE ARE ACTUALLY PASSING A REFERENCE
      CALL FREE_VIRTUAL_PAGES@(A,100000*4)
      END
```

GET_MEMORY_INFO@ ❷

Purpose To obtain information about the memory.

Syntax
```
SUBROUTINE GET_MEMORY_INFO@(NP1,NP2,NP3,NP4,NP5,NP6,NP7)
INTEGER*4 NP1,NP2,NP3,NP4,NP5,NP6,NP7
```

Description Obtains information about the amounts of various sorts of memory in the system. The size of a page is 4096 bytes.

NP1 = Total available pages beneath 640K (DOS memory)
NP2 = Total available pages above 1 Megabyte (extended memory)
NP3 = Remaining DOS pages
NP4 = Remaining extended pages
NP5 = Total disk swap pages
NP6 = Remaining disk swap pages
NP7 = Number of page turns since program start

A page turn is defined as the process whereby useful data is removed from memory to make way for something else. A program will generate no page turns if it is executed with sufficient memory.

GET_STORAGE@

Purpose To get a block of storage of size N bytes from the storage heap.

Syntax
```
SUBROUTINE GET_STORAGE@(ADDR,N)
INTEGER*4 ADDR,N
```

Description ADDR is returned as the address of the first byte of the block. Under DBOS a returned value of -1 indicates that there is insufficient contiguous storage to create the block. Under Win32 a returned value of zero indicates that there is insufficient contiguous storage to create the block.

Notes The heap used by GET_STORAGE@, RETURN_STORAGE@ and SHRINK_STORAGE@ is fully virtual. Pages are provided by the runtime system to satisfy the program's page demands. The program may fail if it uses too much of the address space allocated or if the physical resources on the system are not sufficient to satisfy the program's demands.

Memory allocated using GET_STORAGE@ should be returned using RETURN_STORAGE@. The C function **free** and the C++ operator **delete** can not be used for this purpose.

GET_STORAGE1@ ❷

Purpose To get a block of storage from the storage heap.

Syntax
```
SUBROUTINE GET_STORAGE1@(ADDR,N)
INTEGER*4 ADDR,N
```

Description Gets a block of storage of at least size N bytes from the storage heap. ADDR is returned as the address of the first byte of the block or -1 (for DBOS, zero for Win32) if there is insufficient contiguous storage to create the block. This routine never splits a contiguous area of storage. N is returned with the actual size of block allocated.

One use of this is to allocate a block using GET_STORAGE1@ for storing data whose size you do not know in advance, and then using SHRINK_STORAGE@ to set the block to the size required when all the data has been collected.

Notes See GET_STORAGE@.

LARGEST_BLOCK_AVAILABLE@ ❷

Purpose To obtain the size of the largest free block in the storage heap.

Syntax
```
SUBROUTINE LARGEST_BLOCK_AVAILABLE@(AMOUNT)
INTEGER*4 AMOUNT
```

Description AMOUNT is returned as the size required. The value does not indicate that there is enough physical memory to allocate this amount. This block can be obtained by calling GET_STORAGE@.

MEMORY_AVAILABLE@ ❷

Purpose To get the total size of available heap space.

Syntax
```
SUBROUTINE MEMORY_AVAILABLE@(AMOUNT)
INTEGER*4 AMOUNT
```

Description AMOUNT is returned as the size required. The value does not indicate that there is enough physical memory to allocate this amount. Calls to GET_STORAGE@ can be used to obtain pieces of this space.

RETURN_STORAGE@

Purpose To return a block of storage.

Syntax
```
SUBROUTINE RETURN_STORAGE@(ADDR)
INTEGER*4 ADDR
```

Description Returns a block of storage previously allocated by one of the storage management routines. ADDR must be the address of the start of the storage block to be returned.

SET_PAGES_RESERVE@ ❷

Purpose To warn of a limited page reserve.

Syntax
```
SUBROUTINE SET_PAGES_RESERVE@(N)
INTEGER*4 N
```

Description Specifies that the system should generate the fault “Down to pages reserve” (or take a trap) when the total number of pages remaining in store or on the disk has dropped to N.

SET_TRAP_ON_PAGE_TURN@ ❷

Purpose To warn of the first page turn.

Syntax
```
SUBROUTINE SET_TRAP_ON_PAGE_TURN@
```

Description Specifies that the system should generate the fault "Down to pages reserve" (or take a trap) when the first page turn is generated. This routine uses SET_PAGES_RESERVE@. A typical use for this routine would be in a program package which you do not want to run slowly due to page swapping. By using SET_TRAP@ to trap the event you could ensure that, if the package is run on a machine with insufficient memory, a suitable diagnostic will be generated.

SHRINK_STORAGE@

Purpose To shrink a block of storage.

Syntax

```
SUBROUTINE SHRINK_STORAGE@(ADDR,N)
INTEGER*4 ADDR,N
```

Description Shrinks a block of storage previously allocated by one of the storage management routines. ADDR must be the address of the start of the storage block whose size is to be adjusted. N is the new size of the block. This routine cannot be used to enlarge a storage block.

USE_STORAGE@ ❷

Purpose To offer additional memory to the storage heap.

Syntax

