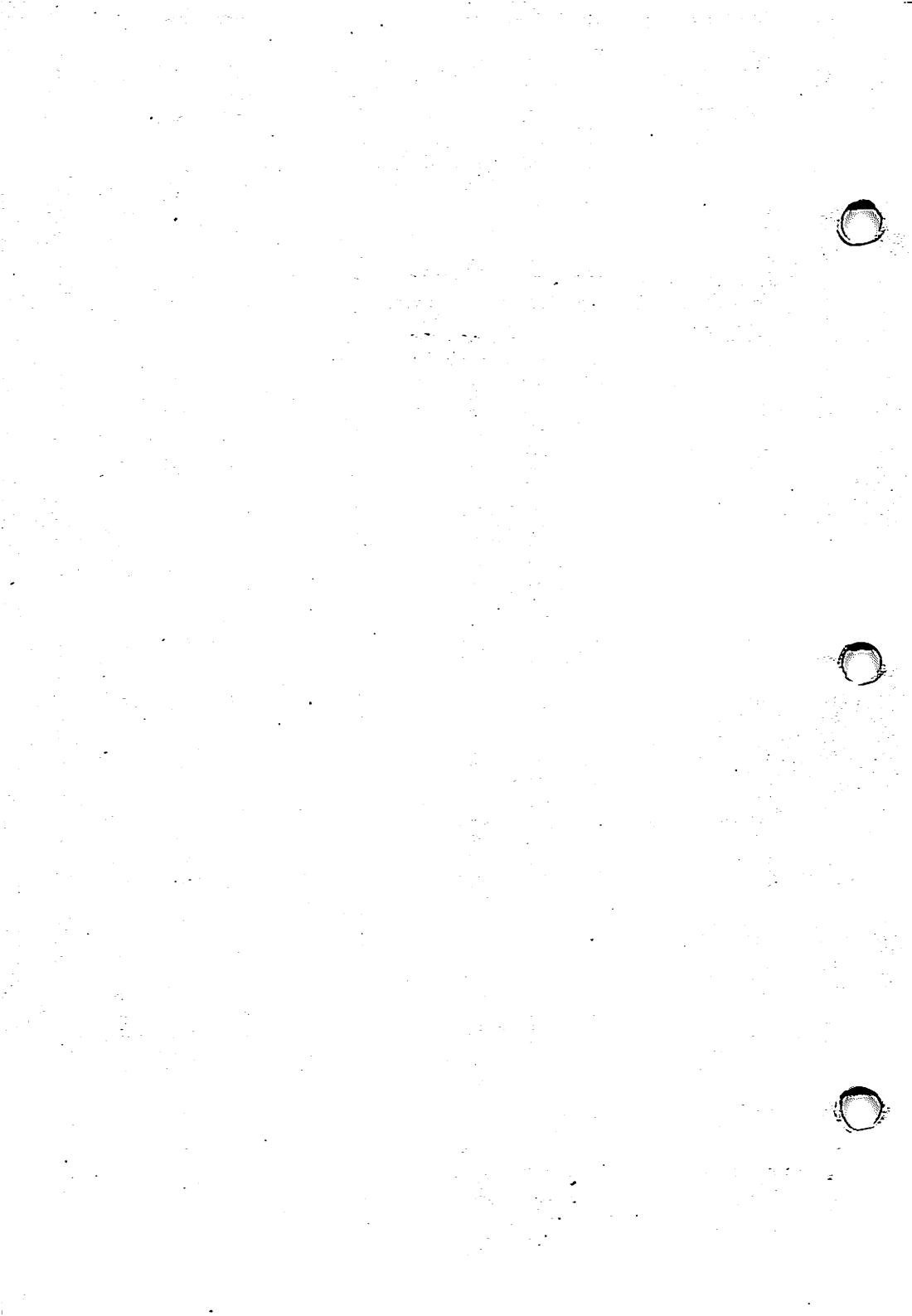


---

## **Communicator Systems Manual**

---

**Part No 0452,009  
Issue 2  
20 July 1987**



Copyright Acorn Computers Limited 1987

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written permission of Acorn Computers Limited.

Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

In case of difficulty please contact your supplier. Every step is taken to ensure that the quality of software and documentation is as high as possible. However, it should be noted that software cannot be written to be completely free of errors. To help Acorn rectify future versions, suspected deficiencies in software and documentation should be notified in writing, using the Fault Report Form supplied, to the following address:

Acorn Computers Limited,  
Fulbourn Road,  
Cherry Hinton,  
Cambridge CB1 4JN

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Published by Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN

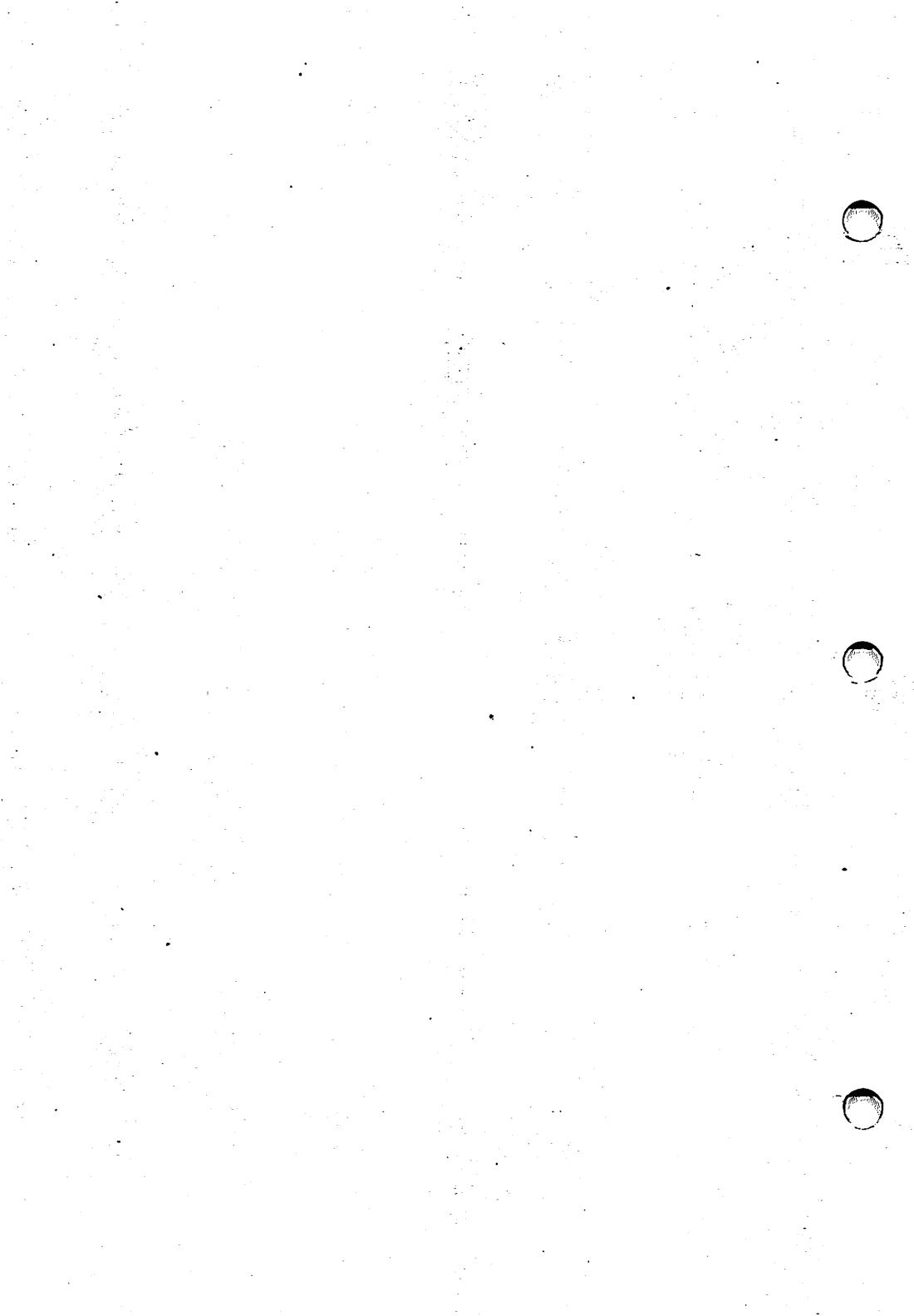


# Contents

<b>1. Introduction</b>	<b>1.1</b>
1.1 Communicator terms and concepts	1.2
1.2 Programmer's model	1.4
1.2.1 System software levels	1.4
1.2.2 System software categories	1.5
1.2.3 Communicator system hardware	1.6
1.2.4 Address decoding and the Communicator memory map	1.8
1.3 Obtaining routine addresses	1.10
1.4 General protocols	1.11
1.5 Test harness for examples in BASIC assembler	1.12
1.6 General calling method	1.13
<b>2. Modules</b>	<b>2.1</b>
2.1 Introduction	2.1
2.2 Module format	2.2
2.3 Module scan	2.3
2.4 Module-type flags	2.3
2.5 Example module	2.4
2.6 Calling a module	2.5
2.7 Module functions	2.6
<b>3. Coroutines</b>	<b>3.1</b>
3.1 Introduction	3.1
3.2 An example demonstrating the use of coroutine functions	3.4
3.3 Coroutine functions	3.8
<b>4. Memory management</b>	<b>4.1</b>
4.1 Memory management COP calls	4.1
<b>5. Task management and the menu program</b>	<b>5.1</b>
5.1 Task pre-emption	5.1
5.2 Event acknowledgement	5.1
5.3 Coroutine context	5.1
5.4 The menu module	5.2
5.4.1 Building the task table	5.2
5.4.2 Task management	5.2
5.5 An example using the use event and context functions	5.3
5.6 BRK handling	5.7
5.7 Task management and context associated functions	5.8
<b>6. Device drivers</b>	<b>6.1</b>
6.1 Introduction to device drivers	6.1
6.2 Simple device control from BASIC	6.2
6.2.1 Opening a device driver	6.2
6.2.2 Sending and receiving data	6.2
6.2.3 Sending commands to a device driver	6.2
6.2.4 Obtaining status information from a device driver	6.3
6.2.5 Closing a device driver	6.3
6.3 Default options and the Configure program	6.4
6.4 Device driver "filenames"	6.5
6.5 Device control from assembler programs	6.6
6.6 Low-level device driver reason codes	6.15
6.6.1 Basic device driver reason codes	6.16
6.6.2 Extended device driver reason codes	6.16

<b>6.7 The clock device driver</b>	<b>6.42</b>
6.7.1 Introduction	6.42
6.7.2 Description	6.42
6.7.3 Clock driver functional interface	6.43
6.7.4 An Example of accessing the clock driver	6.44
6.7.5 Clock driver functions	6.45
<b>6.8 ECONET:</b>	<b>6.53</b>
<b>6.9 RAM:</b>	<b>6.75</b>
6.9.1 32K CMOS non-volatile memory	6.75
6.9.2 32K dynamic RAM	6.75
<b>6.10 CARD: the memory card filing system</b>	<b>6.76</b>
6.10.1 Introduction	6.76
6.10.2 Description	6.76
6.10.3 Card filing system commands	6.77
6.10.4 Card filing system functional interface	6.78
<b>6.11 Autoboot facilities</b>	<b>6.99</b>
6.11.1 Boot files at logon	6.99
6.11.2 Autostart at power-on	6.99
<b>6.12 PRINTER:</b>	<b>6.100</b>
6.12.1 Introduction	6.100
6.12.2 Device driver commands	6.101
<b>6.13 CENTRONICS:</b>	<b>6.102</b>
6.13.1 Introduction	6.102
6.13.2 Device driver commands	6.102
<b>6.14 NETPRINT:</b>	<b>6.103</b>
6.14.1 Introduction	6.103
6.14.2 Software	6.103
<b>6.15 TEXT: the front-end printer driver</b>	<b>6.105</b>
6.15.1 Introduction	6.105
6.15.2 Description	6.105
6.15.3 TEXT: reason codes	6.105
6.15.4 The TEXT: device driver functional interface	6.106
<b>6.16 GRAPHICS:</b>	<b>6.113</b>
6.16.1 Introduction	6.113
6.16.2 Description	6.113
6.16.3 Printer types	6.113
<b>6.17 Introduction to the terminal session software</b>	<b>6.114</b>
6.17.1 Introduction	6.114
6.17.2 Description	6.114
6.17.3 The Phone module	6.114
6.17.4 TShell - the terminal shell	6.115
6.17.5 Keypage	6.115
6.17.6 The emulation modules	6.116
6.17.7 The low-level drivers	6.116
<b>6.18 KEYPAGE: the terminal kernel</b>	<b>6.117</b>
6.18.1 Introduction	6.117
6.18.2 Description	6.117
6.18.3 Keypage operation	6.119
6.18.4 Keypage termination return codes	6.121
6.18.5 Application Escape	6.122
6.18.6 Keypage functions	6.125
<b>6.19 Terminal Emulation Modules</b>	<b>6.141</b>
6.19.1 Purpose	6.141

6.19.2 Usage	6.142
6.19.3 Software Interface	6.143
6.19.4 Software Call Details	6.144
6.19.5 Teletype	6.148
6.19.6 BBC	6.149
6.19.7 VT100	6.150
6.19.8 Videotex	6.151
6.19.9 Keyboard mapping table	6.152
6.20 RS423:	6.153
6.20.1 Introduction	6.153
6.20.2 Hardware	6.154
6.20.3 Device driver commands	6.155
6.20.4 Query commands	6.157
6.20.5 Transmission protocols	6.158
6.21 MODEM:	6.159
6.21.1 Modem device driver commands	6.160
6.22 ECPMODEM:	6.165
6.22.1 Introduction	6.165
6.22.2 General description	6.165
6.22.3 Character mode operation	6.165
6.22.4 Block mode operation	6.165
6.22.5 The driver interface	6.166
6.22.6 Device driver commands	6.167
7. Files	7.1
7.1 Introduction	7.1
7.1.1 Description	7.2
7.1.2 MOS star (*) file commands	7.3
7.1.3 BASIC file commands	7.4
7.1.4 Filing system COP calls	7.5
8. Interrupt handling	8.1
9. AR arithmetic package module	9.1
10. General calls	10.1
11. BBC similar calls	11.1



# 1. Introduction

This manual describes the components that make up the low-level system software resident in the Communicator system ROM. These include Operating System (OS) routines, device drivers, menu handling functions and filing systems. These routines enable the writer of applications programs to interface to the system in a clean, well defined way. It is recommended that they be used in preference to accessing the hardware or system memory locations directly.

It is assumed that the reader is familiar with the architecture of the 65SC816 processor. Information about this is given in the *Communicator Basic Reference Manual*.

## 1.1 Communicator terms and concepts

This section defines some terms and concepts that the reader will encounter in the Systems Manual.

**Bank**

The 68SC816s 16 Mbyte address space is divided into 256 banks of 64 Kbytes.

**Cooperative Multi-tasking**

Communicator is capable of supporting a number of independent active tasks, only one of which is being run at any one time. Unlike most multi-tasking systems, where swapping between different tasks occurs under the control of a scheduler at regular, short intervals, on Communicator movement between tasks is requested by the user pressing pre-empt keys. The task becomes suspended if it is requesting keyboard input or allowing pre-emption. To operate correctly under this mechanism, a task must inherently be well behaved, effectively allowing itself to be stopped.

**Co-routine**

A Co-routine is a loosely coupled process which forms part of a hierarchy of similar co-routines. A co-routine has its own stack and associated code and it may be assigned its own direct page. A co-routine may or may not be a task. A co-routine is analogous to a process in normal operating systems terminology.

See the section on co-routines for more detail.

**Direct Page**

This refers to a 256 byte block of memory assigned within bank zero to be accessed via the 65C816 D-register.

See the section on memory management for more detail.

See also the 65C816 Programmer's Guide.

**Emulation**

Emulation refers to the ability of Communicator to respond correctly to the screen control codes for a variety of devices. These include a VT100 terminal, Videotex terminal, teletype and standard BBC computer. This allows applications that expect one of these types of screen display to use the Communicator display unaltered.

**Handle**

A Handle is an identification value assigned to an object such as a memory pool or a file when allocated or opened by the MOS. The handle is supplied by the application whenever the object needs to be identified to the MOS, (e.g when releasing a memory pool).

**Module**

All software within Communicator is held in the form of modules. A Module has a unique name which is used within the system to access the contents of the module. A module is a sequence of bytes stored in ROM or RAM, starting with a standard Module Header. A module usually contains a program, (either BASIC or machine-code).

See the section on modules for more detail.

**Pre-emption**

When a user presses the HELP or STOP key a pre-empt occurs. The task which is running becomes suspended after it reads the keyboard input corresponding to the pre-empt key. From the point of view of the task this simply means that reading the keyboard takes a variable, possibly infinite length of time. From the point of view of the user the running task is stopped, control goes back to the main menu and another task may be started. Later the suspended task may be restarted, at which point it continues from the state in which it was left.

**Page**

Each memory Bank is divided into 256 pages of 256 bytes each.

**Pool**

A Pool is an area of memory allocated by the MOS memory management system. A pool may have sub-pools allocated from within it. There are five pools initially defined:

HDMMW Entire memory map pool  
HDMMM Entire memory pool  
HDMMT Task memory  
HDMMC CMOS RAM pool  
HDMMV Screen memory pool

Most applications software will only be concerned with the Task Memory pool.

**Shadowing**

Because of the way in which addresses are decoded by hardware within Communicator, a given piece of memory may appear within the memory map at a number of different addresses. There is no actual difference between accessing memory at one value or another, the same bytes are being changed or executed. This effect is used in two fashions within Communicator. Firstly, at power-on, when the system ROM is mapped onto the first bank of memory to provide bootstrap code. Secondly, when in normal operation, the first bank of memory maps onto the first empty bank above the top of system RAM. This allows the memory to be treated as a single contiguous address space. On a 512K Communicator this means that bank zero corresponds with bank eight.

**Task**

A task is a program that is runnable from the Communicator top level menu. A task is started as a single co-routine which may spawn more related +co-routines.

## **1.2 Programmer's model**

Communicator is composed of system hardware and system software. The system software can reside permanently in ROM or non-volatile RAM, or can be down-loaded into RAM from the Econet (Transient Software).

### **1.2.1 System software levels**

Communicator software can be viewed from three levels:

The Module level

The Co-routine level

The Task level

At the lowest level is the Module. All Communicator software exists in module format. A module is sequence of bytes stored in ROM or RAM starting with a standard Module Header. The module header contains information about the contents of the module, including the type of module (e.g. BASIC), the address of the start of the module code, the name of the module, etc. The module is terminated by a CRC for integrity checking. Modules provide the communicator MOS with the basic information it requires to locate and invoke different items of software within the system.

The co-routine level refers to processes within the system. A Co-routine is a loosely coupled process which forms part of a hierarchy of similar co-routines. A co-routine has its own stack and associated code and it may be assigned its own direct page. Co-routines may communicate with other co-routines within the system via the MOS. Co-routines may be created and deleted by other co-routines. Co-routines may relinquish ownership of the processor to other co-routines.

The top level of the system is the Task Level. Tasks are software entities that are visible to the user from the Menu level. A task is always a Co-routine. A co-routine becomes a task when it is installed into a menu. Most tasks can be suspended by hitting a Pre-empt Key.

### 1.2.2 System software categories

The system software can be divided into four categories:

1. Software that provides an operating environment and general access to Communicator hardware for the user and applications software.
2. Filing system software.
3. Software that provides access to specific devices within or attached to Communicator.
4. Applications Software.

Together these components constitute a complete operating environment, allowing users access to the full power of Communicator hardware, whilst isolating them from its internal complexities.

#### *Communicator MOS*

The Communicator MOS, or Machine Operating System, is the core of the system software. It provides control of and access to Communicator system hardware. All other system and applications software relies on the MOS for access to the hardware and for communication with other software.

#### *The filing systems*

The Communicator Filing Systems provide facilities for the storage and retrieval of information, (programs or data), on both internal and external data storage devices. There are two filing systems on Communicator - the RAM Filing System and the Econet Filing System.

The RAM Filing System provides facilities for storing programs and data within System RAM. The user can select from the normal dynamic RAM and the special Non-volatile RAM, (Battery-backed CMOS RAM). Data stored in the normal system dynamic RAM is lost after Communicator is powered down or subjected to a hardware reset, whilst data stored in the non-volatile RAM is retained.

The Econet Filing System, as its name suggests, provides facilities to access and manipulate files on the Econet.

#### *Device drivers*

Access to specific hardware devices within or attached to Communicator is normally via a piece of software called a device driver.

The purpose of a device driver is to provide a level of abstraction between the application requiring the services of a device and the device itself. This means that the application needs no knowledge of the hardware, relying on a set of general routines provided by the device driver to implement specific actions on the associated device.

Applications software should always access devices via device drivers.

Device drivers provide a set of functions conforming to a standard Communicator device driver interface as detailed later in this manual.

A number of device drivers are supplied as part of the System Software, e.g. the VDU device driver. It is also possible for Software Developers to create their own.

#### *Applications software*

Applications Software consists of programs resident in or down-loaded to Communicator which are not part of the MOS, device drivers or Filing Systems. Applications software includes languages such as BASIC, editors such as VIEW, or more general software such as spread-sheets or calculators.

Communicator is supplied with a great deal of built-in applications software and with the aid of the OEM development package, users can easily develop their own.

### **1.2.3 Communicator system hardware**

This document provides a only brief description of Communicator hardware. For more detail see the Communicator Service Manual.

#### ***The microprocessor***

The Acorn Communicator is based around the 6SC816 16-bit microprocessor, an enhanced version of the 8-bit 6502 device, featuring an 8-bit external data bus with a 16-bit internal data path. The address bus is 24 bits wide, giving an address space of 16 Mbytes. Processor speed is normally 2 MHz, dropping to 1 MHz when accessing slow devices or video RAM.

#### ***System RAM***

Communicator is supplied in two versions - one with 512 kbyte of system RAM and the other with 1 Mbyte. The RAM can be accessed at full processor speed, (i.e 2 MHz).

#### ***Video RAM***

Video RAM is accessed through the video ULA chip. There is a total of 32 Kbytes of RAM, up to 20 Kbytes of which is used to hold the screen image. The maximum rate that the processor can access the video RAM is 1 MHz.