```
SUBROUTINE USE_STORAGE@(ADDR,N)
INTEGER*4 ADDR,N
```

Description ADDR is the address of the memory (e.g. the address of an array in common) and N is the number of bytes being offered.

Notes Once memory has been given to the storage heap in this way it must not be referenced in any other way.

Example

```
C       THIS PROGRAM WILL ENLARGE THE STORAGE MAP TO 20 MBYTES
C       CALLS TO GET_STORAGE@ ETC. WILL FOLLOW LATER
        CHARACTER X(10000000)
        COMMON/HEAP/X
        CALL USE_STORAGE@(LOC(X),10000000)
```

USE_VIRTUAL_SCRATCH_FILES@ ❶

Purpose To enable or disable the virtual scratch file facility.

Syntax

```
SUBROUTINE USE_VIRTUAL_SCRATCH_FILES@(OPTION)
LOGICAL*2 OPTION
```

Description Enables or disables (according to whether OPTION is true or false) the virtual scratch file facility. When enabled, any scratch file which is created by an OPEN statement (STATUS='SCRATCH') will be held in virtual memory and never explicitly written to disk. This routine provides a simple way to accelerate programs which were written for a small address space (640K) and which consequently write data out to temporary files. Even if some of the data ends up being paged to disk the use of this routine will usually give substantial performance gains. It is the user's responsibility to ensure that there is sufficient memory (real and/or virtual) to accommodate the files. Currently no one file may exceed 41 Megabytes in size.

If you are writing software to run in a variety of environments it may be useful to call GET_MEMORY_INFO@ in order to decide whether to call this routine.

Note that, once created, a virtual scratch file does not change type if the virtual scratch file facility is turned off. Equally, existing ordinary scratch files are not affected by a call to this routine. This means that it is possible to force some scratch files to use virtual memory, while others are still written to disk.

20.

System information

DBOS_VERSION@ ❷

Purpose To get the current DBOS version number.

Syntax
```
SUBROUTINE DBOS_VERSION@(VERSION)
CHARACTER*6 VERSION
```

Description VERSION is returned as the version number of DBOS that the current program is running under (*not* necessarily the version the program was compiled with). The result is a character string to allow version numbers of the form 1.23. This routine was added at version 2.60, so it may be wise to test the presence of the routine with a call to DYNT@.

DOSPARAM@

Purpose To get a DOS environment parameter value.

Syntax
```
SUBROUTINE DOSPARAM@(PARAM,VALUE)
CHARACTER*(*) PARAM,VALUE
```

Description This routine returns the value VALUE of a DOS parameter PARAM, which has been set using the DOS SET command. This can be very useful while creating environments in which programs are controlled by global information set up in batch files.

Example After the DOS command

```
SET FILENAME=FRED
```

has been executed, the following would open the file FRED:

```
CHARACTER*50 FILE
CALL DOSPARAM@('FILENAME',FILE)
OPEN(FILE=FILE,UNIT=6)
```

DYNT@ ❷

Purpose To test for the presence of a system routine.

Syntax
```
SUBROUTINE DYNT@(NAME,RESULT)
CHARACTER*(*) NAME
INTEGER*4 RESULT
```

Description DYNT@ tests to see if NAME is the name of a system routine or a routine in an active dynamic link library.

If it is, RESULT is set to the address of the routine, otherwise it is set to zero. NAME may be in upper or lower case.

DYNT1@ ❷

Purpose To test for the presence of a user routine.

Syntax
```
SUBROUTINE DYNT1@(NAME,RESULT)
CHARACTER*(*) NAME
INTEGER*4 RESULT
```

Description DYNT1@ tests to see if NAME is the name of a routine in the user's program. If it is, RESULT is set to the address of the routine, otherwise it is set to zero. NAME may be in upper or lower case.

GET_COPROCESSOR_ENVIRONMENT@ ❷

Purpose To obtain the types of processors available on the system.

Syntax
```
SUBROUTINE GET_COPROCESSOR_ENVIRONMENT@(K)
INTEGER*2 K
```

Description This routine returns a bit significant result indicating the type(s) of coprocessor available on the system thus:

bit value	functionality
1	287 functionality
2	387 functionality
4	1167 Weitek functionality
8	3167 Weitek functionality
64	386 or 486 processor
128	Pentium processor

If bit 1 (bit value 2) is set then bit 0 will be also. Likewise if bit 3 is set so bit 2 will also be set.

GET_CURRENT_FORTRAN_IO@

Purpose To access the state of the current Fortran I/O unit.

Syntax

```
 SUBROUTINE GET_CURRENT_FORTRAN_IO@(UNIT,REC,RECL,STATUS,
+  NBYTES)
 INTEGER*2 UNIT,STATUS
 INTEGER*4 REC,RECL,NBYTES
```

Description This routine supersedes GET_CURRENT_FORTRAN_UNIT@. This allows a Fortran device driver (set up with the DEVICE= keyword - see page 113 of the *FTN77 User's Guide*) to access the state of the current unit. This is useful when a device driver is attached to multiple units. The current unit is returned in UNIT, the current record in REC and the current record length in RECL. NBYTES is filled with the number of bytes to read/write.

STATUS is filled thus:

Bit-0 set for FORMATTED, unset for UNFORMATTED I/O,
Bit-1 set for DIRECT, unset for SEQUENTIAL access.

GET_CURRENT_FORTRAN_UNIT@

Purpose To get the unit number for the current I/O operation.

Syntax

```
SUBROUTINE GET_CURRENT_FORTRAN_UNIT@(UNIT)
INTEGER*2 UNIT
```

Description This routine is useful in device drivers (using the DEVICE= I/O keyword) which are to be attached to more than one Fortran stream.

GETENV@ ❺

Purpose To get an environment variable.

Syntax

```
CHARACTER*(*) FUNCTION GETENV@(VARIABLE)
CHARACTER*(*) VARIABLE
```

Return value Returns the value of the specified environment variable.

21.

Text screen/keyboard

The routines in this chapter provide facilities for screen and keyboard I/O and control. Note that routines which control the graphics aspects of the screen are described in chapters 7 and 8.

While the routines described here are nominally for screen and keyboard, they can more accurately be described as for standard output and standard input. As such, DOS I/O redirection will work for these routines.

Certain subroutines which display text on the screen require colour information for the text. This information appears as the least significant 8 bits of an integer (INTEGER*2) as follows:

```
Bit   7  6  5  4  3  2  1  0
      f  b  b  b  t  t  t  t

f    - set for flashing text
bbb  - background colour (between 0 and 7)
tttt - text colour (between 0 and 15)
```

For a list of the colour numbers see page 45. For example, to give flashing red text (colour 4) on a green background (colour 2) the attribute is

1*128 + 2*16 + 4 = 161

that is (f)*128 + (bbb)*16 + (tttt) where 128 and 16 are the appropriate offsets.

COU@

Purpose To output text to the screen with a new line.

Syntax

```
SUBROUTINE COU@(A)
CHARACTER*(*) A
```

Description COU@ takes the message length (which must be less than 1024 characters) from its character argument A.

See also COUA@, COUP@, SOU@, SOUA@.

Example

```
CALL COU@('message to screen')
END
```

COUA@

Purpose To output text to the screen without a new line.

Syntax

```
SUBROUTINE COUA@(A)
CHARACTER*(*) A
```

Description COUA@ takes the message length (which must be less than 1024 characters) from its character argument A. This is useful as a prompt or as part of a sequence of calls which build up a line on the screen.

See also COU@, COUP@, SOU@, SOUA@.

Example

```
CALL COUA@('enter number of samples: ')
READ *,N
. . .
```

COUP@ ❷

Purpose To output text to a given screen position.

Syntax

```
SUBROUTINE COUP@(STRING,ATTRIBUTE,ICOL,IROW)
CHARACTER*(*) STRING
INTEGER*2 ATTRIBUTE,ICOL,IROW
```

Description COUP@ writes STRING at position (ICOL,IROW) relative to (0,0) at the top left of the screen, with the colour information ATTRIBUTE (see the introduction to this chapter). COUP@ takes the message length (which must be less than 1024 characters) from its character argument STRING.

See also COU@, COUA@, SOU@, SOUA@.

Example

```
C       COLOUR 7 IS WHITE
        CALL COUP@('IN THE MIDDLE',7,35,12)
```

DOS_KEY_WAITING@ ❷

Purpose To test if the keyboard buffer is empty.

Syntax

```
LOGICAL*2 FUNCTION DOS_KEY_WAITING@()
```

Description This routine is identical to KEY_WAITING@ except that this routine respects DOS redirection.

Return value DOS_KEY_WAITING@ returns .TRUE. if there is a key waiting to be read from the keyboard buffer, .FALSE. if not.

ECHO_INPUT@ ❷

Purpose To control the echoing of text from standard input.

Syntax

```
SUBROUTINE ECHO_INPUT@(STATE)
LOGICAL*2 STATE
```

Description By default text is echoed to standard output. If STATE is given the value .TRUE. then echoing is performed, if it is .FALSE. then echoing is not performed.

ERRCOU@ ❺

Purpose To output text to the standard error device.

Syntax