#### ***Non-Volatile RAM***

There are 32 Kbytes of Non-Volatile RAM provided in Communicator. This RAM is fabricated in CMOS and is battery backed by a 3.6V Nicad battery.

#### ***Read Only Memory***

Communicator is supplied with up to 512 Kbytes of internal ROM containing the systems software. The ROM is normally accessed at 2 MHz although this can be link-selected to 1 MHz if required. Internal ROMS can be either of type 27512 or 27101. Decoding circuitry exists to access up to 3.5 Mbytes of external ROM on the expansion bus.

#### ***The video ULA***

The video ULA is a CMOS fabricated device providing many of the Communicator hardware functions. The video ULA provides access to the Video RAM, refreshes the display from video RAM, provides a keyboard interface, provides sound generation and supplies the processor clock.

Two sets of video output are provided - TTL level RGB signals and a monochrome composite video signal.

#### ***The MODEM circuit***

Communicator provides circuitry for pulse and tone dialling, along with an on-board Modem for direct connection to the telephone system. The Modem circuitry is based around a 7910 FSK device.

#### ***The asynchronous serial port***

Communicator is fitted with an RS423 port driven by a SCN2641 ACIA. Signals provided are GND, TXD, RXD, CTS and RTS.

The ACIA interfaces with the processor bus at 1 MHz. The baud rate for the receive and transmit line is linked, so split baud rate operation is not possible.

#### ***The Econet port***

Communicator is supplied with an on-board Acorn Econet interface. The circuitry is based around the 68B54 high-speed data-link controller chip. Data rates of up to 200 Kbps are supported.

#### ***The printer port***

Communicator is provided with a CENTRONICS type parallel printer interface. This is driven by half of a 6522 VIA. The interface provides 8 data lines, GND, STROBE and ACKNOWLEDGE.

***The real-time clock***

Communicator is provided with a real-time battery-backed clock. This device records hours, minutes and days with a year count being maintained by system software in the CMOS battery-backed RAM.

The clock is accessed via the IIC bus via system software.

***The Communicator keyboard***

The keyboard is a 90 key, full travel QWERTY device. In addition to the full ASCII character set there are ten user-definable keys, a separate numeric keypad, cursor control keys and task control keys. LEDs on the keyboard give an indication of CAPSLOCK, SHIFTLOCK and telephone LINE IN USE.

***The IIC bus***

The IIC Bus, (Inter-integrated Circuit Bus), is used to connect processors and peripheral ICs with a minimum of wires. The processor is the bus master. Slave devices on the bus include the DTMF dialler and the real-time clock.

***The external expansion bus***

Communicator provides an externally accessible edge connector which allows extra ROM, RAM and peripherals to be attached. As well as the standard 65SC816 bus control lines, there are control lines for the IIC bus. For more details see the Communicator Service Manual.

## *Chapter 1*

### **1.2.4 Address decoding and the Communicator memory map**

The 68SC816's 16 Mbyte address space is divided into 256 "Banks" of 64 Kbytes. Each bank is divided into 256 pages of 256 bytes each.

Chapter 0 of Communicator memory is unusual in that at different times it may address ROM or RAM. This comes about due to the processor requirement that vectors must be in the top page of bank &00 and the processor limitation that the stack must reside in bank &00. This produces a conflict. The stack is maintained in RAM, but vectors must be held in ROM to cope with a RESET or power-down. To resolve the conflict, ROM is located at bank &00 when Communicator is RESET and RAM is there for the rest of the time. After a RESET has been processed, the ROM at Bank &00 is swapped out to bank &FE and the Shadow RAM bank swapped in. The vectors are then copied down into the bank &00 shadow RAM from the ROM at bank &FE. This operation is executed with interrupts disabled, as interrupt actions would be undefined if they were processed before the vectors had been copied.

The following memory map describes the layout of Communicator memory after a RESET has been processed.

**Memory Map For a 512 Kbyte System**

Bank	Contents
600-607	System RAM
608-60F	Copy of System RAM
620-63F	External expansion RAM
640	MODEM
641	Econet ID
642	6522 VIA (printer etc)
643	2641 ACIA (RS423)
644	CH00SWITCH
645	Video ULA
646	CMOS RAM
647	68854 (Econet)
648-67F	Reserved
680-6BF	External expansion IO
6C0-6F7	External expansion ROM
6F8-6F9	Empty
6FA-6FB	ROM bank 4
6FC	ROM bank 3
6FD	ROM bank 2
6FE	ROM bank 0
6FF	ROM bank 1

**Memory Map For a 1 Mbyte System**

Bank	Contents
600-60F	System RAM
610-61F	Copy of System RAM
620-63F	External expansion RAM
640	MODEM
641	Econet ID
642	6522 VIA (printer etc)
643	2641 ACIA (RS423)
644	CH00SWITCH
645	Video ULA
646	CMOS RAM
647	68854 (Econet)
648-67F	Reserved
680-6BF	External expansion IO
6C0-6F7	External expansion ROM
6F8-6F9	Empty
6FA-6FB	ROM bank 4
6FC	ROM bank 3
6FD	ROM bank 2
6FE	ROM bank 0
6FF	ROM bank 1

### **1.3 Obtaining routine addresses**

All references to OS routines are performed symbolically. That is, no assumptions about the absolute locations of routines and data structures should be made. To aid the programmer, both assemblers used in the system (the BASIC assembler and MASM) provide symbols standing for all system addresses.

In BASIC, the special integer variables starting with @ are used, for example @MM%, @CO%. In MASM, header files are provided which define all of the symbols required. These all begin with the £ character, eg £MM, £CO.

In addition to the kernel of routines with two-letter names, such as MM and CO mentioned above, there are several modules in the system which perform tasks such as dealing with the keyboard, the screen, the MODEM etc. It is necessary to call modules through the COP mechanism.

Although not strictly part of the MOS, the various modules and device drivers are described in this Guide.

## 1.4 General protocols

Although certain routines give special meaning to some of the registers, some generalisations about register usage may be made. These are:

BHA This is used to pass three-byte addresses.

B is the data bank register.

H is the high byte of the accumulator.

A is the low byte of the accumulator.

Splitting the 16-bit accumulator into H and A is Acorn terminology and will not be found in the 816 data sheet.

For three-byte addresses, the high byte is passed in B, the middle byte in H and the low byte in A.

X This is used to pass 'reason codes' to routines. Reason codes are always multiples of 2, and indicate which of a number of possible tasks the routine must perform.

Y Resource handles and line offsets are passed in Y. Resource handles are allocated by create-type routines (eg create coroutine) and are used in subsequent transactions with routines in that interface. Line offsets are used in the error handling routines. Resource handles are 16-bits. Consequently, calls which allocate, use or deallocate handles must be called in 16-bit XY mode.

C This flag is cleared on exit from a routine if it succeeded and set if it failed. This is useful for error detection immediately after a routine returns.

Only registers mentioned in the descriptions of routines should be relied upon for meaningful contents on return from a call. If the register is not mentioned it should be assumed to be corrupted by the call, even if experiments tend to suggest otherwise in the version of the operating system you are using. By assuming only what is documented in this guide, you will ensure that your software remains compatible with other releases of the OS.

## 1.5 Test harness for examples in BASIC assembler

There are examples in this manual written in BASIC assembler. These examples have line numbers which allow them to be run in the test harness given below. To make an example work requires that it be placed within this test harness.

```
1 A%=0 :REM initialise HA
2 X%=0 :REM initialise X
3 Y%=0 :REM initialise Y
4 C%=0 :REM initialise carry (bit 0 of P)
5 B%=0 :REM initialise B
6 D%=0 :REM initialise D
9 :
10 : REM BASIC variable initialisations here
99 :
100 DIM code% FNAlign(99) :REM allocate 100 bytes for code
110 FOR pass%=0 TO 3 STEP 3 :REM 2 passes through assembler
120 P% =code% :REM set location counter to start address of code
130 [OPT pass%
140 .begin%
150 :
160 \ assembly language program here
990 :
1000 RTL
1010 ]
1020 NEXT pass%
1030 CALL begin%+@USEB%+@USED% :REM call machine code routine using B% and D% contents
1040 :
1050 PRINT TAB(26);"NVMXDIZC"
1060 PRINT "Processor status register ";
1070 FOR power% =7 TO 0 STEP -1
1090 IF 2^power% AND @P% THEN PRINT "1"; ELSE PRINT "0";
1100 NEXT power%
1110 PRINT TAB(15);"Y register &";RIGHT$("000"+STR$-@Y%,4)
1120 PRINT TAB(15);"X register &";RIGHT$("000"+STR$-@X%,4)
1130 PRINT TAB(22);"BHA &";RIGHT$("00000"+STR$-@BHA%,6)
1140 END
2000 :
2010 DEFFNAlign(N%) :REM ensure allocated bytes do not cross bank boundary
2020 LOCAL P%
2030 DIM P% -1
2040 IF ((P% EOR (P%+N%+1))AND &FF0000) THEN DIM P% &FFFF-(P%AND&FFFF)
2050 =N%
```

An example may consist of assembly language instructions only, in which case the lines of the example should be typed in the space from line 170 to line 980. If the example also contains some BASIC variable initialisations then these should be typed in the space from line 20 to line 90.

The harness is a BASIC program, and when run will assemble the example program in 8-bit mode, execute the object code, and give a print out of the registers on exit from the machine code routine. All examples assume the processor to start in 8 bit accumulator and index register mode. If word mode is required then the instructions WRD &30 and RSPW &30 must be used.

Note: all O/S routines normally execute in WORD mode.

To run the above program type RUN <RETURN>

## 1.6 General calling method

This section outlines the general method for calling an OS routine. The example creates a coroutine, written in BASIC assembler. This example can be used within the assembler harness given on the first page of section 4.

```

20 costksz% = &200 :REM Initialise required stack size
30 codps% = &100 :REM Initialise required direct page size
170 RSP# &30          \ Make sure processor is in word mode.
180 WRD &30          \ Tell assembler that processor is in word mode.
190 PHK               \ Coroutine entry is in my program bank.
200 PLB
210 BNK P%/&10000
220 LDA# coentry% AND &FFFF
230 PEA# costksz%
240 PLD
250 LDX# @COCRE%
260 JSRL @CO%
270 BCS coerr%
280 :
290 COP
300 EQUB @OPADP%
310 EQUW codps%
320 BCS coerr%
330 :
340 TAD
350 LDX# @COENV%
360 JSRL @CO%
370 JSRL @CCO%
380 RTL
390 :
400 Jcoentry%
410 LDA# ASC("/")
410 COP
420 EQUB @OPWRC%
430 JSRL @CWT%
440 :
450 .coerr%
460 BRK
470 EQUB &00
480 EQUUS "Create failed"
490 EQUB &00

```

\ Call the direct page allocator.  
 \ With DP size as argument.  
 \ Something wrong.  
 \ Execute COP call.  
 \ Call the coroutine's environment.  
 \ Call the coroutine.  
 \  
 \ This is the entry point of the  
 \ coroutine we just created.  
 \ Perform arbitrary task  
 \ such as  
 \ printing "/"  
 \ Do a COWAIT to return to parent at 380.  
 \ Error message for failed COCRE.  
 \  
 \

Note that JSRL is always used to call OS routines, which always use RTL to return. Note also that in addition to the JSRL-type OS calls, the 65SC816 COP instruction is also used in most circumstances as a "shorthand" method. The general form of a COP call is:

COP \ The coprocessor instruction.  
EQUB @OPOSB% \ The COP signature.

Following the single-byte COP instruction is a signature (similar to a reason code) byte in the range &00-&7F. This indicates exactly which service is required. In the example above it is the BBC compatible OSBYTE routine. Some COPs also have further arguments in-line, eg the @OPADP% used in the coroutine example. (This takes a two-byte argument giving the size of direct page required.)



## **2. Modules**

### **2.1 Introduction**

A module is a sequence of bytes stored in ROM or RAM. Information at the start of each module is stored in a standard header format. The header indicates, amongst other things, the module type and module length.

The module type depends on the contents of the module, for example a language (such as BASIC), or a filing system (such as the network filing system).

A module must start on a page boundary and must not cross a bank boundary.

A module is an independent software element. There is no direct access between modules. Communication between modules is carried out via the MOS.

All elements of Communicator software are made up of modules.

## 2.2 Module format

The format of a module is as follows.

Offset	Contents
0	BRL to start of code (&82)
1	Offset of BRL instruction (low byte)
2	Offset of BRL instruction (high byte) The three bytes of the BRL instruction consist of the BRL opcode (&82), and a 2 byte offset which will point to the start of the module proper, in this case to yy. The actual 2 byte offset will be (yy-3).
3	Module length (low byte)
4	Module length (middle byte)
5	Module length (high byte) The 3 byte module length is the total number of bytes from the BRL opcode to the byte before the low byte of the CRC. This means that the module length is an offset from the start of the module header to the first byte of the CRC.
6	version number (low byte)
7	version number (high byte) This will be stored in BCD to give the four digits of the version number, Vmm.nn. Modules should be written to return the version number in this form, eg V01.50.
8	module-type flag
9	module-type flag
10	module-type flag
11	module-type flag These 4 bytes contain flags which indicate the type of the module code. For example, there are flags to indicate a device driver, or a filing system; flags to indicate whether the code is position independent or bank independent.
12	These two bytes are reserved for future expansion.
14	Zero-terminated module name The name of the module is a sequence of alphabetic ASCII characters. The dot character "." may also be used. Optionally, a sequence of associated commands used with the module may be appended to the module name string, separated by "/" characters. If commands are included then help information corresponding to each command must be included at offset xx.
xx	Help information. If commands were included in the name string then each command must have a corresponding help string included here, separated by "/" characters and terminated by zero. The strings can be null (//).
	End of header
yy	Start of code    The 2 bytes at offset 1 and 2 are an offset to this byte. This is the first byte of the code which constitutes the module.
	Body of module
length	CRC (low byte)    The 3-byte module length is an offset to this byte. CRC (high byte)
	End of module

Note that the programmer should not attempt to extract information from the module header directly. There is an operating system call to do this, see OPRMI (read module info).

## 2.3 Module scan

At power-on, and after a \*reset command, the operating system searches through memory to find legal module headers. Whenever one is found, the name of that module is added to a list of current modules.

If the operating system finds more than one module with the same name, module suppression takes place. This means that only one of the modules will be added to the current module list, the others being ignored. The module selected will be that with the highest version number. In the case where there is more than one module with the same version number, a module in RAM will take precedence over one in ROM and where two or more are in RAM the one at the highest address wins.

If the operating system fails to detect a module, it could be for one or more of the following reasons.

The module header doesn't start on a page boundary.

The first byte isn't &82 (BRL opcode).

The next two bytes don't point to a sensible address within the module.

The name string is not ASCII.

The length doesn't point to the CRC, or the CRC is incorrect.

## 2.4 Module-type flags

These flags give information about the type of code in the module. The current types are:

MHCPOS Position independent code

MHCBNI Bank independent code

MHCFI File system interface support

MHCDEV Device driver interface support

MHCBA BASIC program

MHCLEN Module may appear in MENU task list

MHCVPD VIEW printer driver

The 4 bytes containing the module-type flags are used by the operating system, and may also be accessed by other programs which use the module. For example, the operating system will examine the flags to see if the code is position independent. If it is then the module can be loaded anywhere into RAM.

## 2.5 Example module

The following is an example module.

```
.moduleheaderstart%
BRL code%
EQUUL mosend%-moduleheaderstart%
EQUW &0150
EQUB @MHCFS%+@MHCBNK%
EQUUL &000000
:
EQUW &0000
EQUS "MOS/FX/EXEC"
EQUB &00
EQUS "/a,x,y/filename"

:
:
:

.code%
:
:
:
mosend%
(EQUB &00)
:
:
:
\ Entry to executable code at label code%
\ 3 byte length
\ Version number in BCD (01.50)
\ Module-type flags
\ Module-type flags
\ Name string starts here
\ 2-byte reserved field
\ Module name and optional commands
\ Zero terminator for name string
\ Help information
\ Note that help info for the module
\ name itself is null

\ Start of executable code

\ End of code
\ If necessary, pad module image to nearest even length
\ with zero byte
\ 2 bytes of checksum here
\ Checksum is word-sum with end-around-carry
\ A word is 2-bytes, each word starting at an even offset
```

## **2.6 Calling a module**

There are 2 methods of calling a module.

The first method uses the Call Module routine OPCMD.

The second method uses the Reference Module routine OPRFR.

OPCMD is a slow method when used repeatedly to call a module, OPRFR being faster, but OPCMD is easy to code and is efficient for calling a module once only.

## **2.7 Module functions**

The following module functions are available:

<b>OPCMD</b>	Call module
<b>OPRFR</b>	Reference module
<b>OPURF</b>	Unreference module
<b>OPRMI</b>	Read module info
<b>OPFMA</b>	Find module from given address
<b>OPWRM</b>	Write module name
<b>OPSLM</b>	Scan list of modules
<b>OPFPO</b>	Find pool owner
<b>OPNLU</b>	Name lookup
<b>OPAM</b>	Add module

# OPCMD call module

**Action:** Call the module whose name, terminated by zero, follows immediately.

**On entry:** Any values required by the module to be passed in the registers.  
Inline string which is the module name terminated by zero (&00).

**On exit:** Carry bit is set and X = 0 if the module was not found.

**To call:**

```
COP
EQUB @OPCMD%
EQU$ modulename$
EQUB &00      from BASIC
OPSYS $OPCMD
= "modulename",0 from MASM
```

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPCMD%	\ Call the module whose name follows.
190 EQU\$ "BASIC"	\ Inline string is name of module to be called.
200 EQUB &00	\ Terminated by zero.

In this case the BASIC module will be called by the OPCMD routine. OPCMD calls the module whose name immediately follows the COP signature. The name must be terminated by zero (&00). If the module is not found then the carry bit is set, X = 0, and control is returned to the user's code. However, if the module code is executed successfully the carry bit may still be set on return to the user's code, so always check X. If carry is set and X = 0 then an error occurred.

The second method uses the Reference Module routine OPRFR.

This method is very much faster than OPCMD if the module is called more than once.

## OPRFR reference module

- Action:** Set up entry and return addresses in an 8 byte block in direct page for the module whose name is pointed to by BHA.
- On entry:** BHA points to the module name which is terminated by zero (&00).  
X points to the start of the 8 byte block in direct page which will be initialised by the OPRFR routine.  
Y = 0
- On exit:** C = 0 means that the module was found and the block initialised.  
C = 1 means that the module was not found.  
DB preserved.
- To call:** COP  
EQUB @OPRFR% from BASIC  
OPSYS £OPRFR from MASM
- Example:** See example for OPURF.

# OPURF unreference module

**Action:** Let the operating system know you no longer require access to a module.

**On entry:** X points to the start of the 8 byte block in direct page which was initialised when the module was referenced.

**On exit:** C = 0 means that the module was found and unreferenced.  
C = 1 means that the value in X was invalid.  
DBHA preserved

**To call:**  
COP  
EQUB @OPURF% from BASIC  
OPSYS £OPURF from MASM

**Example:**

```

170 RSP# &30
175 WRD &30
180 COP
190 EQUB @OPADP%
200 EQUW &0008
210 LDA &00
220 TAD
230 COP
240 EQUB @OPBHA%
250 EQUS "BASIC"
260 EQUB &00
270 :
280 LDY# &00
290 COP
300 EQUB @OPRFR%
310 BCS end%
320 :
330 PHK
340 JSR callmodule%
350 :
360 .end%
370 RTL
380 :
390 .callmodule%
400 PEI &04
410 PEI &02
420 PEI &00
430 RTL

```

\ Put processor in word mode.  
 \ Tell assembler that processor is in word mode.  
 \ Execute COP call.  
 \ Allocate direct page.  
 \ 8 bytes dp required.  
 \ Offset into DP for OPRFR.  
 \ Transfer into Direct Page register.  
 \ Execute COP call.  
 \ Routine to make BHA point to the following string.  
 \ Make BHA point to the start of this string.  
 \ Terminate string with zero.

\ Clear Y to fulfill entry conditions.  
 \ Execute COP call.  
 \ Reference module whose name is pointed to by BHA.  
 \ The 8 byte block in direct page pointed to by X  
 \ is initialised by the OPRFR routine  
 \ to provide the necessary addresses for  
 \ entering and returning from the module.  
 \ If the module is not found then end.

\ Push the high byte of the program counter  
 \ on the stack.  
 \ Jump to the subroutine which calls the module.  
 \ Return from module to here.  
 \ Finish here.

\ Push 2 bytes contained in locations D+5 and D+4 on to the stack.  
 \ Push 2 bytes contained in locations D+3 and D+2 on to the stack.  
 \ Push 2 bytes contained in locations D+1 and D+0 on to the stack.  
 \ Jump to module entry point at address  
 \ just pushed on stack.

Lines 330 and 340 call the routine which calls the module. PHK followed by JSR has a similar effect to JSL, but with the advantage that the 2 instructions are bank independent, thereby conforming to the requirement on Communicator. The above code can therefore be used in any bank.

Lines 400, 410 and 420 push 6 bytes on to the stack. The top 3 (ie those at D+0 to D+2) form the address of the module's entry point minus one. The RTL in line 430 uses this address to enter the module routine.

When the module routine terminates, with an RTL, the first 3 bytes pushed (ie those at D+3 to D+5) are now on the top of the stack. These form a return address at which there is code which acts as an RTL for the initial PHK JSR sequence of lines 330 and 340, thereby returning control to the user's code at line 350.

The other 2 bytes in the 8 byte block are used by the operating system.

Note that the above explanation is merely a guide and the programmer *must not* assume the format of the bytes in the 8 byte block.

The PHK : JSR callmodule% can be called many times, and has much less overhead than calling OPCMD each time, hence this method of calling a module should be used in preference if high speed is required. However, OPURF must be called when access to the module is no longer required.

# OPRMI read module info

**Action:** Read information about the module whose name is pointed to by BHA. The entry address can be read, or the flag and code type bytes.

**On entry:** X = hint (returned by OPSLM) or 0.  
 Y = 0 to read entry address (address of BRL).  
 Y = 2 to read the code type bytes.  
 Y = 4 to read module version.

Note: this COP must be called in 16-bit XY mode to read the module-type flags.