```
SUBROUTINE ERRCOU@(STRING)
CHARACTER*(*) STRING
```

Description ERRCOU@ outputs text followed by a new line taking the message length from its character argument STRING.

ERRCOUA@ ❺

Purpose To output text to the standard error device.

Syntax
```
SUBROUTINE ERRCOUA@(STRING)
CHARACTER*(*) STRING
```

Description ERRCOUA@ outputs text (without a new line) taking the message length from its character argument STRING. This is useful as a prompt or as part of a sequence of calls which build up a line on the screen.

ERRNEWLINE@ ❺

Purpose To write a newline to the standard error device.

Syntax
```
SUBROUTINE ERRNEWLINE@
```

ERRSOU@ ❺

Purpose To output text to the standard error device.

Syntax
```
SUBROUTINE ERRSOU@(STRING)
CHARACTER*(*) STRING
```

Description ERRSOU@ outputs text from the character argument STRING to standard error omitting any trailing blanks, and outputing a new line.

ERRSOUA@ ❺

Purpose To output text to the standard error device.

Syntax
```
SUBROUTINE ERRSOUA@(STRING)
CHARACTER*(*) STRING
```

Description ERRSOUA@ outputs text from the character argument STRING to standard

error omitting any trailing blanks. Does not output a new line.

GET_CURSOR_POS@ ❷

Purpose To get the co-ordinates of the text cursor.

Syntax
```
SUBROUTINE GET_CURSOR_POS@(IH,IV)
INTEGER*2 IH,IV
```

Description This subroutine gets the horizontal position IH and the vertical position IV of the console screen text cursor. If the screen has 80x25 character positions then the range of IH is 0..79 and the range of IV is 0..24 with (0,0) at the top left-hand corner.

See also SET_CURSOR_POS@.

GET_DOS_KEY@ ❷

Purpose To get the next keycode.

Syntax
```
SUBROUTINE GET_DOS_KEY@(K)
INTEGER*2 K
```

Description This routine is similar to GET_KEY@ except that DOS redirection is respected.

GET_DOS_KEY1@ ❷

Purpose To get the waiting keycode.

Syntax
```
SUBROUTINE GET_DOS_KEY1@(K)
INTEGER*2 K
```

Description This routine is similar to GET_KEY1@ except that DOS redirection is respected.

GET_EXTENDED_CHAR@ ❷

Purpose To get the waiting two-byte keycode.

Syntax
```
SUBROUTINE GET_EXTENDED_CHAR@(K)
INTEGER*2 K
```

Description This subroutine gets a two-byte keycode from the keyboard. If the buffer is not empty then K is read and removed from the buffer. If the buffer is empty then K is set to zero and the function does not wait for a keyboard input. The high byte is the scan code for the key whilst the low byte is the ASCII value. Sometimes it is simpler to use the function GET_KEY@ which returns the keycode in a different form.

Example
```
CALL GET_EXTENDED_CHAR@(K)
IF(K.EQ.Z'3B00')THEN
  PRINT *,'F1 Key pressed'
ELSEIF(K.EQ.Z'13C')THEN
  :
  :
ENDIF
```

GET_KEY@

Purpose To get the next keycode.

Syntax
```
SUBROUTINE GET_KEY@(K)
INTEGER*2 K
```

Description This subroutine gets a keycode from the keyboard. If the buffer is not empty then a value is read and removed from the buffer. If the buffer is empty then the function waits for a keyboard input. The value of K assigned such that if the high byte is equal to 0 then the low byte is the ASCII value for the key pressed. If the high byte is equal to 1 then the low byte is the scan code for a Function key or an ALT key combination (the scan code is then K-256).

See also GET_KEY1@, GET_DOS_KEY@, GET_DOS_KEY1@, KEY_WAITING@, GET_KEY_OR_YIELD@.

Example

```
CALL GET_KEY@(K)
IF(K.EQ.Z'13B')THEN
  PRINT *,'F1 Key pressed'
ELSEIF(K.EQ.Z'13C')THEN
  :
  :
ENDIF
```

GET_KEY1@ ❷

Purpose To get the waiting keycode.

Syntax

```
SUBROUTINE GET_KEY1@(K)
INTEGER*2 K
```

Description GET_KEY1@ is the same as GET_KEY@ except that it does not wait for a key to be pressed when the buffer is empty. In this situation K is set to zero.

Example

```
        CALL GET_KEY1@(K)
C       Ensure a key has been pressed
        IF(K.EQ.0)GOTO 100
        IF(K.EQ.Z'13B')THEN
          PRINT *,'F1 Key pressed'
        ELSEIF(K.EQ.Z'13C')THEN
          :
          :
        ENDIF
```

GETCL@ ❷

Purpose To get a line of text from the keyboard.

Syntax

```
SUBROUTINE GETCL@(C,LC)
CHARACTER*(*) C
INTEGER*2 LC
```

Description This routine waits until the next line is typed at the keyboard, and returns it in C. LC is set to the length of the line.

Example

```
        CHARACTER*10 ANS
        INTEGER*2 LC
1       CALL COUA@('Type STOP or GO ')
        CALL GETCL@(ANS,L)
        CALL UPCASE@(ANS)
        IF(ANS.EQ.'STOP')GOTO 10
        IF(ANS.EQ.'GO')GOTO 20
        GOTO 1
        . . .
```

HIDE_CURSOR@ ❷

Purpose To hide the text cursor.

Syntax `SUBROUTINE HIDE_CURSOR@`

Description HIDE_CURSOR@ records the current shape of the cursor and removes it from the screen. A subsequent call to RESTORE_CURSOR@ will return the cursor. This routine is often useful while processing windows in which the cursor is an irrelevance.

KEY_WAITING@ ❷

Purpose To test if the keyboard buffer is empty.

Syntax `LOGICAL*2 FUNCTION KEY_WAITING@()`

Return value KEY_WAITING@ returns .TRUE. if there is a key waiting to be read from the keyboard, .FALSE. if not.

See also DOS_KEY_WAITING@.

Example

```
LOGICAL KEY_WAITING@
:
:
IF(KEY_WAITING@())THEN
  CALL GET_KEY@(KEY)
  CALL PROCESS(KEY)
ENDIF
END
```

NEWLINE@

Purpose To write a carriage return/linefeed to the screen (standard output).

Syntax
```
SUBROUTINE NEWLINE@
```

PRINT_BYTES@

Purpose To write a sequence of hexadecimal values.

Syntax
```
SUBROUTINE PRINT_BYTES@(IARR,N)
INTEGER*2 IARR(*),N
```

Description PRINT_BYTES@ writes N bytes of datum IARR in hexadecimal to standard output, separating each byte value by a space, with no terminating new line.

PRINT_BYTES_R@

Purpose To write a hexadecimal sequence in reverse order.

Syntax
```
SUBROUTINE PRINT_BYTES_R@(IARR,N)
INTEGER*2 IARR(*),N
```

Description This routine is similar to PRINT_BYTES@, but the bytes are written in reverse order.

PRINT_HEX1@

Purpose To print a 1 byte hexadecimal number (2 digits) without a new line.

Syntax
```
SUBROUTINE PRINT_HEX1@(L)
INTEGER*1 L
```

PRINT_HEX2@

Purpose To print a 2 byte hexadecimal number (4 digits) without a new line.

Syntax

```
SUBROUTINE PRINT_HEX2@(L)
INTEGER*2 L
```

PRINT_HEX4@

Purpose To print a 4 byte hexadecimal number (8 digits) without a new line.

Syntax

```
SUBROUTINE PRINT_HEX4@(L)
INTEGER*4 L
```

PRINT_I1@

Purpose To print an INTEGER*1 decimal number without a new line.

Syntax

```
SUBROUTINE PRINT_I1@(I)
INTEGER*1 I
```

PRINT_I2@

Purpose To print an INTEGER*2 decimal number without a new line.

Syntax

```
SUBROUTINE PRINT_I2@(I)
INTEGER*2 I
```

PRINT_I4@

Purpose To print an INTEGER*4 decimal number without a new line.

Syntax

```
SUBROUTINE PRINT_I4@(L)
INTEGER*4 L
```

PRINT_R4@

Purpose To print a REAL*4 value as a number without a new line.

Syntax
```
SUBROUTINE PRINT_R4@(R)
REAL*4 R
```

PRINT_R8@

Purpose To print a REAL*8 value as a number without a new line.

Syntax
```
SUBROUTINE PRINT_R8@(R)
REAL*8 R
```

READ_EDITED_LINE@ ❷

Purpose To input text from a screen position.

Syntax
```
SUBROUTINE READ_EDITED_LINE@(LINE,HP,VP,ATTRIBUTE,IC)
CHARACTER*(*) LINE
INTEGER*2 HP,VP,ATTRIBUTE,IC
```

Description READ_EDITED_LINE@ provides a user friendly text input function for the screen.

LINE contains the default string which is initially displayed on the screen. It may then be edited by using any of the standard line editing key strokes. If the cursor is not repositioned before editing the string then the input string is deleted on the screen.

(HP,VP) provide the character position of the string relative to the top left of the screen. ATTRIBUTE provides the colour of the text and its background (see the introduction to this chapter). Before editing, the colour of the string is reversed.

The subroutine yields a value IC = -1, if the Esc key was pressed to end the edit rather than Enter. Otherwise IC = 0.

RESTORE_CURSOR@ ❷

Purpose To show the text cursor.

Syntax
```
SUBROUTINE RESTORE_CURSOR@
```

Description RESTORE_CURSOR@ restores a cursor which has been hidden by a call to HIDE_CURSOR@. It should only be used after such a call.

SET_CURSOR_POS@ ❷

Purpose To set the co-ordinates of the text cursor.

Syntax
```
SUBROUTINE SET_CURSOR_POS@(IH,IV)
INTEGER*2 IH,IV
```

Description This subroutine sets the horizontal position IH and the vertical position IV of the console screen text cursor. If the screen has 80x25 character positions then the range of IH is 0..79 and the range of IV is 0..24 with (0,0) at the top left-hand corner.

See also GET_CURSOR_POS@.

SET_CURSOR_TYPE@ ❷

Purpose To set the shape of the text cursor.

Syntax
```
SUBROUTINE SET_CURSOR_TYPE@(TYPE)
INTEGER*2 TYPE
```

Description This subroutine sets the shape of the text cursor according to the value of TYPE. The cursor is rectangular and is made up of a number of scan lines (the number depending on the graphics adapter) numbered from the top which is line number zero. The high byte of TYPE is set to the starting scan line and the low byte is set to the ending scan line.

TYPE=Z'0607' is often the default setting for a colour/ graphics adapter.

SOU@

Purpose To output text with a new line, omitting any trailing blanks.

Syntax

```
SUBROUTINE SOU@(A)
CHARACTER*(*) A
```

Description SOU@ writes to the screen from the character argument A. It is identical to COU@ except that the cursor will not scan over any trailing blanks.

See also COU@, COUA@, COUP@, SOUA@.

SOUA@

Purpose To output text without a new line, omitting any trailing blanks.

Syntax

```
SUBROUTINE SOUA@(A)
CHARACTER*(*) A
```

Description SOUA@ writes to the screen from the character argument A omitting any trailing blanks. It is like SOU@ but does not output a new line.

See also COU@, COUA@, COUP@, SOU@.

Example