**On exit:** C = 0 means that the module was found and module information read.  
 C = 1 means that the module was not found.  
 If C = 0 and Y = 0 then BHA contains the entry address of the module, this being the address of the BRL, the first byte of the module header.  
 If C = 0 and Y = 2 then HA contains code type bytes 0 and 1, X contains code type bytes 2 and 3.  
 If C = 0 and Y = 4 then HA = version.  
 D preserved

**To call:**  
**COP**  
**EQUB @OPRMI%** from BASIC  
**OPSYS \$OPRMI** from MASM

**Example:**

```

170 COP          \ Execute COP call.
180 EQUB @OPBHA% \ Routine to make BHA point to the following string.
190 EQUUS "BASIC" \ Make BHA point to the start of this string.
200 EQUB $00      \ Terminate string with zero.
210 :
220 LDX# $00     \ Initialise X.
230 LDY# $00     \ Initialise Y to $00 to read entry address
                  \ or to $02 to read the information bytes.
240 COP          \ Execute COP call.
250 EQUB @OPRMI% \ Read info of module whose name is
                  \ pointed to by BHA.

```

This example will read the entry address of the BASIC module. The entry address is returned in BHA.  
 This COP must be called in 16-bit XY mode to read the module-type flags.

# OPFMA find module given address within module

**Action:** Given an address in BHA, this call will return in BHA the address of the name of the module that exists at that address. If no module exists at that address then the call returns with the carry bit set.

**On entry:** BHA points to somewhere within a module.

**On exit:** If C = 0 then BHA points to the module name.  
If C = 1 then the module was not found.  
D preserved

**To call:** COP  
EQUB @OPFMA% from BASIC  
OPSYS £OPFMA from MASM

**Example:**

30 address% = &F80000 :REM Initialise address	\ Load the accumulator with
170 LDA# address% DIV &10000	\ the high byte of the address.
180 PHA	\ Push the high byte on to the stack.
190 PLB	\ Pull the high byte into the bank register.
195 BNK address%/&10000	\ Tell assembler that B = high byte of address.
200 LDA# (address% AND &FF00) DIV &100	\ Load the accumulator with
210 SWA	\ the middle byte of the address.
220 LDA# (address% AND &FF)	\ Put the middle byte in H.
230 COP	\ Load the accumulator with
240 EQUB @OPFMA%	\ the low byte of the address. \ Execute COP call. \ Read address of name of module which \ contains the address stored in address%.

This example will return either with C = 0 and the address of the module name in BHA, or with C = 1.

# OPWRM write module name

**Action:** Given an address in BHA, this call will send the name of the module which exists at that address to the VDU drivers. If no module exists at that address then the call returns with the carry bit set.

**On entry:** BHA points to somewhere within a module.

**On exit:** If C = 0 then the name of the module has been sent to the VDU drivers.  
If C = 1 then the module was not found.  
D preserved

**To call:**  
COP  
EQUB @OPWRM% from BASIC  
OPSYS £OPWRM from MASM

**Example:**

30 address% = &F80000 :REM Initialise address  
170 LDA# address% DIV &10000

180 PHA

190 PLB

195 BNK address% /&10000

200 LDA# (address% AND &FF00) DIV &100

210 SWA

220 LDA# (address% AND &FF)

230 COP

240 EQUB @OPWRM%

- \ Load the accumulator with
- \ the high byte of the address.
- \ Push the high byte on to the stack.
- \ Pull the high byte into the bank register.
- \ Tell assembler that B = high byte of address.
- \ Load the accumulator with
- \ the middle byte of the address.
- \ Put the middle byte in H.
- \ Load the accumulator with
- \ the low byte of the address.
- \ Execute COP call.
- \ Send name of module which contains
- \ the address stored in address% to the
- \ VDU drivers.

This example will send the name of the module which exists at *address%* to the VDU drivers. If no module exists at *address%* then the call returns with C = 1.

# OPSLM scan list of modules

**Action:** This call returns with BHA pointing to a location containing the name of the next module in the list. X and Y return hints used by the operating system to find this module quickly.

**On entry:** BHA either points to the name of the previous module, or to a free area of memory containing a number of spaces terminated by zero (&00). In the latter case, BHA is preserved, and the name of the first module in the list is written to the location pointed to by BHA. The first name in the list is either truncated or padded with spaces to fit the number of spaces allocated on entry.  
 X = hint (returned by previous OPSLM), or 0.  
 Y = hint (returned by previous OPSLM), or 0.

**On exit:** BHA points to the name of the next module.  
 X = hint (number used by the operating system).  
 Y = hint (number used by the operating system).  
 No registers preserved

**To call:**  
**COP**  
**EQUB @OPSLM%** from BASIC  
**OPSYS ?OPSLM** from MASM

**Example:**

```

170 COP          \ Execute COP call.
180 EQUB @OPBHA% \ Make BHA point to the following string.
190 EQU$ STRING$(16," ") \ Space in which first name will be written.
200 EQUB &00      \ Terminated by zero (&00).
210 LDX# &00      \ Initialise X.
220 LDY# &00      \ Initialise Y.
230 COP          \ Execute COP call.
240 EQUB @OPSLM% \ Scan module list.
                  \ BHA now points to a location containing
                  \ the name of the next module.
                  \ X and Y contain hints.
                  \ Next call of OPSLM will therefore require no
                  \ initialisation to find the next module in the list.
250 COP          \ Execute COP call.
260 EQUB @OPSLM% \ Scan module list.
                  \ BHA now points to a location containing
                  \ the name of the next module, etc.

```

This example will scan the module list from the beginning. On entry, BHA points to a string terminated by zero (&00). In the above example the string contains 16 spaces, which means that OPSLM will scan from the beginning of the module list and will write as much of the first module name as will fit into 16 characters. Only the module name is written and not the optional commands in the module header. If the module name is longer than 16 characters then the written name is truncated to 16. If the string at line 190 in the above example were, for example, EQU\$ "BASIC" then OPSLM would return the address of the name of the next module following BASIC, truncated to 5 characters.

# OPFPO

find pool owner

**Action:** This call attempts to find the module which owns the memory pool pointed to by BHA.

**On entry:** BHA points to the pool address.

Y = 0

**On exit:** If C = 0 then the pool owner has been found, and BHA points to an address within the owner module.

If C = 1 then the owner was not found.

No registers preserved

**To call:**

- COP
- EQUB @OPFPO% from BASIC
- OPSYS EOPFPO from MASM

**Example:**

30 pool% = &025000 ;REM Initialise address within pool	\ Load the accumulator with
170 LDA# pool% DIV &10000	\ the high byte of the address of the pool.
180 PHA	\ Push this byte on to the stack.
190 PLB	\ Pull this byte into the bank register.
195 BNK pool% /&10000	\ Tell assembler that B = high byte of pool.
200 LDA# (pool% AND &FF00) DIV &100	\ Load the accumulator with
210 SWA	\ the middle byte of the address of the pool.
220 LDA# (pool% AND &FF)	\ Put this byte in H.
230 LDY# &00	\ Load the accumulator with
240 COP	\ the low byte of the address of the pool.
250 EQUB @OPFPO%	\ BHA now points to the pool address.
260 BCS end%	\ Load Y with zero to
270 COP	\ fulfill the entry requirements.
280 EQUB @OPWRM%	\ Execute COP call.
290 .end%	\ Find pool owner.
	\ If failed to find pool owner then end.
	\ Execute COP call.
	\ Write owner module name.
	\ End.

This example will attempt to find the owner module of the pool whose address is contained in *pool%*. If the attempt is successful then the name of the owner module will be written to the current output stream using OPWRM.

## OPNLU name lookup

- Action:** This call checks a list of names to see if a specific name is present. If it is then the offset of the name from the start of the list is given in Y.
- On entry:** The 4 byte address at D,0 points to the start of the name list.  
The 4 byte address at D,4 points to the name to be located.  
If Y = 0 then the slash "/" character is used as a delimiter.  
If Y < 0 then the character whose ASCII code is in Y is used as a delimiter.
- On exit:** If C = 0 then the name is in the list at offset Y from the start.  
If C = 1 then the name is not in the list.  
No registers preserved
- To call:**
- |                         |
|-------------------------|
| COP                     |
| EQUB @OPNLU% from BASIC |
| OPSYS £OPNLU from MASM  |

**Example:**

```

30 DIM list% &100 :REM allocate space for list
40 $list% = "HELLO/BASIC/FRED" :REM Initialise list
50 name$ = "BASIC" :REM name for which to search
170 COP                                \ Execute COP call.
180 EQUB @OPADP%                         \ Allocate direct page
190 EQUW &0008                          \ of 8 bytes.
200 TAD                                \ Put the allocated direct page
210 LDA# (list% AND &FF)                \ address into D.
220 STA &00                            \ Load A with the low byte
230 LDAM (list% AND &FF00) DIV &100    \ of the address of the name list.
240 STA &01                            \ Store this byte at D + 0.
250 LDAM list% DIV &10000              \ Load A with the next byte
260 STA &02                            \ of the address of the name list.
270 LDAM &00                            \ Store this byte at D + 1.
280 STA &03                            \ Load A with the next byte
290 PHK                                \ of the address of the name list.
300 PER name%                           \ Store this byte at D + 2.
310 PLA                                \ Load A with zero
320 STA &04                            \ (the fourth byte of the address must be zero).
330 PLA                                \ Push the 2 bytes of the address
340 STA &05                            \ of the label name% on to the stack.
350 PLA                                \ Push the high byte of the program
360 STA &06                            \ counter on to the stack.
370 LDAM &00                            \ Push the 2 bytes of the address
380 STA &07                            \ of the label name% on to the stack.
390 LDY# &00                            \ Pull the low byte into A.
400 COP                                \ Store this byte at D + 4.
410 EQUB @OPNLU%                         \ Pull the next byte into A.
420 TDA                                \ Store this byte at D + 5.
430 COP                                \ Pull the next byte into A.
440 EQUB @OPF2B%                         \ Store this byte at D + 6.
450 RTL                                \ Load A with zero
460 .name%                             \ (the fourth byte of the address must be zero).
470 EQUUS name$                          \ Store this byte at D + 7.
480 EQUB &00                            \ Initialise Y to indicate / delimiter.
                                         \ Execute COP call.
                                         \ Look up name in list.
                                         \ Copy direct page into HA.
                                         \ Execute COP call.
                                         \ Free direct page.
                                         \ Return from subroutine.
                                         \ Set up label for name string.
                                         \ Place the name string in memory.
                                         \ Terminated by zero (&00).

```

This example will allocate a direct page of 8 bytes to hold the addresses of the name list and the name to be located. The address of the start of the name list is held in *list%* and the address of the name to be located is held in the label *name%*. These addresses are stored in their correct places in direct page. The name to be located is held in *name\$*. If the carry is clear on exit then the name exists in the name string at an offset given by *Y*.

# OPAM add module

**Action:** Adds a module to the list of modules held by the MOS.

**On entry:** Y holds the handle of the memory pool in which the module has been loaded.

**On exit:** C = 0 means that the module was added to the list.  
C = 1 means that the module was not added to the list.  
No registers preserved

**To call:**

COP	
EQUB @OPAM%	from BASIC
OPSYS £OPAM	from MASM

**Example:**

```

170 RSP# &30          \ Put processor in word mode.
180 WRD &30          \ Tell assembler that processor is in word mode.
190 LDY# poolhandle%  \ Load Y with memory pool handle.
200 PHY               \ Save pool handle on stack.
210 COP               \ Execute COP call.
220 EQUB @OPAM%       \ Add module to list.
230 PLY               \ Restore pool handle to Y.
240 BCC end%         \ Module added to list so end.
250 LDX# @MMFP%      \ Load X with reason code to free pool.
260 COP               \ Execute COP call.
270 EQUB @OPMM%      \ Call memory management.
280 :                \
290 COP               \ Execute COP call.
300 EQUB @OPBHA%     \ Print following string.
310 EQUUS "Can't install module" \ Error message.
320 EQUB &00          \ Terminated by zero (&00).
330 SEC               \ Set carry to indicate error.
340 end%             \ End.

```

Note: this example assumes that a suitable memory pool has been allocated and the module has been loaded into this memory.  
*poolhandle%* contains the handle to the memory pool.

This example will add the module held in the memory whose pool handle is in *poolhandle%* to the list of modules held by the MOS.

## 3. Coroutines

### 3.1 Introduction

A Coroutine is a self-contained unit, comprising code and data, dedicated to implementing one or more well defined operations. Coroutines are used to build applications requiring loosely coupled processes, but where the full power of multiprocessing is not required. A coroutine has context, that is a set of information, (including dynamic-data and processor registers), which must be maintained in order for it to function correctly. Coroutines must not access the data structures of other routines directly because of the risk of context corruption. Data exchange between routines should be restricted to processor registers via coroutine function calls, or if really necessary, global memory locations.

Coroutines allow the designer to separate functionally independent sections of code.

Coroutines provide advantages in terms of parallel software development. Communication between routines is restricted, meaning that the interface to the software within the coroutine is usually tightly defined. Coroutines can be developed and tested in isolation before being integrated with the rest of a system which is usually easier than attempting to implement, test and integrate pieces of loosely related code within a very large program.

Coroutines help maintain system integrity. If a coroutine fails for some reason, it need not imply the failure of the entire system.

The operating system provides a facility to create and call routines.

Each routine has its own stack, and may be given its own direct page.

Coroutines are organised in a hierarchy. Once a coroutine is created (by its "parent") it may be called using the COCALL procedure. This starts execution at the address specified when the coroutine was created, and suspends operation of the caller.

Coroutines are always in 1 of 3 states:

Current

Active

Dormant

A parent must be current to create a coroutine. When the coroutine is first created it is dormant. When it is called using COCALL, the parent is active, and the coroutine is now current.

When the coroutine subsequently does a COWAIT, control returns to the caller, which restarts from just after the COCALL. The parent is once again current and the coroutine is dormant.

The parent may then do another COCALL later on to restart the coroutine from the point just after the last COWAIT.

### *Chapter 3*

Example:

```
CO1 = CREATE(PROCESS_CH)
CH = COCALL(CO1)
CH = COCALL(CO1)
CH = COCALL(CO1)
```

```
PROCESS_CH
```

```
LOOP
```

```
    : Initialise
    :
    :
    : Make result
    :
    :
    COWAIT(CH)
    JMP LOOP
```

This example gives an idea of how a coroutine is called, but the example will not run - in practice "Initialise" and "Make result" will be a sequence of instructions. Section 3.2 contains a executable listing of an example program to be run in conjunction with the test harness.

The coroutine CO1 is created, and the address of the coroutine is PROCESS\_CH.

When the coroutine is first called using COCALL, the code at PROCESS\_CH is executed until the COWAIT instruction is encountered, whence execution resumes in the parent immediately after the COCALL.

The second COCALL executes the coroutine from after the COWAIT, which hence jumps to LOOP. Thus the Initialise stage is left out.

This and each subsequent COCALL will execute only the "Make result" section.

In a more realistic hierarchy, the CHILD coroutine may create one or more coroutines which it calls with COCALL. When the COWAIT is executed, control is returned to the next instruction in CHILD.

Another way of calling a coroutine is with CORESUME. This passes control to a "sibling" coroutine, ie one which shares the same parent as the current coroutine. If this sibling COWAITs, control is returned to the mutual parent of the coroutines at the instruction after where it called the first child. If the sibling does another CORESUME then control is either passed on to the next sibling, or, if there are only two, toggles between the two siblings each time a CORESUME is executed.

As an example, suppose the current coroutine PARENT has created two dormant children CHILD1 and CHILD2. PARENT does a COCALL(CHILD1) and CHILD1 becomes current. Somewhere in CHILD1, a CORESUME(CHILD2) is performed and CHILD2 becomes current, and CHILD1 becomes dormant. When CHILD2 subsequently executes a COWAIT, execution is returned to PARENT at the instruction just after where it called CHILD1, and both CHILD1 and CHILD2 become dormant. If instead of a COWAIT, CHILD2 executes a CORESUME then control passes back to CHILD1 at the instruction just after the original CORESUME, and CHILD2 becomes dormant.

Each coroutine has its own stack, and may be given its own direct page. The latter is set by calling the memory management system to allocate space, and then COENV to inform the coroutine system. Alternatively, a COP instruction may be used to allocate direct page.

Communication between coroutines is through the BHA and X registers and the C flag. In addition coroutines may pass data through "global" locations. Note that the stack may not be used to pass results or arguments as each coroutine has its own stack.

Error handling in coroutines reflects their hierarchical nature. Each coroutine has the ability to define its own BRK handler address via the CO reason code COBRK . If necessary, the break handler for the active coroutine can be cancelled via the CO reason code COCBH, which resets the break handler to "no handler". When a BRK is encountered, the address set by the coroutine which was active when the BRK happened will be called. If no such address was set, the parent coroutine's BRK handler address is used, if one is set. The search for a handler continues up the coroutine hierarchy until a break handler is found. If the top level is reached without a handler being found, an OPERR (fatal error) is executed.

### 3.2 An example demonstrating the use of coroutine functions

The following piece of code demonstrates the use and invocation of some of the available coroutine functions.

The program creates and invokes a top level coroutine, "coroutine1". Coroutine1 creates two children, "coroutine2" and "coroutine3". Coroutine1 then invokes coroutine2 via the CCO function, which in turn invokes its sibling, coroutine3, via the CRS function. Coroutine3 suspends itself, passing control back to its parent, coroutine1, via the CWT function. Coroutine1 deletes coroutine2 and coroutine3 via the CODEL function, before passing control back to the BASIC module. Coroutine1 is deleted and the program exits.

The code should be run in conjunction with the test harness described in Chapter 1.

```

20 costksz% = &200
30 codps% = &100
170 RSP# &30
175 WRD &30
180 PHK
185 PLB
190 BNK P%/&1000
195 LDA# coentry1% AND &FFFF
200 PEAS costksz%
205 PLD
210 LDX# @COCRE%
215 JSRL @C0%
220 BCC BR1%
225 BRL coerr1%
230 :
235 .BR1%
240 COP
245 EQUB @OPADP%
250 EQUW codps%
255 BCC BR2%
260 BRL coerr1%
265 :
270 .BR2%
275 TAD
280 LDX# @COENV%
285 JSRL @C0%
290 PHY
295 PLA
300 STA cohndl1%
305 JSRL @CCO%
310 LDA cohndl1%
315 PHA
320 PLY
325 LDX# @CODEL%
330 JSRL @C0%
335 BCC BR3%
340 BRL coerr1%
345 .BR3%
350 RTL
355 :
360 :
365 .coentry1%
370 COP
375 EQUB @OPWRS%
380 EQU$ "Hello, Im Coroutine 1!"
385 EQUB &0A
390 EQUB &0D
395 EQUB &00
400 :
405 LDA# coentry2% AND &FFFF
                                \ Make sure processor is in word mode.
                                \ Tell assembler that processor is in word mode.
                                \ Coroutine entry is in my program bank.
                                \
                                \ Tell assembler that B = K.
                                \ Entry Address in BHA.
                                \ Coroutine stack size in D on entry.
                                \
                                \ Reason code for create in X.
                                \ Call coroutine entry.
                                \ Carry on if no errors.
                                \ Error.
                                \
                                \
                                \ Execute COP call.
                                \ Call the direct page allocator.
                                \ With DP size as argument.
                                \ Carry on if no errors.
                                \ Error.
                                \
                                \
                                \ Get Direct Page in D.
                                \ Set the coroutine's environment.
                                \
                                \ Transfer coroutine1's handle into A...
                                \
                                \ ... and save it.
                                \ Call the coroutine.
                                \ Recover coroutine1's handle.
                                \ Transfer it to Y.
                                \
                                \ Reason code for delete in X.
                                \ Call coroutine entry.
                                \ Carry on if no errors.
                                \ Error.
                                \
                                \
                                \
                                \ This is the entry point of coroutine1.
                                \ Execute COP call.
                                \ In this case output a string.
                                \
                                \
                                \
                                \ Entry address of coentry2 in BHA.

```

```

410 PEA# costksz%
415 PLD
420 LDX# @COCRE%
425 JSRL @CO%
430 BCC BR4%
435 BRL coen2%
440 .BR4%
445 COP
450 EQUB @OPADP%
455 EQUW coops%
460 BCC BR5%
465 BRL coen2%
470 .BR5%
475 TAD
480 LDX# @COENV%
485 JSRL @CO%
490 PHY
495 PLA
500 STA cohandle2%
505 :
510 LDA# coentry3% AND &FFFF
515 PEA# costksz%
520 PLD
525 LDX# @COCRE%
530 JSRL @CO%
535 BCC BR6%
540 BRL coen3%
545 .BR6%
550 COP
555 EQUB @OPADP%
560 EQUW coops%
565 BCC BR7%
570 BRL coen3%
575 .BR7%
580 TAD
585 LDX# @COENV%
590 JSRL @CO%
595 PHY
600 PLA
605 STA cohandle3%
610 :
615 LDA cohandle2%
620 PHA
625 PLY
630 JSRL @CCO%
635 LDA cohandle2%
640 PHA
645 PLY
650 LDX# @CODEL%
655 JSRL @CO%
660 BCC BR8%
665 BRL coen2%
670 .BR8%
675 LDA cohandle3%
680 PHA
685 PLY
690 LDX# @CODEL%
695 JSRL @CO%
700 BCC BR9%
705 BRL coen3%
710 .BR9%
715 JSRL @CWT%

```

\ Coroutine stack size in D on entry.  
 \ .  
 \ Reason code for create in X.  
 \ Call coroutine entry.  
 \ Carry on if no errors.  
 \ Error.  
 \ .  
 \ Execute COP call.  
 \ Call the direct page allocator.  
 \ With DP size as argument.  
 \ Carry on if no errors.  
 \ Error.  
 \ .  
 \ Get Direct Page in D.  
 \ Set coroutine2's environment.  
 \ .  
 \ Get coroutine2's handle into A...  
 \ .  
 \ ... and save it.  
 \ .  
 \ Address of coroutine3 in BHA.  
 \ Coroutine stack size in D on entry.  
 \ .  
 \ Reason code for create in X.  
 \ Call coroutine entry.  
 \ Carry on if no errors.  
 \ Error.  
 \ .  
 \ Execute COP call.  
 \ Call the direct page allocator.  
 \ With DP size as argument.  
 \ Carry on if no errors.  
 \ Error.  
 \ .  
 \ Get direct page in D.  
 \ Set coroutine3's environment.  
 \ .  
 \ Get coroutine3's handle into A...  
 \ .  
 \ ... and save it.  
 \ .  
 \ Get coroutine2's handle into A.  
 \ Transfer it to Y.  
 \ .  
 \ Call coroutine2.  
 \ Recover coroutine2's handle.  
 \ Transfer it to Y.  
 \ .  
 \ Reason code for delete in X.  
 \ Call coroutine entry.  
 \ Carry on if no errors.  
 \ Error.  
 \ .  
 \ Recover coroutine3's handle.  
 \ Transfer it to Y.  
 \ .  
 \ Reason code for delete in X.  
 \ Call coroutine entry.  
 \ Carry on if no errors.  
 \ Error.  
 \ .  
 \ Return to calling routine.

*Chapter 3*

```
720 :          \
725 :          \
730 .coentry2% \ This is the entry point for coroutine2
735 COP          \ Execute COP call ...
740 EQUB @OPWRS \ ... to output a string.
745 EQUS "Hello, Im Coroutine2!" \
750 EQUB &0A \
755 EQUB &0D \
760 EQUB &00 \
765 :          \
770 LDA cohandle3% \ Get coroutine3's handle into A.
775 PHA          \ Transfer it to Y.
780 PLY          \
785 JSRL @CRS% \ Invoke sibling, coroutine3.
790 JSRL @CWT% \ Suspend coroutine2.
795 :          \
800 :          \
805 .coentry3% \ This is the entry point for coroutine3.
810 COP          \ Execute COP call...
815 EQUB @OPWRS% \ ... to output a string.
820 EQUS "Hello, Im coroutine3!" \
825 EQUB &0A \
830 EQUB &0D \
835 EQUB &00 \
840 :          \
845 JSRL @CWT% \ Suspend coroutine3 and return to coroutine1.
850 :          \
855 :          \
860 .coerr1% \ Error message for coroutine1.
865 BRK          \
870 EQUB &00 \
875 EQUS "Error with coroutine1."
880 EQUB &00 \
885 :          \
890 :          \
895 .coerr2% \ Error message for coroutine2.
900 BRK          \
905 EQUB &00 \
910 EQUS "Error with coroutine2"
915 EQUB &00 \
920 :          \
925 :          \
930 .coerr3% \ Error message for coroutine3.
935 BRK          \
940 EQUB &00 \
945 EQUS "Error with coroutine3."
950 EQUB &00 \
955 :          \
960 :          \
965 .cohandle1 \ Storage for coroutine1's handle.
970 EQUW &0000 \
975 .cohandle2 \ Storage for coroutine2's handle.
980 EQUW &0000 \
985 .cohandle3 \ Storage for coroutine3's handle.
990 EQUW &0000 \
```

Lines 195-285 create coroutine1 and its associated environment.

Line 305 invokes coroutine1 via the CCO function.

Lines 310-350 delete coroutine1 and exit from the program.

Line 365 is the entry point for coroutine1.  
Lines 370-395 Output a message to the screen from coroutine1.  
Lines 405-485 create coroutine2 and its associated environment.  
Lines 510-590 create coroutine3 and its associated environment.  
Line 630 invokes coroutine2 from coroutine1 via the CCO function.  
Lines 635-705 delete coroutine2 and coroutine3 via CODEL.  
Line 715 returns control to BASIC via CWT.  
Line 730 is the entry point for coroutine2.  
Lines 735-760 output a message to the screen from coroutine2.  
Line 785 invokes coroutine3 via the CRS function.  
Line 805 is the entry point for coroutine3.  
Lines 810-835 output a message to the screen from coroutine3.  
Line 845 returns control to coroutine1 via CWT.

### 3.3 Coroutine functions

The main coroutine function is CO. Which provides facilities for the creation and deletion of coroutines and their associated environment.

CO is invoked with one of the following reason codes in register X:

COCRE	Create a coroutine
COENV	Set a coroutine's environment
COREN	Restore current coroutine's environment
CODEL	Delete a coroutine
CODES	Destroy coroutine
COBRK	Set the coroutine break handler address
COCBH	Cancel break handler for this coroutine
COIAM	Return handle of the current coroutine
COINTI	Initialise language
COKILL	Kill language
COHELP	Language help

In addition the following coroutine functions are available:

CCO	Call coroutine
CWT	Suspend a coroutine
CRS	Resume a sibling coroutine

# CO Create, delete, environment

This call is used to create a coroutine, set an existing coroutine's environment, or delete an existing coroutine.

## COCRE - Create a coroutine

This sets up the initial execution address of a new coroutine and declares its stack size.

<b>Entry:</b>	BHA	Entry address of coroutine
	D	Desired size of coroutine's stack
	X	COCRE
<b>Exit:</b>	C=0	Coroutine created successfully
	Y	Handle to be used in subsequent calls
or	C=1	Coroutine could not be created

Notes: The stack size must be a non-zero multiple of 256 bytes - &200 is a good size.

## COENV - Set a coroutine's environment

This specifies the direct page to be used whenever a coroutine is entered. If the coroutine is to use any shared direct page workspace (as distinct from direct page workspace allocated by itself), this should be called immediately after a coroutine is created, or at least before it is called for the first time.

<b>Entry:</b>	Y	Handle
	D	Direct page to be used by coroutine
	X	COENV
<b>Exit:</b>	C=0	Environment set correctly
or	C=1	Environment could not be set

## COREN - Restore current coroutine's environment

This call may be used to restore D to the value set by a previous call to COENV. It is useful as many of the OS calls corrupt D.

<b>Entry:</b>	X	COREN
<b>Exit:</b>	D	Value set in previous call to COENV

## CODEL - Delete a coroutine

This deletes an existing coroutine.

<b>Entry:</b>	Y	Handle
	X	CODEL
<b>Exit:</b>	B,D	Preserved
	C=0	Coroutine deleted successfully
or	C=1	Coroutine could not be deleted

### **CODES - Destroy coroutine**

This acts as CODEL, but does not check that the routine to be deleted has a parent. The use of this call is to delete a routine in which a BRK (error condition) has occurred, and which has no BRK handler of its own.

<b>Entry:</b>	Y	Handle
	X	CODES
<b>Exit:</b>	B,D	Preserved
	C=0	Coroutine was deleted
or	C=1	Coroutine could not be deleted

### **COBRK - Set the coroutine break handler address**

This routine sets the address of the routine to be called when a BRK is encountered within the current coroutine.

<b>Entry:</b>	BHA	BRK handler address
	X	COBRK
<b>Exit:</b>	C=0	Address was set OK
or	C=1	Address could not be set

### **COCBH - Cancel break handler for this coroutine**

This call resets the break handler for the current coroutine to a "no handler" state (as when the coroutine is created). If a BRK occurs in a routine with no BRK handler, the operating system will attempt to use its parent's handler. This carries on up the hierarchy until a valid BRK handler is found or the top level is reached. In the latter case, an OPERR COP instruction is executed.

<b>Entry:</b>	X	COCBH
<b>Exit:</b>	B,D	Preserved
	C=0	Break handler was cancelled
	BHA	Address of old break handler
or	C=1	There was no break handler to cancel

### **COIAM - Return handle**

This call returns the handle of the current coroutine, ie the coroutine from which the call to COIAM is made.

<b>Entry:</b>	X	COIAM
<b>Exit:</b>	B,D	Preserved
	C=0	The handle was found
	Y	The current coroutine's handle
	C=1	No handle could be found for the caller

### **COINIT - Initialise language**

### **COKILL - Kill language**

## **COHELP - Language help**

These calls relate to the language entry point reason codes. The CO reason codes call a given coroutine at its entry point as if it were a language, with the appropriate reason code in X. The entry is called as a subroutine, not a coroutine, so is expected to return with an RTL.

**Entry:**    Y              Handle of coroutine to be called  
              BHA            Arguments  
              X              "

**Exit:**    B,D            Preserved  
              BHA,X,Y      Results from language entry

## CCO - Call coroutine

This is used to call a coroutine from its creator (parent). On the first call the entry address given in the COCRE call is used. Subsequently, the address after the last CWT executed by the routine is used.

**Entry:**      Y            Handle of child  
                BHA,X,C    Arguments to child coroutine

**Exit:**        BHA,X,C    Results from child coroutine  
                B,D        Preserved

Note: C is the C flag.

# CWT - Suspend a coroutine

This call causes the execution of the current (child) coroutine to be suspended and control to be passed back to its parent.

**Entry:**    BHA,X,C    Results to be passed to parent

**Exit:**    BHA,X,C    Arguments from next call by parent  
              D           As determined by set environment

## **CRS** - Resume a sibling coroutine

This acts as a CWT, but instead of returning control back to the parent, it passes control to a dormant sibling (ie another child of its parent). It acts effectively as if the calling coroutine did a CWT and the parent immediately did a CCO to the sibling coroutine. When the sibling subsequently does a CWT, control returns to the parent.

<b>Entry:</b>	<b>Y</b>	Handle of sibling coroutine
	<b>BHA,X,C</b>	Arguments to sibling routine
<b>Out:</b>	<b>BHA,X,C</b>	Arguments from next call by parent
	<b>D</b>	As determined by set environment

## 4. Memory management

The operating system manages the allocation of memory in the machine. Requests may be made by programs for small (less than &10000 or 65536) or large (more than &10000) areas of memory. In fact, the calls to allocate a large area may be used to allocate less than &10000 bytes, but the space allocated may (or may not) span a bank boundary. Memory allocated by the "small area" calls is guaranteed to lie in one bank only. "Large ascending" calls should be used in preference wherever possible, even if the amount of memory requested is "small".

Free space is divided into "pools" which are accessed through handles allocated by the system. Handles are always 16-bit and so all calls to the memory management routine must be made in 16-bit XY mode. Memory may be requested in the ascending or descending direction. In the former case, it is taken from the lower region of the pool; in the latter case it is taken from the upper region. Thus the amount of unallocated space in a pool shrinks towards the middle as space is allocated from it.

Bank zero is taken by the memory management system to be at the top of the pool, so it is customary to allocate ascending pools. A descending call will usually allocate memory in bank zero and so should not be used. If bank zero is required then use the COP call OPADP.

When a task has finished with a block of memory, it may return it to the pool, so that another task may use it. Initially there are several defined. These have symbolic handles accessible through the usual MASM £ symbols or BASIC @...% pseudo-variables. The initial pools are:

HDMMW	Entire machine memory map pool
HDMMM	Entire memory pool
HDMMT	Task memory
HDMMC	CMOS RAM pool
HDMMV	Screen memory pool

Large or small areas of memory may be allocated. "Large" is defined to be greater than &10000 bytes; "small" is zero to &FFFF bytes. The task memory pool is used for allocating large areas of workspace for major tasks. These large areas may then be divided between routines of the task by passing the handle of the pool to the routines. Alternatively, the allocated workspace might be used by the task as its main work area (eg the program area for the BASIC task).

To allocate bank zero workspace, a special call (MMAZB) is provided which is guaranteed to allocate memory from bank zero. However, the most frequent use of bank zero is for direct page workspace and for stacks. COP calls OPADP and OPAST are available for these purposes.

Note that the end address of a pool is one higher than the last location that is actually available. This does not usually concern programmers, as they deal with base and length attributes rather than base and end addresses.

Note that a further allocation from the old pool would start from the end of the new pool. The call to return the end address of a pool gives the address of the first byte above the last byte inside the pool.

A descending allocation is similar to that shown above, except that the new pool is taken from the top of the parent pool.

The memory management functions are general purpose, not being restricted to allocation of system memory alone. Applications may use the functions for maintaining memory structures within their own allocated RAM.

There is only one call to the memory management interface. The various facilities provided by it are distinguished by the reason code in X on entry. These reason codes are given in subsequent sections.

Calls to the memory management routine are made using a COP with signature OPMM. The following reason codes are implemented by OPMM:

MMBAS	Return pool base
MMTOP	Return pool top
MMLEN	Return pool length
MMFND	Find a memory pool by address
MMASD	Allocate small descending area of pool
MMAZB	Allocate a bank zero area
MMASA	Allocate a small ascending area of pool
MMALD	Allocate large descending area of pool
MMALA	Allocate large ascending area of pool
MMAX	Allocate an explicit region within a pool
MMFP	Free a specified pool of memory
MMCHK	Check the integrity of the MM system
MMMRG	Merge two memory pools into one

In addition to the pool memory management interface COP call, there are some COP calls associated with more general memory management functions. These are:

OPADP	Allocate direct page
OPADF	allocate fast direct page
OPFZB	Free zero bank pool
OPAST	Allocate stack
OPFST	Free stack
OPCVS	Convert stack index
OPCVD	Convert direct page index
OPBYX	BYXD to BHA conversion

# OPMM call the memory management routine

- Action:** This calls the memory management routine. The reason code is passed in X, and the pool handle is passed in Y. All handles are 16-bit, so calls to OPMM must be made in 16-bit XY mode.
- On entry:** X = reason code.  
Y = pool handle.
- On exit:** If C = 0 then the call succeeded.  
If C = 1 then the call failed, X = error code, BHA points to zero-terminated error message.  
D preserved
- To call:** COP  
EQUB @OPMM% from BASIC  
OPSYS £OPMM from MASM

**Example:**

To call the memory management routine with reason code MMASA (allocate a small ascending area of memory).

```
170 RSP# &30      \ Put processor in word mode.  
175 WRD &30      \ Tell assembler that processor is in word mode.  
180 LDVY @HDMMT%  \ Load Y with the pool handle,  
190 LDX# @MMASA%  \ in this case, the task memory pool.  
200 LDA# &0100    \ Load X with the reason code,  
                  \ in this case MMASA.  
210 COP          \ Load HA with the number of bytes  
                  \ to be allocated.  
                  \ We'll choose 256 bytes.  
220 EQUB @OPMM%  \ Execute COP call.  
230 BCS end%     \ Call memory management routine.  
240 :            \ Memory not allocated so end.  
                  \ Continue from here.  
                  \ Y contains handle of newly allocated pool.  
250 .end%        \ BHA contains base address of newly allocated pool.  
260 RTL          \ End here.  
                  \ Return from subroutine.
```

This example will allocate an ascending pool of 256 bytes somewhere in task memory (task memory excludes such areas as ROM, CMOS RAM, VDU RAM).

### **MMBAS - Return pool base**

This call returns the base address of a given memory pool.

<b>Entry:</b>	<b>Y</b>	Handle of memory pool
	<b>X</b>	MMBAS
<b>Exit:</b>	<b>D,Y</b>	Preserved
	<b>C=0</b>	Success
	<b>BHA</b>	Base address of memory pool
or	<b>C=1</b>	The handle was not a valid one

Note: although the system performs some checking on the validity of the pool handle, it is possible to pass an undefined handle to the routine which is not detected as such.

### **MMTOP - Return pool top**

This routine returns the address of the byte after the last byte of the pool whose handle is given in Y. That is, if the last available byte in a pool is at address &0231FF, the address returned in BHA will be &023200.

<b>Entry:</b>	<b>Y</b>	Handle of memory pool
	<b>X</b>	MMTOP
<b>Exit:</b>	<b>D,Y</b>	Preserved
	<b>C=0</b>	Success
	<b>BHA</b>	Top address of pool
or	<b>C=1</b>	The handle was not a valid one.

See note for MMBAS

### **MMLEN - Return pool length**

This call returns the length in bytes of a given memory pool.

<b>Entry:</b>	<b>Y</b>	Handle of memory pool
	<b>X</b>	MMLEN
<b>Exit:</b>	<b>D,Y</b>	Preserved
	<b>C=0</b>	Success
	<b>BHA</b>	Length of memory pool
or	<b>C=1</b>	The handle was not a valid one

See note for MMBAS.

### **MMFND - Find a memory pool by address**

This routine takes an address and returns the handle of a pool containing that address if possible. Note that if the address is within a large pool, the handle will only be returned if that pool contains no sub-pools.

<b>Entry:</b>	BHA	Memory address within pool
	X	MMFND
	Y	HDMMT, the task pool handle
<b>Exit:</b>	D	Preserved
	C=0	A pool containing the address exists
	Y	Handle of pool
or	C=1	No pool found

### **MMASD - Allocate small descending area of pool**

This call allocates a region of memory from a specified pool. The region is taken from the top of the pool, and must be less than 64K bytes long. Moreover, the routine will only succeed if the pool would not cross a bank boundary.

<b>Entry:</b>	Y	Handle of memory pool
	HA	Number of bytes to allocate
	X	MMASD
<b>Exit:</b>	C=0	Pool allocated successfully
	Y	Handle of newly allocated pool
	BHA	Base address of newly allocated pool
or	C=1	Failed to allocate new pool
	B,D	Preserved

### **MMAZB - Allocate a bank zero area**

This call allocates memory from bank zero, in a way similar to MMASD. Note that this call should be used only when it is essential that the memory allocated is in bank zero, eg to store an interrupt routine. Bank zero is a relatively precious resource on the 65SC816 and should not be over used if many tasks are to be supported.

<b>Entry:</b>	HA	Number of bytes to allocate
	X	MMAZB
<b>Exit:</b>	C=0	Pool allocated successfully
	BHA	Base address of newly allocated pool
	Y	Handle of new direct page pool
or	C=1	Failed to allocate new pool
	B,D	Preserved

Note that the COPs OPADP, OPADF and OPAST use MMAZB to allocate their space. Also, although the address in BHA refers to a bank zero location, B may actually contain a non-zero value. This is because several banks "ghost" into bank zero due to the hardware memory decoding.

**MMASA - Allocate small ascending area of pool**

This call allocates a region of memory from a specified pool. The region is taken from the lower area of the pool. A maximum of 64K bytes may be allocated by this call. The call will only succeed if the pool would not cross a bank boundary.

<b>Entry:</b>	Y	Handle of memory pool
	HA	Number of bytes to allocate
	X	MMASA
<b>Exit:</b>	C=0	Pool allocated successfully
	Y	Handle of newly allocated pool
	BHA	Base address of newly allocated pool
or	C=1	Failed to allocate new pool
	B,D	Preserved

**MMALD - Allocate large descending area of pool**

This call allocates a region of memory from a specified pool. The region is taken from the upper region of the pool.

<b>Entry:</b>	Y	Handle of memory pool
	BHA	Number of bytes to allocate
	X	MMALD
<b>Exit:</b>	C=0	Pool allocated successfully
	Y	Handle of newly allocated pool
	BHA	Base address of newly allocated pool
or	C=1	Failed to allocate new pool
	B,D	Preserved

**MMALA - Allocate large ascending area of pool**

This call allocates a region of memory from a specified pool. The region is taken from the bottom of the pool.

<b>Entry:</b>	Y	Handle of memory pool
	BHA	Number of bytes to allocate
	X	MMALA
<b>Exit:</b>	C=0	Pool allocated successfully
	Y	Handle of newly allocated pool
	BHA	Base address of newly allocated pool
or	C=1	Failed to allocate new pool
	B,D	Preserved

## *Chapter 4*

### **MMAX - Allocate an explicit region within a pool**

This call allocates a sub-pool within a specified pool. The base address of new pool and the handle of the pool from which to take the new area are passed to the routine. The length of the new pool must be less than 64K bytes.

<b>Entry:</b>	Y	Handle of memory pool
	BHA	Base address of new pool
	D	Length of new pool
	X	MMAX
<b>Exit</b>	C=0	Pool allocated successfully
	BHA	Address of new pool
	Y	Handle of newly allocated pool
or	C=1	Failed to allocate new pool
	B	Preserved

### **MMFP - Free a specified pool of memory**

This call de-allocates a specified memory pool, allowing it to be re-used by another task.

<b>Entry:</b>	Y	Handle of pool to be freed
	X	MMFP
<b>Exit:</b>	B,D	Preserved
	C=0	Pool freed successfully
or	C=1	Pool could not be freed

### **MMCHK - Check the integrity of the MM system**

This call is a system-level routine to check that the data structures maintained by the memory management software is self-consistent. If the system is OK, the routine returns, otherwise an OPERR is executed.

<b>Entry:</b>	X	MMCHK
<b>Exit:</b>	D,B	Preserved
	A,X,Y	Preserved

### **MMMRG - Merge two memory pools into one**

This call combines two existing contiguous pools of memory into a single, larger one. The pools must not overlap, ie the call may not be used to merge a pool with one of its sub-pools.

<b>Entry:</b>	Y	Handle of pool 1
	HA	Handle of pool 2
	X	MMMRG
<b>Exit:</b>	C=0	The pools were merged successfully
	Y	Handle of composite pool
or	C=1	The pools could not be merged
	B,D	Preserved

Note that the handle of the composite pool may be different from both of the source handles.

## **4.1 Memory management COP calls**

There are a number of COP calls associated with memory management. These are described on the following pages.

# OPADP allocate direct page

**Action:** This call is used to request a direct page of a certain size, the size being a 2 byte number which follows the call. The call returns with HA pointing to an address in bank zero which can be used as the base of the direct page.

**On entry:** Inline 2 byte size (in bytes) of direct page requested.

**On exit:** C = 0 means that the direct page has been allocated, and HA contains the address of the base.

C = 1 means that the call failed to allocate a direct page.

XY preserved

Note: D is not preserved.

**To call:**

COP	
EQUB @OPADP%	
EQUW length%	from BASIC
OPSYS OPADP	
& \$LENGTH	from MASM

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPADP%	\ Allocate direct page with the
	\ following number of free bytes.
190 EQUW &0FF	\ In this case we'll choose 255 bytes.
200 TAD	\ HA will now contain the address of the direct page.
	\ Transfer the direct page address held in HA
	\ into the direct page register.

This example will allocate a direct page of 255 bytes length and place the start address of this direct page in the direct page register (D).

Note that OPFZB should be used to free the space allocated by OPADP after it is finished with.

# OPADF allocate fast direct page

**Action:** This call is identical to OPADP in every respect, except that, if possible, the direct page base allocated will lie on a page boundary. The call returns with HA pointing to an address in bank zero which can be used as the base of the direct page.  
As this call, by definition, allocates space inefficiently, it should only be used by applications where a fast direct page is imperative.

**On entry:** Inline 2 byte size (in bytes) of direct page requested.

**On exit:** C = 0 means that the direct page has been allocated, and HA contains the address of the base.

C = 1 means that the call failed to allocate a direct page.

XY preserved

Note: D is not preserved.