```
CHARACTER*20 FRED
. . .
CALL COUA@('using the fact that FRED=')
CALL SOUA@(FRED)
CALL COUA@(' enter the value of N: ')
READ *,N
. . .
```

22.

Text windows (DOS)

This chapter describes routines which implement text mode windowing. Any positional arguments (HP, VP, HS, VS) are text screen relative (i.e. character positions), with position (0,0) being the top left-hand corner of the screen. HANDLE is an identifier given a value when a window is first created and used subsequently. It is analogous to a DOS file handle. Text is written to a window (using WCOU@ or WCOUP@, not WRITE or PRINT) with a given attribute (colour) IAT as described in the introduction to chapter 21. IAT=-1 implies the use of a default attribute which is assigned when the window is created.

The following routines work equally well when the screen is in graphics mode with the exception that window shadows are not available.

The FTN77 library does not provide direct support for graphics windows, although routines such as GET_SCREEN_BLOCK@ and RESTORE_SCREEN_BLOCK@ allow users to implement their own graphics windowing system.

CONCEALW@ ❷

Purpose To move a window to the bottom of the stack.

Syntax
```
SUBROUTINE CONCEALW@(HANDLE)
INTEGER*2 HANDLE
```

Description This routine moves a text window to the bottom of the stack, making it the least visible. If only one window has been created, then CONCEALW@ will hide it.

See also WCREATE@, POPW@, MOVEW@.

KILLW@ ❷

Purpose To remove a text window.

Syntax
```
SUBROUTINE KILLW@(HANDLE)
INTEGER*2 HANDLE
```

Description KILLW@ removes a text window created by WCREATE@ from the system freeing the associated memory and disassociating the handle.

See also WCREATE@.

MOVEW@ ❷

Purpose To change the position of a window on the screen.

Syntax
```
SUBROUTINE MOVEW@(HANDLE,HP,VP)
INTEGER*2 HANDLE,HP,VP
```

Description This routine moves a text window so that its top left hand corner is at the specified position (HP,VP) relative to the top left of the screen. The routine removes the window and redraws it at the new position.

See also WCREATE@.

POPW@ ❷

Purpose To move a window to the top of the stack.

Syntax
```
SUBROUTINE POPW@(HANDLE)
INTEGER*2 HANDLE
```

Description POPW@ moves a text window to the top of the stack making it the most visible. When a window is created, it must be popped to make it visible even if it is the only one that has been created.

See also WCREATE@, CONCEALW@, MOVEW@.

SCROLL_DOWN@ and SCROLL_UP@ ❷

Purpose To scroll text in a window.

Syntax
```
SUBROUTINE SCROLL_DOWN@(HANDLE)
SUBROUTINE SCROLL_UP@(HANDLE)
INTEGER*2 HANDLE
```

Description These routines scroll the text in a given window one line at a time with re-display on each call. Text that is scrolled out of a window cannot be recovered by a call to the other function in the pair.

SET_CURSOR_POSW@ ❷

Purpose To set the cursor position for a text window.

Syntax
```
SUBROUTINE SET_CURSOR_POSW@(HANDLE,HP,VP)
INTEGER*2 HANDLE,HP,VP
```

Description This routine sets the position of the text cursor in a window to (HP,VP) relative to the top left of the window.

See also WCREATE@, WCOUP@, WCOU@.

WBORDER@ ❷

Purpose To set the border style for a text window.

Syntax
```
SUBROUTINE WBORDER@(HANDLE,BORDER)
INTEGER*2 HANDLE,BORDER
```

Description The WBORDER@ routine selects the border style BORDER for the window with handle HANDLE as follows.

BORDER	style
0	Empty spaces as border
1	Double horizontal and vertical lines (the default)
2	Single horizontal and vertical lines
3	Double horizontal and single vertical lines
4	Single horizontal and double vertical lines
5	Thick horizontal and vertical lines

See also WCREATE@, WDBORDER@.

WCLEAR@

Purpose To clear a text window

Syntax
```
SUBROUTINE WCLEAR@(HANDLE)
INTEGER*2 HANDLE
```

Description WCLEAR@ clears a window to the background colour and border supplied to WCREATE@.

WCOU@

Purpose To write text to a window.

Syntax
```
SUBROUTINE WCOU@(C,IAT,HANDLE)
CHARACTER*(*) C
INTEGER*2 IAT,HANDLE
```

Description WCOU@ writes the string C to the given window. Each call of WCOU@ begins a new line with the first call writing to the first line of the window. IAT provides the colour attributes of the text and its background (see the introduction to chapter 21).

See also SET_CURSOR_POSW@, WCOUP@.

WCOUP@ ❷

Purpose To write text to a window position.

Syntax
```
SUBROUTINE WCOUP@(C,IAT,HP,VP,HANDLE)
CHARACTER*(*) C
INTEGER*2 IAT,HANDLE
INTEGER*2 HP,VP
```

Description WCOUP@ writes a string C to the window at the position (HP,VP) relative to the top left of the window. IAT provides the colour attributes of the text and its background (see the introduction to chapter 21).

See also SET_CURSOR_POSW@, WCOU@.

WCREATE@ ❷

Purpose To create a text window.

Syntax
```
SUBROUTINE WCREATE@(HP,VP,HS,VS,IAT,HANDLE)
INTEGER*2 HANDLE,IAT
INTEGER*2 HP,VP,HS,VS
```

Description WCREATE@ yields a handle HANDLE for a text window with top left corner at (HP,VP) measured in character positions with (0,0) at the top left of the screen. HS and VS provide the width and height (in characters) of the window. The input value for the colour attribute IAT sets the border colour (which is also the default text colour) and the background colour for the window (see the introduction to chapter 21). The window must be popped (see POPW@) to make it visible.

See also KILLW@.

WDBORDER@ ❷

Purpose To set the default border style for all subsequent text windows created.

Syntax
```
SUBROUTINE WDBORDER@(BORDER)
INTEGER*2 BORDER
```

Description This routine selects the default border style for all text windows that will be created. BORDER has the same meaning as in WBORDER@.

See also WCREATE@, WBORDER@.

WDSHADOW@

Purpose To set the default shadow style for all subsequent text windows created.

Syntax
```
SUBROUTINE WDSHADOW@(SHADOW)
INTEGER*2 SHADOW
```

Description This routine selects the default shadow style for all text window that will be created. SHADOW has the same meaning as in WSHADOW@.

See also WCREATE@, WSHADOW@.

WMEMORY@

Purpose To get the memory pointer for a text window.

Syntax
```
SUBROUTINE WMEMORY@(HANDLE,PTR)
INTEGER*2 HANDLE
INTEGER*4 PTR
```

Description WMEMORY@ returns the memory pointer PTR to a window.

WREAD_EDITED_LINE@

Purpose To input text from a window position.

Syntax
```
SUBROUTINE WREAD_EDITED_LINE@(LINE,HP,VP,HANDLE,IAT,IC)
CHARACTER*(*) LINE
INTEGER*2 HP,VP,HANDLE,IAT,IC
```

WREAD_EDITED_LINE@ provides a user friendly text input routine for use with the text windowing system.

LINE contains the default string which is initially displayed on the screen. It may then be edited by using any of the standard line editing key strokes. If the

cursor is not repositioned before editing the string then the input string is deleted on the screen.

(HP, VP) provide the character position of the string relative to the top left of the window. IAT provides the colour attributes of the text and its background (see the introduction to chapter 21). Before editing, the colour of the string is reversed.

Description The routine yields a value IC = -1, if the Esc key was pressed to end the edit rather than Enter. Otherwise IC = 0.

WSHADOW@ ❷

Purpose To set the shadow style for a text window.

Syntax