**To call:**

COP	
EQUB @OPADF%	
EQUW length%	from BASIC
OPSYS COPADF	
& \$LENGTH	from MASM

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPADF%	\ Allocate direct page with the
	\ following number of free bytes.
190 EQUW &001F	\ In this case we'll choose 31 bytes.
	\ HA will now contain the address of the direct page
	\ on a page boundary if possible (A = &00).
200 TAD	\ Transfer the direct page address held in HA
	\ into the direct page register.

This example will allocate a direct page of 31 bytes length and place the start address of this direct page in the direct page register (D).

Note that OPFZB should be used to free the space allocated by OPADF after it is finished with.

## OPFZB free zero bank pool

**Action:** Tells the operating system that space allocated in bank zero is finished with. This should be done every time space is no longer required so that bank zero is not exhausted.

**On entry:** HA contains the address of the bank zero pool base.

**On exit:** If C = 0 then the pool has been freed.

If C = 1 then the routine could not free the pool.

No registers preserved

**To call:** COP  
EQUB @OPFZB% from BASIC  
OPSYS tOPFZB from MASM

**Example:**

Add this example to the end of the example given for OPADP.

```
210 COP      \ Execute COP call.  
220 EQUB @OPFZB%\ Free zero page pool pointed to by HA.
```

This example will free the direct page allocated in the example given for OPADP.

# OPAST allocate stack

**Action:** This call is used to request a stack of a certain size, the size being a 2 byte number which follows the call. The call returns with HA containing an address which can be transferred to the stack pointer.

**On entry:** Inline 2 byte size (in bytes) of stack requested.

**On exit:** C = 0 means that the stack has been allocated, and HA contains the address to be transferred to the stack pointer.  
C = 1 means that the call failed to allocate a stack.  
DBXY preserved

**To call:**

COP	
EQUB @OPAST%	
EQUW length%	from BASIC
OPSYS COPAST	
& \$LENGTH	from MASM

**Example:**

170 RSP# &30	\ Put processor in word mode.
180 WRD &30	\ Tell assembler that processor is in word mode.
190 TSA	\ Put old stack pointer in HA.
200 STA temp%	\ Save the old stack pointer.
210 COP	\ Execute COP call.
220 EQUB @OPAST%	\ Allocate stack with the
	\ following number of free bytes.
230 EQUW &0200	\ In this case we'll choose 512 bytes.
	\ HA will now contain the address of the base of the stack.
	\ Remember that the stack moves downwards in memory
	\ so the base of the stack is actually the highest address.
240 TAS	\ Transfer the stack base address held in HA
	\ into the stack pointer.
250 :	\ Code for use with new stack goes here.
500 LDA temp%	\ Get old stack pointer in HA.
510 TAS	\ Restore old stack pointer.
520 RTL	\ End.
530 .temp%	\ Label at which to store old stack pointer.
540 EQUW &0000	\ 2 bytes space.

This example will allocate a stack of 512 bytes length and place the base address of this stack in the stack pointer (S). When returning to the BASIC environment (as in the case of the test harness) the original stack must first be restored.

Note that OPFST should be used to free the space allocated by OPAST after it is finished with.

## OPFST free stack

**Action:** Tells the operating system that space allocated to a stack is finished with. This should be done every time a stack is no longer required. Only space allocated by OPAST can be freed by this call.

**On entry:** HA contains an address within the stack.  
(This means that the current value of the stack pointer can usually be used to free the stack.)

**On exit:** If C = 0 then the stack has been freed.  
If C = 1 then the routine could not free the stack.  
DBAXY preserved

**To call:**  
COP  
EQUB @OPFST% from BASIC  
OPSYS COPFST from MASM

**Example:**

Insert this example in the example given for OPAST.

```
470 TSA          \ Transfer the contents of the stack
                 \ pointer to HA.
480 COP          \ Execute COP call.
490 EQUB @OPFST% \ Free stack pointed to by HA.
```

This example will free the current stack (assuming it was allocated by OPAST). Note however that the stack pointer must then be restored to its old value, see example for OPAST.

# OPCVS convert stack index

**Action:** This call takes a stack pointer relative address and converts it to an absolute BHA address. The offset from the stack pointer is held in HA on entry.

**On entry:** HA contains a 16 bit offset from the stack pointer.

**On exit:** If C = 0 then BHA contains the absolute address (S+HA).  
If C = 1 then the stack offset is invalid.  
DXY preserved

**To call:** COP  
EQUB @OPCVS% from BASIC  
OPSYS £OPCVS from MASM

**Example:**

```
170 COP          \ Execute COP call.  
180 EQUB @OPCVS% \ Convert stack relative address.
```

This example will put the value S+HA into BHA. S+HA should always be in bank zero. If it spills into bank 1 then the address is invalid and the call exits with C set.

## OPCVD convert direct page index

**Action:** This call takes a direct page relative address and converts it to an absolute BHA address. The offset from the direct page register is held in HA on entry..

**On entry:** HA contains a 16 bit offset from the direct page register.

**On exit:** If C = 0 then BHA contains the absolute address (D+HA).  
If C = 1 then the direct page offset is invalid.  
DXY preserved

**To call:** COP  
EQUB @OPCVD% from BASIC  
OPSYS £OPCVD from MASM

**Example:**

```
170 COP          \ Execute COP call.  
180 EQUB @OPCVD%    \ Convert direct page relative address.
```

This example will put the value D+HA into BHA. D+HA should always be in bank zero. If it spills into bank 1 then the address is invalid and the call exits with C set.

# OPCVS convert stack index

**Action:** This call takes a stack pointer relative address and converts it to an absolute BHA address. The offset from the stack pointer is held in HA on entry.

**On entry:** HA contains a 16 bit offset from the stack pointer.

**On exit:** If C = 0 then BHA contains the absolute address (S+HA).  
If C = 1 then the stack offset is invalid.  
DXY preserved

**To call:** COP  
EQUB @OPCVS% from BASIC  
OPSYS £OPCVS from MASM

**Example:**

```
170 COP          \ Execute COP call.  
180 EQUB @OPCVS% \ Convert stack relative address.
```

This example will put the value S+HA into BHA. S+HA should always be in bank zero. If it spills into bank 1 then the address is invalid and the call exits with C set.

## OPCVD convert direct page index

**Action:** This call takes a direct page relative address and converts it to an absolute BHA address. The offset from the direct page register is held in HA on entry..

**On entry:** HA contains a 16 bit offset from the direct page register.

**On exit:** If C = 0 then BHA contains the absolute address (D+HA).  
If C = 1 then the direct page offset is invalid.  
DXY preserved

**To call:** COP  
EQUB @OPCVD% from BASIC  
OPSYS EOPCVD from MASM

**Example:**

170 COP \ Execute COP call.  
180 EQUB @OPCVD% \ Convert direct page relative address.

This example will put the value D+HA into BHA. D+HA should always be in bank zero. If it spills into bank 1 then the address is invalid and the call exits with C set.

# OPBYX BYXD to BHA conversion

- Action:** Converts a BBC-type BYX pointer to a Communicator-type BHA pointer. If Y is non-zero then YX is simply copied to HA. If Y is zero then X is taken as an offset from the direct page register D. BHA is calculated as the absolute value of this direct page address.
- On entry:** BYX points to an address, or Y = 0 and XD points to a direct page address.
- On exit:** BHA points to equivalent address.  
D preserved
- To call:** COP  
EQUB @OPBYX% from BASIC  
OPSYS EOPBYX from MASM

**Example:**

170 PHK	\ Push the high byte of the program counter.
180 PLB	\ Pull this byte into the bank register.
185 BNK P%&10000	\ Tell assembler that B = K.
190 LDY# (label% AND &FF00) DIV &100	\ Initialise Y.
200 LDX# label% AND &FF	\ Initialise X.
210 COP	\ Execute COP call.
220 EQUB @OPBYX%	\ Convert BYX to BHA address.
230 RTL	\ Return from subroutine.
240 label%	\ Label for BYX and BHA to point to.

This example will set BYX to point to *label%*, and then convert the BYX pointer to a BHA pointer.



## 5. Task management and the menu program

The Communicator Operating System provides facilities for the task and event management.

A task is a program that can be run from the system's top-level coroutine - the menu program. A task is started as a single coroutine which may span more related coroutines. Most tasks on Communicator are comprised of only a single coroutine.

In the context of task management, an event or pre-empt event is generated when the user presses one of the pre-empt keys on the Communicator's keyboard. Pressing a pre-empt key signals to Communicator that the user wishes to invoke a new task. There are five pre-empt keys available: PHONE, COMP, CALC, STOP and HELP. A pre-empt event actually occurs when its keycode reaches the head of the keyboard buffer. In addition, when the ESCAPE function is enabled and the user presses the ESCAPE key, an escape event is generated.

When an event is signalled it must be acknowledged using OPAEV (acknowledge event) before it is processed.

### 5.1 Task pre-emption

A task cannot be pre-empted without its consent. A task allows pre-emption either by a call to the MOS function OPPRE (allow pre-emption), by a call to the function OPRDC (read a character from the keyboard), or by an OPRLN call (read a line from the keyboard).

When a task allows itself to be pre-empted, the MOS checks to see if a pre-empt event has occurred. If it has, the MOS suspends the current coroutine, and restores and reinvokes its parent. In the case of most tasks, where only one coroutine is spanned, the menu coroutine is immediately restored. As tasks can span more than one coroutine, the MOS leaves the pre-empt key at the head of the keyboard buffer. This means that the restored coroutine will be suspended as soon as it decides to allow pre-emption, thus ensuring that the pre-empt event is eventually passed up the chain to the top-level coroutine - the Menu. It is the responsibility of the Menu program to remove the pre-empt character from the keyboard buffer.

Note: in order for the system to function correctly, all coroutines within a Task should allow pre-emption at some point.

### 5.2 Event acknowledgement

After a coroutine has called on one of the functions to allow pre-emption it should check the returned status to see if pre-emption actually occurred. If a coroutine detects that an event has occurred it must determine the type of event, (pre-empt key or escape), and acknowledge it via the OPAEV function.

### 5.3 Coroutine context

When a Tasks coroutine is suspended in response to a pre-empt event, it is the responsibility of its parent coroutine to save any of the suspended coroutine's associated context.

A task coroutine's context can include its current screen and its VDU and MOS context.

The MOS provides functions to save and restore the contents of the screen being displayed at a particular moment. The function OPSSC, (save screen), will save the current screen in a 20 Kbyte block of memory and return a handle for future reference. The function OPRSC, (restore screen), will restore the contents of the screen, (specified by the supplied handle), to the display.

Within Communicator system software there exist global MOS and VDU variables specifying various system attributes such as font tables, screen type, function key definitions etc. Where a task coroutine is dependent on the contents of any of these variables, it is essential that its parent save them when the coroutine is suspended and restore them before the coroutine is reinvoked. The MOS provides two functions to automate the saving and restoration of these global variables. OPSCK saves the contents of the global variables and returns a handle for future reference. OPRCX restores the VDU and MOS context

specified by the supplied handle.

## **5.4 The menu module**

The menu program acts as a user interface to the operating system's memory management and coroutine facilities, allowing multiple task creation, invocation, suspension and deletion.

The menu maintains the flow of control between tasks, allowing task switching via the pre-empt keys.

On power up, after initialisation, the MOS looks for the module called "MENU" and creates and invokes a coroutine with its entry at the start of the module. This coroutine is now the top level coroutine within the system task management environment.

The menu's operation can be broken down into two parts, the building of a task table and the management of task activity .

### **5.4.1 Building the task table**

The task table is the list of tasks available to the user from the menu program.

The task table is built up from either a task list specified in the configuration file "RAM:c.main", located in non-volatile RAM, or from the down-loaded file "RAM:c.main" located in ordinary RAM.

A task may be added to the task table AFTER it has been built, via the MAIN pseudo-device-driver, (refer to the device-driver section for more detail).

### **5.4.2 Task management**

After the task table has been built, the menu searches for modules with names corresponding to entries within it. If a module with a task table entry is located and the MHCMEN (include in menu) flag is set in the flags field of its module header, the menu will set up the environment necessary to run the task.

The menu program provides the facilities to switch between tasks that it has created. When a task is pre-empted the menu clears and acts on the pre-emption event that caused its suspension and saves the context of the coroutine. The MOS provides the functions to manage the coroutines but it is the responsibility of the menu to maintain the necessary control routes, memory handles and status details which together define a task.

When the menu receives a pre-empt event it must acknowledge it via the MOS function OPAEV and remove the pre-empt key from the head of the keyboard buffer to prevent any spurious pre-emption when the next task is run.

When the menu invokes a task, it is either started from its entry point (newly selected task), or restarted from the first instruction after the point where it was suspended. Before restarting a pre-empted task, the menu must restore the previously saved screen, VDU and MOS context.

**Note:** System programmers may create their own versions of the Menu module, providing a different user interface or a different method of task management.

## 5.5 An example using the use event and context functions

The following piece of code demonstrates the use and invocation of some of the functions available for handling pre-emption events and context saving and restoration.

The program creates two children "routine1" and "routine2" and then invokes routine1.

Routine1 displays a message and waits for the user to press a key via the function OPRDC. On return from the function, routine2 checks to see if it was pre-empted, (i.e. the key pressed was a pre-empt key). If it was pre-empted, routine2 acknowledges the event via the OPAEV function. Routine2 then loops back to its entry point and starts again.

When control is returned to the parent after routine1 is suspended for the first time, the pre-empt key is removed from the keyboard buffer and the pre-empt event acknowledged. The screen and context of routine1 are saved via the functions OPSSC and OPSCX before routine2 is invoked.

Routine2 displays message and waits for the user to press a key via the function OPRDC. If the key pressed is not a pre-empt key, routine2 loops round and repeats the process. When the user presses a pre-empt key routine2 is suspended and control passed back to the parent.

When control is passed back to parent from routine2, the pre-empt key is cleared from the keyboard buffer and the event acknowledged. The context and screen of routine1 are restored via the functions OPRCX and OPRSC respectively, before it is reinvoked.

When routine1 is pre-empted for the second time, the parent clears the and acknowledges the pre-empt event before exiting back to BASIC.

The code should be run in conjunction with the test harness described in Chapter1.

```

20 costksz% = $200
30 codps% = $100
170 RSP &30
175 WRD &30
180 PHK
185 PLB
190 BNK P%/$1000
195 LDW coentry2% AND &FFFF
200 PEAW costksz%
205 PLD
210 LDX# @COCRE%
215 JSRL @CO%
220 BCC BR1%
225 BRL coent2%
230 :
235 BR1%
240 COP
245 EQUB @OPADP%
250 EQUW codps%
255 BCC BR2%
260 BRL coerr2%
265 :
270 .BR2%
275 TAD
280 LDX# @COENV%
285 JSRL @CO%
290 PHY
295 PLA
300 STA cohandle2%
305 :
310 LDW coentry1% AND &FFFF
315 PEAW costksz%
320 PLD
325 LDX# @COCRE%
330 JSRL @CO%
335 BCC BR3%
340 BRL coerr1%
                                         \ Make sure that processor is in word mode.
                                         \ Tell assembler that processor is in word mode.
                                         \ Coroutine entry is in my program bank.
                                         \
                                         \ Tell assmebler that B = K.
                                         \ Entry Address in BHA.
                                         \ Coroutine stack size in D on entry.
                                         \
                                         \ Reason code for create in X.
                                         \ Call coroutine entry.
                                         \ Carry on if no errors.
                                         \ Error.
                                         \
                                         \
                                         \ Execute COP call.
                                         \ Call the direct page allocator.
                                         \ With DP size as argument.
                                         \ Carry on if no errors.
                                         \ Error.
                                         \
                                         \
                                         \ Get Direct Page in D.
                                         \ Set the coroutine's environment.
                                         \
                                         \ Transfer coroutine2's handle into A...
                                         \
                                         \ ... and save it.
                                         \
                                         \ Entry Address in BHA.
                                         \ Coroutine stack size in D on entry.
                                         \
                                         \ Reason code for create in X.
                                         \ Call coroutine entry.
                                         \ Carry on if no errors.
                                         \ Error.

```

```

345 :
350 .BR3%
355 COP
360 EQUB @OPADP%
365 EQUW copds%
370 BCC BR4%
375 BRL coerr1%
380 :
385 .BR4%
390 TAD
395 LDXW @COENV%
400 JSRL @CC0%
405 PHY
410 PLA
415 STA cohandle1%
420 JSRL @CC0%
425 :
430 COP
435 EQUB @OPRDC%
440 LDA @EVPRE%
445 COP
450 EQUB @OPAEV%
455 COP
460 EQUB @OPSSC%
465 BCC BR5
470 BRL coerr1%
475 .BR5%
480 PHY
485 PLA
490 STA schandle1%
495 COP
500 EQUB @OPSCX%
505 BCC BR6%
510 BRL coerr1%
515 .BR6%
520 PHY
525 PLA
530 STA ctxhandle1%
535 :
540 LDA &0C
545 COP
550 EQUB @OPWRC%
555 LDA cohandle2%
560 PHA
565 PLY
570 JSRL @CC0%
575 :
580 COP
585 EQUB @OPRDC%
590 LDA @EVPRE%
595 COP
600 @EQUB OPAEV%
602 :
605 LDA ctxhandle1%
610 PHA
615 PLY
620 COP
625 EQUB @OPRCX%
630 LDA schandle1%
635 PHA
640 PLY
645 COP
\\.
\\ Execute COP call.
\\ Call the direct page allocator.
\\ With DP size as argument.
\\ Carry on if no errors.
\\ Error.
\\.
\\ Get Direct Page in D.
\\ Set the coroutine's environment.
\\.
\\ Transfer coroutine1's handle into A...
\\.
\\ ... and save it.
\\ Call the coroutine.
\\.
\\ Execute COP call.
\\ Clear the pre-empt key from the keyboard buffer.
\\ Acknowledge the pre-empt EVENT.
\\ Execute COP call ...
\\ ... to send Ack to the MOS.
\\ Execute COP call...
\\ ... to save the screen.
\\ Carry on if no errors.
\\ Error.
\\.
\\ Transfer the saved screen's handle into A...
\\.
\\ ... and store it.
\\ Execute COP call...
\\ ... to save the coroutine's context.
\\ Carry on if no errors.
\\ Error.
\\.
\\ Transfer the saved context's handle to A...
\\.
\\ ... and store it.
\\.
\\ Form Feed to clear the screen.
\\ Invoke COP call ...
\\ ... to output the LF in A to the VDU.
\\ Get coroutine2's handle into A ...
\\ ... and transfer it into Y.
\\.
\\ Call coroutine2%
\\.
\\ Execute COP call...
\\ ... to clear the pre-empt key from kybd buffer.
\\ Ack the pre-empt EVENT.
\\ Execute COP call...
\\ ... to send the ACK to the MOS.
\\.
\\ Get handle of coroutine1's context in A...
\\ ... and transfer it to Y.
\\.
\\ Execute COP call...
\\ ... to restore coroutine1's context.
\\ Get handle of coroutine1's context in A...
\\ ... and transfer it to Y.
\\.
\\ Execute COP call...

```

```

650 EQUB @OPRSC%
655 :
660 LDA cohandle1%
665 PHA
670 PLY
675 JSRL @CCO%
680 :
685 COP
690 EQUB @OPRDC%
695 LDA @EVPRE%
700 COP
705 EQUB @OPAEV%
710 RTL
715 :
720 :
725 .coentry1%
730 COP
735 EQUB @OPWRS%
740 EQUUS "Hello, Im Coroutine1!""
745 EQUB &0A
750 EQUB &0D
755 EQUB &00
760 :
765 COP
770 EQUB @OPRDC%
775 BCC coentry1%
780 LDA @EVPRE%
785 COP
790 EQUB @OPAEV%
795 BRA coentry1%
800 :
805 :
810 .coentry2%
815 COP
820 EQUB @OPWRS%
825 EQUUS "Hello, Im Coroutine2!""
830 EQUB &0A
835 EQUB &0D
840 EQUB &00
845 :
850 COP
855 EQUB @OPRDC%
860 BCC coentry2%
865 LDA @EVPRE%
870 COP
875 EQUB @OPAEV%
880 BRA coentry2%
885 :
890 :
895 .coerr1%
900 BRK
905 EQUB &00
910 EQUUS "Error with coroutine1."
915 EQUB &00
920 :
925 :
930 .coerr2%
935 BRK
940 EQUB &00
945 EQUUS "Error with coroutine2."
950 EQUB &00
955 :

```

\ ... to restore coroutine1's screen.  
 \ ... to clear the pre-empt key from kybd buffer.  
 \ Ack the pre-empt event.  
 \ Execute COP call...  
 \ ... to send the ACK to the MOS.  
 \ Exit back to BASIC.  
 \.  
 \.  
 \ This is the entry point for coroutine1.  
 \ Execute COP call...  
 \ ... to output a message to the screen.  
 \.  
 \.  
 \.  
 \ Execute COP call...  
 \ ... to wait for a character from the keyboard.  
 \ We weren't pre-empted, go round again.  
 \ We were pre-empted so ACK the pre-empt KEY.  
 \ Execute COP call...  
 \ ... to send the ACK to the MOS.  
 \ Go round again.  
 \.  
 \.  
 \ This is the entry point for coroutine2.  
 \ Execute COP call...  
 \ ... to output a message to the screen.  
 \.  
 \.  
 \.  
 \ Execute COP call...  
 \ ... to wait for a character from the keyboard.  
 \ We weren't pre-empted, go round again.  
 \ We were pre-empted so ACK the pre-empt KEY.  
 \ Execute COP call...  
 \ ... to send the ACK to the MOS.  
 \ Go round again.  
 \.  
 \.  
 \ Error message for coroutine1.  
 \.  
 \.  
 \ Error message for coroutine2.  
 \.  
 \.  
 \.

*Chapter 5*

```
960 :  
965 .cohandle1%          \ Storage for coroutine1's handle.  
970 EQUW &0000  
975 .cohandle2%          \ Storage for coroutine2's handle.  
980 EQUW &0000  
985 .scrhandle%          \ Storage for coroutine1's screen handle.  
990 EQUW &0000  
995 .ctxhandle%          \ Storage for coroutine1's context handle.  
999 EQUW &0000
```

Lines 195-300 create coroutine2 and its associated environment.

Lines 310-415 create coroutine1 and its associated environment.

Line 420 invokes coroutine1.

Lines 430-450 clear the pre-empt key from the keyboard buffer and acknowledge the pre-empt event.

Lines 455-530 save coroutine1's screen and context.

Lines 540-550 Clear the screen.

Lines 555-570 Invoke coroutine2.

Lines 580-600 Remove the pre-empt key from the keyboard buffer and acknowledge the pre-empt event.

Lines 605-650 Restore coroutine1's context and screen.

Lines 660-675 Reinvoke coroutine1.

Lines 685-705 Remove the pre-empt key from the keyboard buffer and acknowledge the pre-empt event.

Line 710 Exits back to BASIC.

Line 725 Is the start of coroutine1.

Lines 730-755 Output a message from coroutine1 to the screen.

Lines 765-770 Read a character from the keyboard allowing pre-emption.

Line 775 Checks for a pre-empt and loops back to the start of coroutine1 if none occurred.

Lines 780-795 Acknowledge the pre-empt event and loop back to the start of coroutine1.

Line 810 Start of coroutine2.

Lines 815-840 Output a message from coroutine2 to the screen.

Lines 850-855 Read a character from the keyboard allowing pre-emption.

Line 860 Checks for a pre-empt and loops back to the start of coroutine2 if none occurred.

Lines 865-880 Acknowledge the pre-empt event and loop back to the start of coroutine2.

## 5.6 BRK handling

A convention adopted by languages which provide their own BRK handler (see section on Coroutines) is to use the BRK instruction to cause an error condition. The format of an error, using BASIC assembler conventions to illustrate, is:

```
BRK  
EQUB ERRNO%  
EQUS "Error message"  
EQUB &00
```

That is, the BRK instruction is followed by a one-byte error number (with zero meaning "fatal error"), followed by a textual message which is terminated by a zero byte.

When the BRK is executed, the OS passes control, through the appropriate indirections, to the BRK handler of currently active coroutine, which should deal with the error as it sees fit.

Two MOS calls are provided, STSBK to read the error number (the "signature") and STTBK to read the message, one character at a time.

## **5.7 Task management and context associated functions**

The functions associated with task management and context saving and restoration are invoked via COP calls.

The following functions are available:

<b>OPAEV</b>	Acknowledge event
<b>OPPRE</b>	Allow pre-emption
<b>OPRDC</b>	Read character from the keyboard
<b>OPRLN</b>	Read line from the keyboard
<b>OPSSC</b>	Save screen
<b>OPSCX</b>	Save context of VDU and MOS
<b>OPRCX</b>	Restore context of VDU and MOS
<b>OPRSC</b>	Restore screen
<b>OPFCX</b>	Free context of VDU and MOS (including fonts)
<b>OPFSC</b>	Free Screen
<b>STSBK</b>	Read the break signature
<b>STSBK</b>	Read BRK message text

# OPAEV acknowledge event

- Action:** This call acknowledges the occurrence of an event.
- On entry:** HA contains the event mask which specifies the events to be acknowledged. Possible event masks are @EVEESC% (EVEESC in MASM) to acknowledge the ESCAPE event, and @EVPRE% (EVPRE in MASM) to acknowledge the pre-empt event.
- On exit:** If C = 0 then the specified event is acknowledged.  
If C = 1 then an attempt was made to clear an event which had not previously been set using OPSEV. DBAXY preserved
- To call:**  
COP  
EQUB @OPAEV% from BASIC  
OPSYS LOPAEV from MASM
- Example:** see example for OPRDC.

# OPPRE allow pre-emption

**Action:** This call is used to allow pre-emption of the current task at a convenient point. The call allows the MOS to check whether the user has selected another task. If another task has been requested then the call returns with the carry set, and the event must be acknowledged using OPAEV.

**On entry:** No requirements.

**On exit:** If C = 0 then the module was not pre-empted.

If C = 1 then the module was pre-empted.

DBHAXY preserved

**To call:**

COP

EQUB @OPPRE% from BASIC

OPSYS COPPRE from MASM

**Example:**

```

170 COP          \ Execute COP call.
180 EQUB @OPPRE% \ Allow pre-emption.
190 BCS pre-empt% \ If pre-emption requested then branch.
200 RTL          \ If not then carry on with current task.
210 .pre-empt%   \ Event handler.
220 RSP# &30     \ Put processor in 16-bit mode.
230 WRD &30     \ Tell assembler that processor is in 16-bit mode.
240 LDA# @EVPRE% \ Load HA with event mask.
250 COP          \ Execute COP call.
260 EQUB @OPAEV% \ Acknowledge pre-empt.
270 :            \ Continue with pre-empt handling.

```

This example will check to see if the current task has been pre-empted. If it has not then the current task is continued. If it has then the event is acknowledged.

# OPRDC read character from the keyboard

- Action:** Read a character from the keyboard into A. If the keyboard input buffer is empty then the call will wait for input. During this call the current task may be pre-empted, in which case the carry flag will indicate as much. If either ESCAPE is pressed or a pre-empt occurs then the next call must be to OPAEV to acknowledge the event.
- On entry:** No conditions
- On exit:** If C = 0 then A contains the ASCII value of the character.  
If C = 1 then  
if HA = @SCESC% (ESCESC in MASM) then the ESCAPE key was pressed  
or if HA = @SCPREG% (LSCPREG in MASM) then the current task was pre-empted.  
DBXY preserved
- To call:** COP  
EQUB @OPRDC% from BASIC  
OPSYS EOPRDC from MASM

**Example:**

```
170 .read%
180 COP
190 EQUB @OPRDC%
195 BCS event%
200 COP
210 EQUB @OPWRC%
220 BCC read%
230 BRA end%
240 .event%
250 RSP# &30
255 WRD &30
260 CMP# @SCESC%
270 BEQ escape%
280 CMP# @SCPREG%
290 BNE end%
300 LDA# @EVPRE%
310 COP
320 EQUB @OPAEV%
330 BRA end%
340 .escape%
350 LDA# @EVEESC%
360 COP
370 EQUB @OPAEV%
380 .end%
```

\ Set label called *read%*.
 \ Execute COP call.
 \ Read a character into A.
 \ If C = 1 then go to the event handler.
 \ Execute COP call.
 \ Send the character in A to the VDU drivers.
 \ Branch if carry clear to the label *read%*.
 \ Go to end.
 \ Event handler routine.
 \ Put processor in word mode.
 \ Tell assembler that processor is in 16-bit mode.
 \ See if HA indicates ESCAPE pressed.
 \ If it does then branch.
 \ See if A indicates pre-empt.
 \ If it doesn't then end.
 \ Load HA with event mask.
 \ Execute COP call.
 \ Acknowledge pre-empt.
 \ Go to end.
 \ Escape routine.
 \ load HA with event mask.
 \ Execute COP call.
 \ Acknowledge ESCAPE.
 \ End here.

This example will read a character from the keyboard and send it to the VDU drivers. It will repeat this process until either ESCAPE is pressed, or the task is pre-empted.

# OPRLN

read line from keyboard

- Action:** This call will read a line of characters from the keyboard. The call uses a control block which gives the address at which the characters are to be put, the length of this buffer, and the number of characters to be sent to the VDU drivers. This number can be zero in which case no characters are printed. During this call the current task may be pre-empted, in which case the carry flag will indicate as much. If either ESCAPE is pressed or a pre-empt occurs then the next call must be to OPAEV to acknowledge the event.
- On entry:** BHA points to a control block somewhere in memory. The format of the control block is as follows.
- +0 4 byte pointer to the keyboard buffer. The least significant byte first (0) and the most significant byte last (3). The most significant byte must be zero (&00).
  - +4 2 byte buffer size. The buffer can be any size up to 65535 characters.
  - +6 2 byte prefix length. This is the number of characters in the buffer which will be sent to the VDU drivers. Therefore it must not be greater than the buffer size. If the prefix length is zero then no characters are sent to the VDU drivers.
  - +8 4 byte zero word for future expansion.
- On exit:** If C = 0 then the line has been read and the characters are in the buffer.  
A contains the terminator of the input string (CR ASCII 13 decimal).  
The length of the string in the buffer is given by the 2 bytes at offset +6 in the control block.  
If C = 1 then  
if HA = @SCESC% (ESCESC in MASM) then the ESCAPE key was pressed or  
if HA = @SCPRES% (ESCPRE in MASM) then the current task was pre-empted.  
2 byte length at offset +6.  
DBXY preserved  
Offsets 0 to 5 remain the same.
- To call:**
- COP
  - EQUB @OPRLN% from BASIC
  - OPSYS £OPRLN from MASM

**Example:**

```

30 DIM buffer% 256 :REM allocate space for buffer
40 size%<=255
50 prefix%<0
170 BRA readline%
180 .conblock%
190 EQUD buffer%
200 EQUW size%
210 EQUW prefix%
220 EQUD &00000000
230 .readline%
240 RSPW &30
250 WRD &30
260 PHK
270 PLB
280 BNK P%&10000
290 LDA# conblock% AND &FFFF
300 COP
310 EQUB @OPRLN%
320 BCC end%
330 :
340 CMP# @SCESC%
350 BEQ escape%
360 CMP# @SCPREG%
370 BNE end%
380 LDA# @EVPRE%
390 COP
400 EQUB @OPAEV%
410 BRA end%
420 .escape%
430 LDA# @EVESC%
440 COP
450 EQUB @OPAEV%
460 .end%

```

\ Size of buffer.  
 \ No characters to VDU drivers.  
 \ Branch to set up BHA and execute  
 \ the COP call.  
 \ Set up label for control block.  
 \ Set the 4 byte buffer address.  
 \ Set the 2 byte buffer size.  
 \ Set the 2 byte prefix length.  
 \ Set 4 bytes to zero (&00).  
 \ Label to which to branch.  
 \ Put processor in word mode.  
 \ Tell assembler that processor is in 16-bit mode.  
 \ Push the high byte of the program  
 \ counter on the stack.  
 \ Pull the high byte into B.  
 \ Tell assembler that B = K.  
 \ Load HA with address of label.  
 \ BHA now points to the control block.  
 \ Execute COP call.  
 \ Read a line from the keyboard  
 \ into the buffer.  
 \ If no event then end.  
 \ Event handler routine.  
 \ See if A indicates ESCAPE pressed.  
 \ If it does then branch.  
 \ See if A indicates pre-empt.  
 \ If it doesn't then end.  
 \ Load HA with event mask.  
 \ Execute COP call.  
 \ Acknowledge pre-empt.  
 \ Go to end.  
 \ Escape routine.  
 \ load HA with event mask.  
 \ Execute COP call.  
 \ Acknowledge ESCAPE.  
 \ End here.

This example will read a line of input from the keyboard into a buffer. The address of the buffer is given by *buffer%*. The size of the buffer is given by *size%*. The number of characters to be sent to the VDU drivers is given by *prefix%*.

# OPSSC

save screen

**Action:** Used by the menu program (and others) to save the contents of the screen display.

**On entry:** Must be in 16 bit XY mode.

**On exit:** If C = 0 then the call succeeded and Y contains the screen pool handle.  
If C = 1 then the call failed.  
D preserved

**To call:** COP  
EQUB @OPSSC% from BASIC  
OPSYS £OPSSC from MASM

**Example:**

```
170 RSP# &10      \ Put processor in 16-bit XY mode.
172 WRD &10      \ Tell assembler that processor is in 16-bit XY mode.
175 COP          \ Execute COP call.
180 EQUB @OPSSC%  \ Save screen.
190 BCC next%    \ If call succeeded then continue.
200 .end%        \ End here if failed to save screen.
210 RTL          \ Return from subroutine.
220 .next%       \ Here if screen saved.
230 PHY          \ Push the pool handle returned by Y
                  \ on the stack for use when restoring or freeing the screen.
```

Note: this program requires the example given for OPRSC before it will work.

This example will attempt to save the screen for use when the current task is resumed. If the call succeeds then the pool handle returned in Y will be pushed on the stack for use with the OPRSC call at a later date.

**This COP must be called in 16-bit XY mode (handles are 16 bits).**

# OPSCX save context of VDU and MOS

**Action:** Used by the menu program (and others) to save the context of the screen and the operating system variables when switching to a different task. It must be used only after an OPSSC call.

**On entry:** No requirements.

**On exit:** If C = 0 then the call succeeded and Y contains the context pool handle.  
If C = 1 then the call failed.  
D preserved

**To call:** COP  
EQUB @OPSCX% from BASIC  
OPSYS £OPSCX from MASM

**Example:**

Add this example to the end of the example given for OPSSC.

```
240 COP          \ Execute COP call.
250 EQUB @OPSCX% \ Save context of VDU and MOS.
260 BCS opsc%    \ Couldn't save context so restore screen and end.
270 PHY          \ Push the pool handle returned by Y
                  \ on the stack for use when restoring or freeing the context.
```

Note: this program requires the examples given for OPSSC, OPRCX and OPRSC before it will work.

This example will attempt to save the context of the VDU and MOS for use when the current task is resumed. If the call succeeds then the pool handle returned in Y will be pushed on the stack for use with the OPRCX call at a later date.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPRCX restore context of VDU and MOS

**Action:** Used by the menu program (and others) to restore a context previously saved using OPSCX. It must be used only before an OPRSC call.

**On entry:** Y = context pool handle allocated by OPSCX. Must be in 16 bit XY mode.

**On exit:** If C = 0 then the context is restored and the pool freed.  
If C = 1 then the call failed.  
D preserved

**To call:** COP  
EQUB @OPRCX% from BASIC  
OPSYS £OPRCX from MASM

## Example:

Add this example to the end of the example given for OPSCX.

```
280 PLY          \ Pull the context pool handle from the stack.  
290 COP          \ Execute COP call.  
300 EQUB @OPRCX% \ Restore context.  
310 BCS oprsc%  \ Couldn't restore context so tidy stack  
                  \ by attempting to restore screen.
```

Note: this program requires the examples given for OPSSC, OPSCX and OPRSC before it will work.

This example will attempt to restore a previously saved context whose pool handle is in Y.

This COP must be called in 16-bit XY mode (handles are 16 bits).

## OPRSC restore screen

**Action:** Used by the menu program (and others) to restore the contents of a screen previously saved using OPSSC.

**On entry:** Y = screen pool handle allocated by OPSSC.

**On exit:** If C = 0 then the screen is restored and the pool freed.  
If C = 1 then the call failed.  
D preserved

**To call:** COP  
EQUB @OPRSC% from BASIC  
OPSYS £OPRSC from MASM

**Example:**

Add this example to the end of the example given for OPRCX.

```
320 .oprsc%          \ Label to which to branch to restore screen.  
330 PLY              \ Pull the screen pool handle from the stack.  
340 COP              \ Execute COP call.  
350 EQUB @OPRSC%    \ Restore screen.  
                      \ If C set on exit then call failed.
```

Note: this program requires the examples given for OPSSC, OPSCX and OPRCX before it will work.

This example will attempt to restore a previously saved screen whose pool handle is in Y.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPFCX free context of VDU and MOS (including fonts)

**Action:** Frees a saved context that is no longer required.

**On entry:** Y = context pool handle allocated by OPSCX.

**On exit:** If C = 0 then the context is freed and the pool handle freed.  
If C = 1 then the call failed to free the context.  
D preserved

**To call:**  
COP  
EQUB @OPFCX% from BASIC  
OPSYS £OPFCX from MASM

**Example:**

Add this example to the end of the example given for OPSCX.

```
280 PLY          \ Pull the context pool handle from the stack.  
290 COP          \ Execute COP call.  
300 EQUB @OPFCX% \ Restore context.  
310 BCS opfsc%  \ Couldn't free context so tidy stack  
                  \ by attempting to free screen.
```

Note: this program requires the examples given for OPSSC, OPSCX and OPFSC before it will work.

This example will attempt to free the context whose pool handle is in Y.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPFSC free screen

**Action:** Frees a saved screen which is no longer required.

**On entry:** Y = screen pool handle allocated by OPSSC.

**On exit:** If C = 0 then the screen is freed and the pool handle freed.  
If C = 1 then the call failed.  
D preserved

**To call:** COP  
EQUB @OPFSC% from BASIC  
OPSYS EOPFSC from MASM

**Example:**

Add this example to the end of the example given for OPFCX.

```
320 .opfsc%          \ Label to which to branch to free screen.  
330 PLY             \ Pull the screen pool handle from the stack.  
340 COP             \ Execute COP call.  
350 EQUB @OPFSC%    \ Free screen.  
                      \ If C set on exit then call failed.
```

Note: this program requires the examples given for OPSSC, OPSCX and OPFCX before it will work.

This example will attempt to free a saved screen whose pool handle is in Y.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# STSBK    Read the BRK signature

**Action:** This call returns the error number associated with the last BRK executed.

**On Entry:** X = STSBK

**Exit:** A = Error number of the last error

HY = State to be used in subsequent calls

B,D Preserved

Note that HY must be preserved between the STSBK call and subsequent calls to STTBK (see below).

## **STTBK** Read BRK message text

**Action** This call returns the next character in the error message associated with the last BRK executed.

**Entry:** HY = State returned by last STSBK or STTBK  
X = STTBK

**Exit:** B,D Preserved  
C=0 Then next byte was read  
A = Next byte in BRK message text  
HY = State for next STTBK  
or C=1 Last character has been read

## **6. Device drivers**

### **6.1 Introduction to device drivers**

Device control is effected by device driver modules. These modules conform to a standard device driver interface which defines closely how information may be transferred to and from the device. This interface, although designed initially for the control of hardware devices, also provides a useful general mechanism for communication between modules.

Devices are modelled as streams. Each stream has a name (for example the serial stream is "RS423", the modem is "MODEM"). In order to use a stream, it must first be opened, to establish a connection and to allow the device driver to initialise its resources. Once open, the stream allows data and control information to be passed to and/or from the device. After use, it should be closed to free the stream for use by other programs.

## 6.2 Simple device control from BASIC

This section introduces the principles of controlling device drivers from application programs, with programming examples in BASIC.

### 6.2.1 Opening a device driver

To open a device, the BASIC keywords OPENIN, OPENOUT or OPENUP are used. These operations exchange the name of the device for a handle which is used for subsequent access to the device. The handle is a 16-bit quantity, which is defined to be non-zero if the open succeeded. Thus, to open a device you might do the following:

```
10 data_handle% = OPENUP "RS423":REM open the RS423 driver  
20 IF data_handle% = 0 THEN PRINT $@ERMSG%:END :REM check OPEN worked
```

OPENUP should be used whenever you want 2-way data communication with the device; use OPENOUT if data output only is required (eg for printer channels), and OPENIN if you require read-only access to the device.

If the OPEN fails for any reason, a zero handle will be returned to BASIC, and @ERMSG% will point to an error message.

### 6.2.2 Sending and receiving data

Opening a device establishes a data channel which may be accessed using BPUT#, BGET# and GETS# to read and write byte or string values. For device drivers controlling i/o ports (eg RS423 or MODEM), this channel is used by the application program to send data for transmission, and to read received data from the device driver. For example, to read data received at the RS423 port, and to output data to the port, the following BASIC lines might follow on from the previous example:

```
30 in_byte% = BGET# data_handle% :REM read a byte  
40 in_string$ = GETS# data_handle% :REM read an ASCII string  
50 BPUT# data_handle% , 42 :REM output an explicit byte value  
60 BPUT# data_handle% , out_byte% :REM output a byte stored as an integer variable  
70 BPUT# data_handle% , "Hello world" :REM output a string
```

The data channel is fully transparent; that is, any desired byte value in the range 0 to 255 can be transmitted or received. GETS# must therefore be used with care, as this keyword is intended only to read character strings.

The behaviour of device drivers in case of errors (eg full or empty buffers) on the data channel is device-dependent, and in some cases, programmable. See the individual device descriptions below for full details. If any errors are returned to BASIC, this will be done by the usual @ERC% mechanism; if in doubt, always check that @ERC% is zero after each read or write operation.

### 6.2.3 Sending commands to a device driver

In addition to the data channel, all device drivers support a control channel which may be used to send commands to the driver. Such commands are used to control the future behaviour of the device driver, or to achieve special effects. For ease of programming, all information flow in this channel is in the form of readable ASCII strings.

The control channel is accessed from BASIC using the same keywords (BGET#, GETS# and BPUT#) as are used for the data channel; the two channels are distinguished by their handles. The handle returned on opening a device is always an even number, and is used to access the data channel; to access the control channel, handle + 1 must be used. For each BPUT or BGET call, the operating system identifies which channel is required by testing whether the handle supplied is odd or even.

The following program illustrates the use of commands, and continues from the previous examples:

```
80 control_handle% = data_handle% + 1  
90 BPUT# control_handle% , "D1200" :REM select 1200 Baud data rate  
100 BPUT# control_handle% , "F1" :REM flush the output buffer
```

Commands sent down the control channel usually consist of an upper-case letter, which specifies the command, followed by an ASCII string containing a parameter to the command; only a single parameter is allowed. For example, in line 90 above, the "D" command sets the baud rate for the RS423 driver, and

"1200" is a command parameter representing the required rate.

The end of a command must be marked by a suitable terminator; any control character will act as a terminator, but it is conventional to use a carriage return (ASCII code 13 decimal). This is supplied automatically by BASIC's string handlers in the examples above.

The commands available are device-dependent, and are described in detail below. Where relevant, compatibility of commands between drivers has been maintained. Note that all commands and their parameters are case-sensitive: for example, in line 100 above, only "Fix" will work; "fix" and "FTX" would both fail.

Five main types of parameter are supported by device-driver commands. These are all illustrated in the following (somewhat contrived) section of a BASIC program. This example assumes that drivers have already been opened, and that modem% is the control handle for the modem driver, and printer% is the control handle for the printer driver.

```

10          :REM The printer F command takes no parameter
20 BPUT# printer% , "F"
30          :REM flush the printer buffer
40          :REM The modem B command takes a
41          :REM decimal number in the range 0 to 65535
50 BPUT# modem% , "B25"
60          :REM send a break signal of duration 25 units
70          :REM in the range 0 to 65535
80          :REM The modem I command takes a
81          :REM hexadecimal number in the range 0 to FFFFFF
90 BPUT# modem% , "I0"
100         :REM set Input buffer size to &C0 bytes
110         :REM The modem D command takes a parameter which is one
120         :REM of a limited set of options predefined for this command
130 BPUT# modem% , "D300"
140          :REM Other valid parameters are "1200/75" or "75/1200"
150          :REM The modem C command takes a parameter which is
160          :REM an arbitrary ASCII string of up to 30 characters
170 BPUT# modem% , "CTE0223/12345" :REM dial a specified string

```

In the examples above, the limitations specified for decimal, hexadecimal and string parameters are the maximum range which can be handled by the operating system. These limits may be further constrained by the individual commands. For example, any arbitrary ASCII string may be passed by the operating system to the modem C command, but the command would reject strings containing characters which are invalid within dial strings. Where such constraints exist, they are described under the individual driver commands below.