```
SUBROUTINE WSHADOW@(HANDLE,SHADOW)
INTEGER*2 HANDLE,SHADOW
```

Description The WSHADOW@ routine selects the shadow style SHADOW for the window with handle HANDLE as follows. The shadow is a row and column, respectively below and to the right of the window, offset one character. The style of the shadow is formed by changing the text/background attribute of the corresponding character. The colours 7 and 0 refer to the palette register numbers for the default text and background colours which are usually white and black (see SET_PALETTE@ for details).

SHADOW	style
0	No shadow (the default)
1	Intense bit of background attribute turned off
2	Text colour = 7, background colour = 0
3	No text, solid colour = 0
4	No text, chequered colour 7 on colour 0
5	No text, solid colour = 7

See also WCREATE@, WDSHADOW@.

WTITLE@ ❷

Purpose To assign a title to a text window.

Syntax

```
SUBROUTINE WTITLE@(HANDLE,C,IAT)
CHARACTER*(*) C
INTEGER*2 IAT,HANDLE
```

Description The WTITLE@ routine assigns a title to the window with handle HANDLE. C is the string to be used as a title. If the string is wider than the window then it is cut short. IAT provides the colour of the text and its background as in WCREATE@. If IAT = -1, the default values for the text and its background are supplied (see WCREATE@).

23.

Time and date

CLOCK@

Purpose To get a time in seconds.

Syntax

```
SUBROUTINE CLOCK@(R)
REAL*4 R
```

Description This routine is usually used to time a process as shown in the example below.

Notes It should not be used to time processes under DESQview as it returns elapsed time, which is *not* CPU time, if multiple windows are in use.

See also DCLOCK@, HIGH_RES_CLOCK@, SECONDS_SINCE_1980@.

Example

```
      CALL CLOCK@(START)
      . . .
C     some calculation
      . . .
      CALL CLOCK@(FINISH)
      PRINT *,'elapsed time used = ',FINISH-START
      END
```

CONVDATE@ Ⓢ

Purpose To get the date in numeric form.

Syntax

```
SUBROUTINE CONVDATE@(SECS,IDW,IDAY,IMONTH,IYEAR)
INTEGER*4 SECS
```

```
INTEGER*2 IDW,IDAY,IMONTH,IYEAR
```

Description Converts SECS into a day of the week, IDW (0 = Sunday) and the day, month and year.

DATE@

Purpose To get the date in the form MM/DD/YY (American format).

Syntax
```
CHARACTER*8 FUNCTION DATE@()
```

See also EDATE@, FDATE@.

Example