See the descriptions of individual device drivers for details of the commands available.

#### 6.2.4 Obtaining status information from a device driver

In addition to sending commands, the control channel may be used to read status information from a device driver. This is done using the "?" command, as follows:

```

110 BPUT# control_handle% , "?W" :REM prompt for number of data bits
120 IF GET$# control_handle% = "W5" THEN PRINT "Invalid data format":CLOSE# data_handle%:STOP
130 REM read back number of data bits, and shut down if it's invalid

```

The "?" command is available in all current device drivers. It takes a parameter which is generally a valid command letter for the device driver. Thus "?W" in line 110 means "What is the current setting of your W command?". Assuming status information is available for the W command, the device driver will respond by writing "W", followed by the current setting, to the status buffer, where it may be read as shown in line 120.

Other, more specialised uses of the "?" command are described below for some device drivers.

#### 6.2.5 Closing a device driver

When access to a device is no longer required by an application program, the program should close the stream. This allows the device driver to free its resources and hardware, so that they are available for use by other tasks. From BASIC, use the CLOSE statement, thus:

```
999 CLOSE# data_handle% :REM shut down device driver
```

### **6.3 Default options and the Configure program**

The Configure program (described in the User guide) provides a user-friendly way to create files in non-volatile CMOS memory which contain sets of commands for certain device drivers. These files act as default options whenever device drivers are opened.

When a device driver is OPENed, the operating system will search for its Configure file, and if found, it will pass the commands in this file to the device's control channel. This occurs before the handle is returned to the application program, and allows the user to tailor the behaviour of the devices at run-time.

These default settings are used only at OPEN time; they may be overridden at any time thereafter by new commands issued by the application program. Applications programmers should therefore consider carefully whether to include explicit device driver commands within the program, or to allow the user's defaults to prevail. For example, if a program must communicate with external hardware via the RS423 port, it may be considered desirable to set the input and output buffer sizes explicitly according to the needs of the application, but to leave the data rate and format commands unspecified, so that the application can be easily tailored to suit different external hardware at run time.

## 6.4 Device driver "filenames"

The string passed when opening a driver is of the following form:

"<module name>:<file specifier> <command sequence>"

(Note space between file specifier and command sequence.)

where

<module name>	is the name of the module containing the driver
<file specifier>	is a qualifier whose interpretation is driver dependent. For example, if the driver is a filing system, the file specifier is the filename.
<command sequence>	If the first character of the specifier is "~", the driver is opened without initialising any hardware. This type of open is provided for the Configure program, so that the list of commands and parameters can be read without allocating any hardware.  is a sequence of control channel commands to be sent to the driver after opening. Any ";" characters in this string will be substituted by carriage returns.

As an example, the following might open a stream to the modem, using phone line 2, with 400 bytes allocated as the input buffer, and using the xon/xoff protocol:

```
data_handle% = OPENUP "MODEM: W7;400;Hxon/xoff;Podd"
```

(Note space after colon.)

This will open the MODEM driver with xon/xoff protocol, 7 data bits, and odd parity.

When explicit commands must be sent to a driver at open time, it is preferable to include them in the open string in this way, as this allows the driver to check that the commands have executed successfully before proceeding with the OPEN. For i/o port drivers, it also ensures that all required options are set before data reception is enabled from the port. This avoids any possible problems which might arise from spurious data reception before correct data formats and communications protocols have been set.

## **6.5 Device control from assembler programs**

Device control from assembler programs should be implemented using operating system COP calls.

OPOPN and OPCLS are used to open and close drivers, while OPBPT and OPBGT perform the BGET and BPUT functions for both data and control channels. Note that OPBPT and OPBGT only support single byte operations; if string handling required, it must be performed explicitly, or else the programmer must resort to BASIC.

Certain device-drivers (eg Keypage) support calls with non-standard reason codes (ie in addition to OPEN, CLOSE, and data and control BGET and BPUT). Such calls can only be made from assembler programs, and should use the OPCUH operating system COP call.

Error handling differs slightly from BASIC programs. All the above COP calls indicate errors by returning with the carry flag set - do not rely on OPOPN returning a zero handle from failed OPENs. In addition, X will indicate an error code, and if bit 0 of X is set, BHA will point to an error message.

These COP calls are described in detail in the following pages, using examples written for the BASIC Assembler. For completeness, the calls OPAH, OFFH and OPVH are also described; these are used internally within device drivers, and should not normally be needed by applications programmers.

# OPOPN

open device driver

**Action:** This call opens a device driver.

**On entry:** BHA points to the name (with colon), terminated by CR (&0D).  
 Y = &40 for input  
 Y = &80 for output  
 Y = &C0 for update

**On exit:** C = 0 means that the device driver is open and Y = handle.  
 If C = 1 then the device failed to open, the error code is in X, and BHA points to a zero-terminated error message.  
 D preserved

**To call:** COP  
 EQUB @OPOPN% from BASIC  
 OPSYS EOPOPN from MASM

**Example:**

170 RSP# &10	\ Put processor in 16-bit XY mode.
175 WRD &10	\ Tell assembler that processor is in 16-bit XY mode.
180 COP	\ Execute COP call.
190 EQUB @OPBHA%	\ Make BHA point to following string.
200 EQUS "MODEM"	\ Driver name here, including colon.
210 EQUB &00	\ Terminated by zero.
220 LDY# &C0	\ Open type in Y (OPENUP).
230 COP	\ Execute COP call.
240 EQUB @OPOP <small>N</small> %	\ Open driver.
250 BCS end%	\ Go to end if failed.
260 :	\ Driver open, handle in Y.
270 PHV	\ Save handle.
280 :	\ Rest of code here, including OPCLS.
900 end%	\ End here.

Note: In order to run, this example needs to be completed by the addition of the example given for OPCLS on the next page.

This example will attempt to open the MODEM for input and output.

This COP must be called in 16-bit XY mode (handles are 16 bits).

## OPCLS close device driver

**Action:** This call closes a device driver which has been opened using OPOPN.

**On entry:** Y = handle allocated by OPOPN.

**On exit:** If C = 0 then driver is closed.

If C = 1 then the driver close failed, X = error code and BHA points to a zero-terminated error message.

D preserved

**To call:**

COP  
EQUB @OPCLS% from BASIC  
OPSYS COPCLS from MASM

**Example:**

Add this example to the end of the previous example given for OPOPN.

870 PLY	\Get handle from stack.
880 COP	\Execute COP call.
890 EQUB @OPCLS%	\Close device driver. \Carry set on exit if call failed.

Note: this example will not run without the addition of the example given for OPOPN.

This COP must be called in 16-bit XY mode (handles are 16 bits).

This example will attempt to close the MODEM driver opened in the previous example.

# OPBGT BBC OSBGET

**Action:** Read one byte from an open device or file. The channel must previously have been opened using OPOPN.

**On entry:** Y contains the handle from OPOPN.

**On exit:** A contains the byte read from the file.  
If C = 1 then an error has occurred, and A is invalid.  
No registers preserved

**To call:** COP  
EQUB @OPBGT% from BASIC  
OPSYS EOPBGT from MASM

**Example:**

170 RSP# &30	\ Put processor in 16-bit mode.
172 WRD &30	\ Tell assembler that processor is in word mode.
175 PHK	\ Push the current bank on to the stack.
180 PLB	\ Pull this value into the bank register.
185 BNK P%&10000	\ Tell assembler that B = K.
190 LDA# filename%	\ Load HA with the middle and low bytes of the label <i>filename</i> %.
191 AND &FFFF	\ BHA now points to the label <i>filename</i> %.
210 LDY# &40	\ Load Y with the open type (OPENIN).
220 COP	\ Execute COP call.
230 EQUB @OPOP%N	\ Call OPEN routine.
240 BCS end%	\ If file not opened then end.
260 COP	\ Execute COP call.
270 EQUB @OPBGT%	\ Read byte from file into A.
280 .end%	\ Set label for end.
290 RTL	\ Return from subroutine.
300 .filename%	\ Set label <i>filename</i> %.
310 EQUUS "RS423: "	\ Insert device name string.
320 EQUB &0D	\ name must be terminated by CR (&0D).

This example will open the RS423 device driver for input, and then read the first byte from that device into A.

**This COP must be called in 16-bit XY mode (handles are 16 bits).**

# OPBPT BBC OSBPUT

**Action:** Write one byte to an open file or device. The channel must previously have been opened using OPOPN.

**On entry:** Y contains the file handle from OPOPN.  
A contains the byte to be written.

**On exit:** No registers preserved

**To call:** COP  
EQUB @OPBPT% from BASIC  
OPSYS £OPBPT from MASM

**Example:**

170 RSPW &30	\ Put processor in 16-bit mode.
172 WRD &30	\ Tell assembler that processor is in word mode.
175 PHK	\ Push the current bank on to the stack.
180 PLB	\ Pull this value into the bank register.
185 BNX P%/&10000	\ Tell assembler that B = K.
190 LDA# filename% AND &FFFF	\ Load HA with the middle and low byte of the label <i>filename%</i> . \ BHA now points to the label <i>filename%</i> .
210 LDY# &C0	\ Load Y with the open type (OPENUP).
220 COP	\ Execute COP call.
230 EQUB @OPOPN%	\ Call OPEN routine.
240 BC\$ end%	\ If file not opened then end.
260 LDA# byte%	\ Load A with byte to be sent to file.
270 COP	\ Execute COP call.
280 EQUB @OPBPT%	\ Write byte from A into file.
290 .end%	\ Set label for end.
300 RTL	\ Return from subroutine.
310 .filename%	\ Set label <i>filename%</i> .
320 EQUIS "MODEM:"	\ Insert device name string.
330 EQUB &0D	\ name must be terminated by CR (&0D).

This example will write the byte contained in *byte%* to the Modem.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPAH allocate 16 bit handle

**Action:** This call allocates a 16 bit handle to the device whose name is pointed to by BHA. The handle is returned in Y.

**On entry:** BHA points to the device name. Device drivers are called with a colon appended to the module name to distinguish them from files. For example, to allocate a handle for the driver module MODEM, BHA must point to MODEM:  
Y = 0.

**On exit:** If C = 0 then handle was allocated and Y = handle.  
If C = 1 then the call failed to allocate a handle.  
No registers preserved

**To call:** COP  
EQUB @OPAH% from BASIC  
OPSYS \$OPAH from MASM

**Example:**

170 RSPW &30	\ Put processor in 16-bit mode.
175 WRD &30	\ Tell assembler that processor is in 16-bit mode.
180 PHK	\ Push the high byte of the program counter
	\ on to the stack.
190 PLB	\ Pull this byte into the bank register.
195 BNK P%&10000	\ Tell assembler that B = K.
200 LDA# name% AND &FFFF	\ Load HA with the 2 bytes of the
	\ address of the label <i>name%</i> .
	\ BHA now points to the label <i>name%</i> .
210 LDY# &00	\ Initialise Y.
220 COP	\ Execute COP call.
230 EQUB @OPAH%	\ Allocate handle.
240 PHY	\ Save handle on stack.
250 BRA continue%	\ Continue after data block.
260 .name%	\ Set a label <i>name%</i> .
270 EQUIS "NET"	\ Put the device name in memory.
280 EQUIS ":"	\ Append colon.
290 EQUIS "filename"	\ Optional filename can be used with NET: driver.
300 EQUB &00	\ Terminate with zero (&00).
310 .continue%	\ Continue from here.
320 BCC next%	\ If handle allocated then branch.
330 RTL	\ Return from subroutine.
340 .next%	\ See OPCUH for continuation.

Note: this example will not run without the addition of the examples given for OPCUH and OPFH.

This example will allocate a handle to the file called *filename* on the NET driver.

This COP must be called in 16-bit XY mode (handles are 16 bits).

## OPCUH call device driver using handle

**Action:** Calls the device driver whose handle was allocated by OPAH.

**On entry:** Y = handle.

Other registers and/or data as required by device driver.

**On exit:** HAXYB registers as returned from device driver.

If C = 0 then the codes returned from driver are valid.

If C = 1 then either the handle was invalid, or the codes returned by the driver are invalid.

D preserved

**To call:**

COP

EQUB @OPCUH% from BASIC

OPSYS £OPCUH from MASM

**Example:**

Add this example to the end of the example given for OPAH.

350 COP

\ Execute COP call.

360 EQUB @OPCUH%

\ Call device driver whose handle is in Y.

Note: this example will not run without the addition of the examples given for OPAH and OPFH.

This example will call the device driver whose handle was allocated by OPAH.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPFH free handle

**Action:** This call frees the handle allocated by OPAH.

**On entry:** Y = handle.

**On exit:** If C = 0 then the handle is freed.  
If C = 1 then the call failed to free the handle.  
No registers preserved

**To call:** COP  
EQUB @OPFH% from BASIC  
OPSYS £OPFH from MASM

## Example:

Add this example to the end of the example given for OPCUH.

```
370 PLY          \ Get handle from stack.  
380 COP          \ Execute COP call.  
390 EQUB @OPFH%  \ Free handle.
```

Note: this example will not run without the addition of the examples given for OPAH and OPCUH.

This example will free the handle allocated in the example given for OPAH.

This COP must be called in 16-bit XY mode (handles are 16 bits).

## OPVH validate handle

**Action:** Checks whether a handle is correct for the given device name.

**On entry:** BHA points to the zero-terminated device name.  
Y = handle

**On exit:** If C = 0 then handle valid.  
If C = 1 then handle not valid.  
DY preserved

**To call:**  
COP  
EQUB @OPVH% from BASIC  
OPSYS £OPVH from MASM

## **6.6 Low-level device driver reason codes**

At the module level, operating system driver COP calls and BASIC device driver function map down to device driver reason codes.

The device driver reason codes define the standard device driver interface. Most drivers in the system only implement a subset of the device driver interface. In addition, many device drivers implement non-standard reason codes in order to accelerate functionality.

Standard device driver reason codes can be divided into two groups.

The first group includes basic device driver functions such as opening and closing devices, sending a byte to a device etc. Parameters are passed to and returned from these functions via registers.

The second group of extended device driver reason codes have associated control blocks for transferring data and control information between the application and the driver. These functions are, in the main, associated with file system drivers. Most device drivers do not implement these reason codes.

If necessary an application can use device driver reason codes to access the device drivers at the module level, thus reducing some operating system call overheads. It is NOT recommended that device drivers are accessed in this way.

### **6.6.1 Basic device driver reason codes**

The following is a list of the standard device driver reason codes:

DVRST	reset the device driver
DVOPN	open the device driver
DVCLS	close the device driver
DVBGT	get a byte from the device driver
DVBPT	put a byte to the device driver
DVCGT	get a control byte from the device driver
DVCPT	put a control byte to the device driver
DVEOF	return EOF status for the device/file

### **6.6.2 Extended device driver reason codes**

The following is a list of the extended device driver reason codes:

DVBGB	get a block of bytes from a file
DVBPB	put a block of bytes to a file
DVLOD	load a file into memory
DVSAV	save a block of memory to a file
DVRLE	read the LOAD and EXEC addresses from a file
DVWLE	write LOAD and EXEC addresses to a file
DVRAT	read file attributes
DVWAT	write file attributes
DVRSP	read sequential file pointer
DVWSP	write sequential file pointer
DVRPL	read a file's physical length
DVRLL	read a file's logical length
DVWLL	write a file's logical length
DVRCH	read catalog header
DVRFN	read file/object names from a directory
DVDEL	delete an object
DVREN	rename an object

# DVRST

reset the device

This reason code is not currently defined.

## DVOPN open the device driver

**Action:** Attempts to open a device driver or file in the specified mode (INPUT, OUTPUT or UPDATE) and if successful returns a 16 bit handle. The function is invoked with a pointer to the driver or file name in BHA and the mode in Y. The 16 bit handle is returned in Y.

**On entry:**

- BHA -> name
- Y = open mode
  - &40 - INPUT
  - &80 - OUTPUT
  - &C0 - UPDATE

**On exit:**

- C = 0 open succeeded
- Y = 16 bit handle
- C = 1 open failed
- X = error code
- BHA -> error message

# DVCLS close the device driver

Action: Attempts to close the specified driver or file. Called with 16 bit device or file handle in Y.  
On entry: Y = 16 bit handle  
On exit: C = 0 closed OK  
C = 1 close failed  
X = error code  
BHA -> error message

## DVBGT get a byte from the device driver

**Action:** Attempts to get a data byte from the device driver. Called with a 16 bit handle in Y. If successful returns the data byte in A.

**On entry:** Y = 16 bit handle

**On exit:** C = 0 got a byte OK  
A = data byte

C = 1 failed to get a byte  
X = error code  
BHA -> error message

# DVBPT put a byte to the device driver

**Action:** Attempts to write a byte to the device driver. Called with a 16 bit handle in Y and the data byte in A.

**On entry:** Y = 16 bit handle  
A = data byte

**On exit:** C = 0 put byte succeed  
C = 1 put byte failed  
X = error code  
BHA -> error message

## DVCGT get a control byte from the device driver

**Action:** Attempt to get a control byte from the driver. Called with a 16 bit handle in Y. Returns control byte in A if successful.

**On entry:** Y = 16 bit handle

**On exit:** C = 0 success

A = control byte

C = 1 failure

X = error code

BHA -> error message

# DVCPT put a control byte to the device driver

**Action:** Attempts to write a control byte to the device driver. Called with 16 bit handle in Y and control byte in A.

**On entry:** Y = 16 bit handle  
A = control byte

**On exit:** C = 0 success

C = 1 failure  
X = error code  
BHA -> error message

## DVEOF return EOF status for device/file

**Action:** Returns the End Of File status for a device driver. Called with 16 bit handle in Y. Returns EOF status in A if successful.

**On entry:** Y = 16 bit handle

**On exit:** C = 0 success

A = EOF status byte

0 - not at End Of File

1 - at End Of File

C = 1 failure

X = error code

BHA -> error message

# DVBGB get a block of bytes from a file

**Action:** Read a block of bytes from a file into a buffer. Called with a 16 bit file handle in Y and a pointer to a control block in BHA. The control block contains the byte transfer count and a pointer to the data buffer.

**On entry:** Y = 16 bit file handle  
BHA -> control block  
    CB+&00 = pointer to data buffer  
    CB+&04 = transfer count

**On exit:** C = 0 read successful, data in buffer  
X = 2 if less bytes read than requested otherwise X = 0  
transfer count set to number of bytes read  
C = 1 read failed  
X = error code  
BHA -> error message

## DVBPB put a block of bytes to a file

**Action:** Attempts to write a block of bytes to a file. Called with a 16 bit file handle in Y and BHA pointing to a control block. The control block contains a byte transfer count and a pointer to the data buffer.

**On entry:** Y = 16 bit file handle  
BHA -> control block  
CB+&00 = pointer to data buffer  
CB+&04 = transfer count

**On exit:** C = 0 write succeeded  
C = 1 write failed  
X = error code  
BHA -> error message

# DVLOD load a file into memory

- Action:** Attempts to load a file into a specified area of memory. Called with a pointer to a control block in BHA. The control block contains a pointer to the name of the file, the initial memory location where the file is to be loaded and the length of the memory location. If the length field is zero then the file is transferred in its entirety, otherwise it specifies the maximum number of bytes to be transferred and is updated upon exit to show how many bytes were read. On return, if the C flag is zero, X will contain zero if the file was completely loaded or two if not all of the file could be loaded. If the file is transferred successfully, the EXEC and LOAD entries of the control block will be set to the entries from the file's directory details.
- On entry:** BHA -> control block  
CB+&00 = pointer to filename  
CB+&04 = file LOAD address  
CB+&08 = file EXEC address  
CB+&0C = initial memory location  
CB+&10 = length of memory area
- On exit:** C = 0 transfer was OK  
LOAD and EXEC updated  
transfer count updated if required  
X = 0 if transfer complete, X = 2 if only partial transfer  
C = 1 transfer failed  
X = error code  
BHA -> error message

# DVS AV

save a block of memory to a file

**Action:** Attempts to save a block of memory from a specified location to a file. Called with a pointer to a control block in BHA. The control block contains a pointer to the name of the file, the initial memory location of the data block and the length of the block. In addition the LOAD and EXEC fields of the control block should be initialised to the values required for the equivalent fields in the file's directory entry.

**On entry:** BHA -> control block  
CB+&00 = pointer to filename  
CB+&04 = file LOAD address  
CB+&08 = file EXEC address  
CB+&0C = initial memory location  
CB+&10 = length of memory area

**On exit:** C = 0 data saved OK  
C = 1 data save failed  
X = error code  
BHA -> error message

# DVRLE

read the LOAD and EXEC addresses of a file

- Action:** Attempts to get the READ and LOAD addresses of the specified file from the file's directory entry. Called with a pointer to a control block in BHA. The control block contains a pointer to the filename. On return, if successful, the control block will contain the LOAD and EXEC addresses for the file.
- On entry:** BHA → control block  
CB+&00 = pointer to filename  
CB+&04 = file LOAD address  
CB+&08 = file EXEC address
- On exit:** C = 0 success  
LOAD and EXEC addresses in set up in control block  
C = 1 failed  
X = error code  
BHA → error message

## DVWLE write LOAD and EXEC addresses for a file

**Action:** Attempts to set the READ and LOAD addresses of the specified file in the file's directory entry. Called with a pointer to a control block in BHA. The control block contains a pointer to the filename and the new LOAD and EXEC addresses for the file.

**On entry:** BHA -> control block  
CB+&00 = pointer to filename  
CB+&04 = file LOAD address  
CB+&08 = file EXEC address

**On exit:** C = 0 success  
C = 1 failed  
X = error code  
BHA -> error message

# DVRAT read file attributes

**Action:** Attempts to get a copy of the specified file's attributes from its directory entry. Called with a pointer to a control block in BHA. The control block contains a pointer to the file name. If successful the attributes are returned in the control block. X is set to one if the object found is a file or two if its a directory.