```
CHARACTER*8 DATE@
PRINT *,'program run on ',DATE@()
```

DCLOCK@

Purpose To get a time in seconds.

Syntax
```
SUBROUTINE DCLOCK@(R)
REAL*8 R
```

Description This routine is usually used to time a process as shown in the example for CLOCK@. This routine differs from CLOCK@ only in that its argument is REAL*8. Its main purpose is for use in conjunction with /DREAL, when all variables declared REAL*4 are actually compiled as REAL*8.

See also HIGH_RES_CLOCK@.

EDATE@

Purpose To get the date in the form DD/MM/YY (European format).

Syntax
```
CHARACTER*8 FUNCTION EDATE@()
```

See also DATE@, FDATE@.

Example

```
CHARACTER*8 EDATE@
PRINT *,'program run on ',EDATE@()
```

FDATE@

Purpose To get the date in text form.

Syntax `CHARACTER*(*) FUNCTION FDATE@()`

Return value FDATE@ returns the date in the form:
Thursday February 11, 1988

See also DATE@, EDATE@.

Example

```
CHARACTER*20 FDATE@
PRINT *,'Program run on ',FDATE@()
```

HIGH_RES_CLOCK@

Purpose To obtain the CPU time accurate to 1 microsecond.

Syntax

```
REAL*8 FUNCTION HIGH_RES_CLOCK@(ALIGN)
LOGICAL*2 ALIGN
```

Description This function returns the CPU time as seconds since midnight accurate to about 1 microsecond (although the cost of the function call is approximately 100 microseconds because it must call DBOS). To achieve this precision the system clock is reprogrammed in mode 2. This could in principle affect other software, although we are not aware of any problems. The clock remains programmed in mode 2 until the system is rebooted. If the ALIGN argument is set .TRUE., this function will not return until after the next clock tick to help to obtain consistent timings. Obviously the second of a pair of calls to HIGH_RES_CLOCK@ should have ALIGN set to .FALSE.. Although the function is defined as REAL*8 it actually returns an 80-bit precision result.

See also DCLOCK@.

Example

```
REAL*8 T1,T2,HIGH_RES_CLOCK@
T1=HIGH_RES_CLOCK@(.TRUE.)
CALL SOME_PROCESS
T2=HIGH_RES_CLOCK@(.FALSE.)
PRINT *,'Time required = ',T2-T1
END
```

SECONDS_SINCE_1970@

Purpose To get the number of seconds from a fixed date.

Syntax

```
SUBROUTINE SECONDS_SINCE_1970@(DR)
REAL*8 DR
```

Description Returns the value of DR as the number of seconds that have elapsed since 12.00am on 1st January 1970.

SECONDS_SINCE_1980@

Purpose To get the number of seconds from a fixed date.

Syntax

```
SUBROUTINE SECONDS_SINCE_1980@(DR)
REAL*8 DR
```

Description This routine returns the value of DR as the number of seconds that have elapsed since 12.00am on 1st January 1980. It can be used in a similar way to CLOCK@ (or DCLOCK@). SECONDS_SINCE_1980@ should be used when making timings which straddle midnight.

SET_ALARM_CLOCK@ ❷

Purpose To set the elapsed time before an alarm.

Syntax

```
SUBROUTINE SET_ALARM_CLOCK@(TIME)
INTEGER*4 TIME
```

Description After calling this routine an alarm clock event will occur after TIME ticks. You should have used SET_TRAP@ with a code of 3 to set up an interrupt routine

to handle the interrupt.

Example

```
      EXTERNAL ALARM
      INTEGER*4 Q,TICKS
      CALL SET_TRAP@(ALARM,Q,3)
      TICKS=500
      CALL SET_ALARM_CLOCK@(TICKS)
 1    PRINT *,'TESTING'
      GOTO 1
      END
      INTERRUPT SUBROUTINE ALARM
      PRINT *,'TIME UP!'
C     Note that the rules of interrupt subroutines forbid a return from this
C     point, since we have done an I/O operation and will in all probability
C     be returning to another I/O statement. To continue we would need to use
C     subroutine JUMP@
      STOP
      END
```

TIME@

Purpose To get the time in the format HH:MM:SS.

Syntax

```
CHARACTER*8 FUNCTION TIME@()
```

Example

```
      CHARACTER*8 TIME@
      PRINT *,'program starting execution at ',TIME@()
```

TODATE@ ❺

Purpose To convert the time given to a date for the form MM/DD/YY (American format).

Syntax

```
CHARACTER*(*) FUNCTION TODATE@(SECS)
INTEGER*4 SECS
```

TOEDATE@ ❺

Purpose To convert the time given to a date in the format DD/MM/YY (European format).

Syntax
```
CHARACTER*(*) FUNCTION TOEDATE@(SECS)
INTEGER*4 SECS
```

TOFDATE@ ❺

Purpose To get the date in text form.

Syntax
```
CHARACTER*(*) FUNCTION TOFDATE@(SECS)
```

Return value TOFDATE@ returns the date in textual format based on time given in the form: “Friday January 29, 1993”.

TOTIME@ ❺

Purpose To return the time in the form ‘‘HH:MM:SS’’.

Syntax
```
CHARACTER*(*) FUNCTION TOTIME@(SECS)
INTEGER*4 SECS
```

Return value Returns the time corresponding to SECS.

Index

F

G

H

I

J

T

U

V

W

Y