**On entry:** BHA -> control block  
CB+&00 = pointer to file/object name  
CB+&04 = file attributes

**On exit:** C = 0 got attributes OK  
attributes returned in attributes field of the control block  
X = 1 object found was a file  
or  
X = 2 object found was a directory  
C = 1 failed to get attributes  
X = error code  
BHA -> error code

## DVWAT write file attributes

**Action:** Attempts to get a copy of the attributes of the specified file in its directory entry. Called with a pointer to a control block in BHA. The control block contains a pointer to the file name and the new file attributes.

**On entry:** BHA -> control block  
CB+&00 = pointer to file/object name  
CB+&04 = file attributes

**On exit:** C = 0 set attributes OK  
C = 1 failed to set attributes  
X = error code  
BHA -> error code

# DVRSP read sequential file pointer

**Action:** Attempts to get the current value of the specified file's sequential pointer. Called with a 16 bit file handle in Y and a pointer to a control block in BHA. If successful the contents of the file's sequential pointer are returned in the control block.

**On entry:** Y = 16 bit file handle  
BHA -> control block  
CB+&00 = sequence number

**On exit** C = 0 got pointer OK  
sequential file pointer returned in control block  
C = 1 failed to get file pointer  
X = error code  
BHA -> error message

## DVWSP write sequential file pointer

**Action:** Attempts to set the specified file's sequential pointer to a supplied value. Called with a 16 bit file handle in Y and a pointer to a control block in BHA. The new value for the sequential file pointer is supplied in the control block.

**On entry:** Y = 16 bit file handle

BHA -> control block

CB+&00 = sequence number

**On exit** C = 0 set pointer OK

C = 1 failed to set file pointer

X = error code

BHA -> error message

# DVRPL read a file's physical length

- Action:** Attempts to get the physical length of a specified file. Called with a pointer to a control block in BHA. The control block contains a pointer to the file name. On return, if successful, the physical length of the file will be in the control block.
- On entry:** BHA -> control block  
CB+&00 = pointer to file name  
CB+&04 = length
- On exit:** C = 0 got length OK  
physical file length in length field of control block  
C = 1 failed to get length  
X = error code  
BHA -> error message

## DVRLL read a file's logical length

- Action:** Attempts to get the logical length of a specified file. Called with a 16 bit file handle in Y and a pointer to a control block in BHA. On return, if successful, the logical length of the file will be in the control block.
- On entry:** Y = 16 bit file handle  
BHA -> control block  
CB+&00 = length
- On exit:** C = 0 got length OK  
logical file length in length field of control block  
C = 1 failed to get length  
X = error code  
BHA -> error message

# DVWLL write a file's logical length

- Action: Attempts to set the logical length of a specified file. Called with a 16 bit file handle in Y and a pointer to a control block in BHA. The control block contains the new logical length for the file.
- On entry: Y = 16 bit file handle  
BHA -> control block  
CB+&00 = length
- On exit: C = 0 set length OK  
C = 1 failed to set length  
X = error code  
BHA -> error message

# DVRCH read catalog header

**Action:** Attempts to read a catalog header from a given pathname. The pathname may contain wildcards. Called with a pointer to a control block in BHA. The control block contains a pointer to the pathname. If successful, the header information is returned in a 49 byte buffer referenced by the control block.

**On entry:** BHA -> control block  
CB+&00 = pointer to pathname  
CB+&04 = pointer to 49 byte buffer area.

**On exit:** C = 0 got header information OK  
buffer contents:  
offset  
&00 &00=owner, &FF=public  
&01 Cycle number  
&02 Boot option  
&03-&0C Directory name (without wildcards)  
&0D-&1C CSD disc name  
&1D-&26 CSD name  
&27-&30 LIB name  
  
C = 1 failed to get header  
X = error code  
BHA -> error message

# DVRFN

read file/object names from a directory

- Action: Given the pathname, which may contain wildcards, this call attempts to read the number and names of entries in the specified directory. The information is stored in a supplied buffer. Each name is terminated with a zero (&00). Called with a pointer to a control block in BHA. The control block contains a pointer to the pathname, a pointer to the buffer, the type of information required and sixteen bytes of filing system workspace which should be zeroed for the first call. If there are names remaining, the function may be invoked again as long as the contents of the control block are not disturbed.
- On entry: BHA -> control block  
CB+00 = pointer to CARD name  
CB+04 = pointer to buffer  
CB+08 = type of information  
0 - file name only (16 chars)  
1 - short info (20 chars)  
2 - full info (80 chars)  
CB+0A = number of entries  
CB+0C = 16 bytes of file system workspace
- On exit: C = 0 read succeeded  
X = the number of names read  
names and info are in the buffer  
C = 1 read failed  
X = error code  
BHA -> error message

## DVDEL delete an object

**Action:** Attempts to delete a specified object. Called with a pointer to the object name in BHA.

**On entry:** BHA -> object name

**On exit:** C = 0 object deleted OK

C = 1 failed to delete object

X = error code

BHA -> error message

# DVREN rename an object

- Action: Attempts to rename the specified object. Called with a pointer to a control block in BHA. The control block contains a pointer to the old object name and a pointer to the new object name.
- On entry: BHA -> control block  
CB+&00 = pointer to old object name  
CB+&00 = pointer to new object name
- On exit: C = 0 rename succeeded  
C = 1 rename failed  
X = error code  
BHA -> error message

## **6.7 The clock device driver**

### **6.7.1 Introduction**

The clock device driver is responsible for maintaining and providing access to a system clock and calendar service.

### **6.7.2 Description**

The clock device driver is located in the Clock Module. The clock driver is one of a family of "old" device drivers. These old drivers were implemented before the standard device driver interface was defined and therefore do not conform to it. Clock driver services must be requested via references to the clock module with supplied reason codes specifying the desired facility. For more detail on referencing and calling modules, see Chapter 2.

The clock driver maintains a record of the current time (in hours and minutes), day, week, month and year.

The clock driver provides functions for other modules to read and set the time and date.

In addition to the providing an interface for other modules, the clock driver can be accessed by the user via the star (\*) commands \*time and \*set.

### 6.7.3 Clock driver functional interface

The clock driver provides a set of services which can be invoked via its functional interface.

To access the clock driver the calling routine must first obtain its entry point via the OPRFR COP function. Having obtained the entry point, the caller invokes the clock driver supplying a reason code specifying the required service along with any necessary parameters.

As an alternative, the clock driver can be invoked via the OPCMD COP call.

The following functions are provided by the clock driver interface:

CKCMD	Enter the clock command handler.
CKINIT	Initialise the clock handler and claim memory for it.
CKDAY	Obtain the day of the week.
CKREAD	Read the time and date.
CKSET	Set the time and date.
CKPAGE	Obtain the clock handler's direct page address.
CKMNTH	Get the number of days in a given month.

### 6.7.4 An Example of accessing the clock driver

The following piece of example code accesses the clock driver function via the OPRFR COP call. The program invokes the clock driver command handler, passing it the command to print the time and date.

```
170 SEP# &30          \ Make sure processor is in byte mode.  
180 BYT &30          \ Tell assembler that processor is in byte mode.  
190 COP  
200 EQUB @OPADP%  
210 EQUW &0008  
220 TAD  
225 :  
230 COP  
240 EQUB @OPBHA  
250 EQU$ "CLOCK"  
260 EQUB &00  
270 LDX &00  
275 LDY &00  
280 COP  
290 EQUB @OPRFR%  
300 BCS end%  
305 :  
310 LDX @CKCMD%  
320 LDX &02  
330 PHK  
340 JSR callmodule%  
345 :  
350 end%  
360 RTL  
365 :  
367 :  
370 callmodule%  
380 PEI &04  
390 PEI &02  
400 PEI &00  
410 RTL          \ Routine to invoke the referenced module.  
                  \ Push contents of D+5 & D+4 on the stack.  
                  \ Push contents of D+3 & D+2 on the stack.  
                  \ Push contents of D+1 & D+0 on the stack.  
                  \ Jump to module entry address just pushed on the stack.
```

Lines 190-220 Allocate an 8 byte block of Direct Page and transfers a pointer to it into the Direct Page Register.

Lines 230-260 Get a pointer to the name of the clock driver into BHA.

Lines 270-300 Set up a reference to the clock driver in Direct Page.

Lines 310-320 Set the clock driver reason code to CKCMD (clock driver command handler), specifying "ltime" as the command to be invoked.

Lines 330-340 Invoke the subroutine "callmodule" in a bank independent manner.

Line 360 Exit back to BASIC.

Line 370 Entry point of "callmodule".

Lines 380-400 Push the address of the clock driver's entry point minus 1 onto the stack.

Line 410 invoke the clock driver via an RTL.

### **6.7.5 Clock driver functions**

Clock driver reason codes are described on the following pages.

## **CKCMD enter the command handler**

**Action:** This reason code invokes the clock driver command handler. The command handler is called with a command code number in Y specifying which command is to be invoked and a pointer to a parameter string in BHA where required.

The functions invoked are the same as those available to the user from the BASIC command line (\*time and \*set).

**On entry:** BHA -> a parameter string (when required).

Y = command number

where

0	=	info command
4	=	ltime command
8	=	set command
12	=	info command

**On exit:** C = 1 error

C = 0 OK.

## **CKINIT Initialise the clock handler and claim memory for it.**

**Action:** This function initialise the clock handler and obtains some workspace for it.

**On entry:** None

**On exit:** C = 1    Bad Initialisation  
              C = 0    Ok.

## **CKDAY** obtain the day of the week.

**Action:** This function returns the day of the week that a particular date falls on. The function is called with the date in BHA, the day number is returned in X and a pointer to the day string in BHA.

**On entry:**  
B = Year-1900  
H = month  
A = date

**On exit:**  
C = 1 if failed  
C = 0 if OK  
X = Day number (0->6)  
BHA -> Day string (e.g. "Thursday")

# CKREAD

read the time and date.

**Action:** This function returns the current time and date to the calling routine. The information is returned as a pointer to a string, or in a parameter block supplied by the caller. The returned information is formatted according to the contents of a supplied option byte.

**On entry:** Y = Option Byte  
where

bit 0 reset =	date format dd/mm/yy
bit 0 set =	date format day month year
bit 1 reset =	Don't print day of the week
bit 1 set =	Print day of the week
bit 2 reset =	Short form of day (e.g. "MON")
bit 2 set =	Long form of day (e.g. "MONDAY")
bit 3 reset =	No date suffix (e.g. "27")
bit 3 set =	Date suffix (e.g. "27th")
bit 4 reset =	Short form of month (e.g. "NOV")
bit 4 set =	Long form of month (e.g. "NOVEMBER")
bit 5 reset =	24 hour time format
bit 5 set =	12 hour time format
bit 6 reset =	Don't print "AM" and "PM"
bit 6 set =	Print "AM" and "PM"
bit 7 reset =	BHA -> parameter block on exit.
bit 7 set =	BHA -> string on exit.

BHA -> Parameter block (if bit 7 is set in option byte)

**On entry:** C = 1 if error  
C = 0 if OK  
BHA -> parameter block containing return information  
or  
BHA -> string containing return information

# CKSET set the time and date.

**Action:** This function allows the calling routine to set the time and date. The function is invoked with a pointer to a parameter string containing the new time/date info in BHA. The string takes the format "dd/mm/yy hh:mm"

where:

dd = date  
mm = month  
yy = year  
hh = hours  
nn = minutes

**On entry:** BHA -> string containing new date and time.

**On exit:** C = 0 for successful update  
Y = 0

C = 1 Update attempt failed

Y = Error Code.

where:

1	=	date < 1
2	=	no "/" after date
3	=	month < 1
4	=	month > 12
5	=	no "/" after month
6	=	year < 1900
7	=	year > 2155
8	=	no ":" after year
9	=	date > length of specified month
10	=	hour > 23
11	=	no ":" after hour
12	=	minute > 59
13	=	IIC bus error

# CKPAGE obtain the clock handler's direct page address

**Action:** This function returns a pointer to the Clock handler's direct page in register D.

**On entry:** None

**On exit:** D → direct page

## **CKMNTH** get the number of days in a given month

**Action:** This function returns the number of days in a specified month, taking account of leap years. The calling function supplies the month and year in BHA and the number of days are returned in X. In addition the first day of the month is returned in Y (e.g. 0 for Sunday) and a pointer to the month name as a string is returned in BHA.

**On entry:** B = Year-1900  
H = month

**On exit:** C = 1 if error

C = 0 if OK  
X = number of days in the month  
Y = the first day in the month  
BHA → month name string

## **6.8 ECONET:**

The Econet module uses the standard device driver interface. Calls to this module should therefore always be made with a reason code in X, a 16-bit handle in Y (except for OPEN), and the byte to be PUT (if any) in A. All the reason codes given in the introduction are supported.

A% = OPENUP "ECONET:" :REM A% will always be even  
B% = A% + 1 :REM B% is now the control handle  
BPUT# B%, "S254" :REM select station number 254  
CLOSE# A%

The commands are as follows:

N            Read network number  
S            Read or write station number  
P            Set protection.

One of the following options may be set:

- safe
- view and notify allowed
- unprotected

# OPECO call low-level Econet routines

**Action:** This call performs data transfer to and from the Econet filing system. It performs several tasks depending upon the reason code in X.

**On entry:**

X = reason code.

The reason codes (described in more detail later) are as follows:

X = @ECOTX%	Transmit a packet.
X = @ECORO%	Open control block for normal reception.
X = @ECOPR%	Open control block to receive remote procedure call.
X = @ECOOP%	Open control block to receive OS procedure calls.
X = @ECORP%	Poll for reception.
X = @ECORR%	Read receive block.
X = @ECORD%	Delete receive block.
X = @ECOXB%	Allocate RAM for extra control blocks.
X = @ECONB%	Return number of free receive control blocks.

Other registers contain or point to arguments.

**On exit:**

C = 0 means that the call was successful.

If C = 1 and X < 0 then the call failed and the error code is in X.

If C = 1 and X = 0 then the Econet module is not present.

D preserved

**To call:**

COP

EQUB @OPECO% from BASIC

OPSYS £OPECO from MASM

**Example:**

```

30 DIM txspace% 8 :REM allocate 8 bytes for broadcast transmission
40 DIM rxspace% 8500 :REM allocate 8500 bytes receive space
50 $txspace% = "TYPE" + CHR$(32) + CHR$(32) :REM Initialise Tx block
60 txspace%?7 = 1 : txspace%?7 = 0 :REM initialise Tx block

170 RSP# &30          \ Put processor in word mode.
180 WRD &30          \ Tell assembler that processor is now in word mode.
190 BRA start%        \ Go to start of code.

200 .txblock%         \ Start of Tx control block.
210 EQUB &80          \ Control code.
220 EQUB &9C          \ Port number.
                           \ In this case the command receive port
                           \ for all the printers on the network.
                           \ Broadcast to all stations.
230 EQUW &FFFF
240 EQUD txspace%
250 EQUD txspace% + 8
260 EQUD &00000000
270 .rxblock%         \ Start of Rx control block.
280 EQUB &00          \ Flag (irrelevant when opening).
290 EQUB &00          \ Any port.
300 EQUW &0000
310 EQUD rxspace%
320 EQUD rxspace% + &500
330 EQUD &00000000
                           \ Not required.

340 .start%           \ Start of code.
350 PHK               \ Push high byte of program counter.
360 PLB               \ Pull high byte into B.
370 BNK P%/&10000
380 LDA# rxblock%    \ Tell assembler that B = K.
                           \ Load HA with address.
                           \ BHA now points to the start of the Rx control block.
390 LDX# @ECORO%
400 COP               \ Load X with reason code.
410 EQUB @OPEC0%
420 BCS end%          \ Execute COP call.
                           \ Open for normal reception.
430 PHY               \ If error then end.
440 .retry%           \ Save the 16-bit handle.
                           \ Set label to retry transmission.
450 PHK               \ Push high byte of program counter.
460 PLB               \ Pull high byte into B.
470 BNK P%/&10000
480 LDA# txblock%    \ Tell assembler that B = K.
                           \ Load HA with address.
                           \ BHA now points to the start of the Tx control block.
490 LDX# @ECOTX%
500 COP               \ Load X with reason code.
510 EQUB @OPEC0%
520 BCC poll%         \ Execute COP call.
                           \ Transmit packet.
                           \ If packet transmitted then poll for reception.
530 TXA               \ If packet not transmitted then transfer X to A.
540 SWA               \ Swap H and A to get low byte of X in H.
550 ASL A              \ Shift high bit of low byte of X into C.
560 BCS delete%       \ If X > &7F then delete receive block.
570 BRA retry%        \ If X < &7F then retry transmission.
580 .poll%             \ Poll for reception.
590 LDA# &FFFF
600 STA rxblock%     \ Load HA with &FFFF.
                           \ and store this in the first two bytes of the Rx block.
                           \ The Rx block is now not needed so we shall use it as a counter.
610 .loop%             \ Label to loop to.
620 DEC rxblock%      \ Decrement the counter.
                           \ If it has turned out then delete the Rx block.
630 BEQ delete%       \ Load X with reason code.
640 LDX# @ECORP%
650 PLY               \ Pull 16-bit handle into Y.

```

## Chapter 6

```
660 PHY
670 COP
680 EQUB @OPEC0%
690 BCS yend%
700 TXA
710 LSR A
720 BCC loop%
730 PHK
740 PLB
750 BNK P%/$10000
760 LDA# rxblock%

770 LDX# @ECORR%
780 PLY
790 COP
800 EQUB @OPEC0%
810 BRA end%
820 .delete%
830 LDX# @ECORD%
840 PLY
850 COP
860 @OPEC0%
870 BRA end%
880 .yend%
890 PLY
900 .end%

    \ Save the handle.
    \ Execute COP call.
    \ Poll for reception.
    \ If error then tidy up stack and end.
    \ Transfer X to A.
    \ Shift low bit into C.
    \ If no reception then poll again.
    \ Push high byte of program counter.
    \ Pull high byte into B.
    \ Tell assembler that B = K.
    \ Load HA with address.
    \ BHA now points to an area of memory in which
    \ to copy the Rx block from the Econet workspace.
    \ Load X with reason code.
    \ Pull 16-bit handle into Y.
    \ Execute COP call.
    \ Read Rx block.
    \ Go to end.
    \ Start of Rx control block delete.
    \ Load X with reason code.
    \ Pull 16-bit handle into Y.
    \ Execute COP call.
    \ Delete Rx control block.
    \ Go to end.
    \ Tidy stack and end.
    \ Pull Y to leave stack as found.
    \ End.
```

This example will open a receive block and then make a broadcast transmission to all stations logged on to the network. If the transmission fails then it will either retry or delete the receive control block in the Econet workspace. If transmission is successful then it will poll for reception in a closed loop. Normally a program would carry on doing other tasks and would poll for reception only intermittently. If reception does not occur within 65535 polls then the receive control block is deleted. If reception is successful then the receive control block is copied back from the Econet workspace to the user workspace so it can be read.

This call must be made in 16-bit XY mode - handles are 16 bits.

# X = @ECOTX% transmit a packet

**Action:** Transmit up to &500 bytes from memory to the destination Eonet station(s).

**On entry:** X = @ECOTX%

BHA points to a control block in memory.

The format of the control block is as follows.

Offset

0	single-byte control code (high bit set)	(or immediate op type)
1	single-byte port number	(zero for immediate ops)
2	two-byte destination station	(&FFFF for broadcast to all stations logged on)
4	four-byte buffer start	(high byte must be 0)
8	four-byte buffer end + 1	(Not more than &500 from buffer start)
C	four-byte immediate op remote address etc	(not required for normal transmission)
10		

Length of block &10 bytes.

**On exit:** C = 0 means transmission was successful.

If C = 1 then the call failed to transmit and the error code is in X.

BHA points to zero-terminated error message.

If X > &7F then failure was total - not worth retrying.

If X < &80 then the transmission failed - worth retrying.

D preserved

**To call:**

LDX# @ECOTX%

COP

EQUB @OPECO% from BASIC

LDXIM £ECOTX

OPSY8 £OPECO from MASM

Note the 2 differences from the BBC implementation:

There is no Poll for Transmission call, so the Tx call does not return until the transmission is complete.

A broadcast transmission to all stations logged on to the network no longer requires that the data be put in the control block. Buffer start and end addresses are used to point to a maximum 8 byte buffer.

# X = @ECORO% open for normal reception

**Action:** After issuing this call, the net interface will accept incoming data from the station specified in the control block. A copy of the control block is made in the Econet workspace, so once this call has been issued the control block may be overwritten.

**On entry:** X = @ECORO%  
BHA points to a control block in memory.

The format of the control block is as follows.

**Offset**

0	single-byte flag	(flag irrelevant when opening)
1	single-byte port number	(zero for any port)
2	single-byte source station number	(&0000 for
3	single-byte source network number	any station)
4	four-byte buffer start	(high byte must be 0)
8	four-byte buffer end + 1	
C	four-byte reserved	reserved for event address
10		

Length of block &10 bytes.

**On exit:** C = 0 means open was successful, handle in Y.  
If C = 1 then the call failed to open and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved

**To call:**  
 LDXI @ECORO%  
 COP  
 EQUB @OPECO% from BASIC  
 LDIXM £ECORO  
 OPSYS £OPECO from MASM

# X = @ECOPR% open to receive remote procedure call

**Action:** After issuing this call, the net interface will accept a remote procedure call from the station specified in the control block. A copy of the control block is made in the Econet workspace, so once this call has been issued the control block may be overwritten.

**On entry:** X = @ECOPR%  
BHA points to a control block in memory.  
The format of the control block is as follows.

Offset		
0	two-byte procedure number	
2	single-byte source station number	(&0000 for any station)
3	single-byte source network number	
4	four-byte buffer start	(high byte must be 0)
8	four-byte buffer end + 1	
C	four-byte reserved	reserved for event address
10		

Length of block &10 bytes.

**On exit:** C = 0 means open was successful, handle in Y.  
If C = 1 then the call failed to open and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved

**To call:** LDXW @ECOPR%  
COP  
EQUB @OPECO% from BASIC  
LDXIM EECOPR  
OPSYS EOPECO from MASM

# X = @ECOOP% open to receive OS procedure call

**Action:** After issuing this call, the net interface will accept an OS procedure call from the station specified in the control block. A copy of the control block is made in the Econet workspace so once this call has been issued the control block may be overwritten.

**On entry:** X = @ECOOP%  
BHA points to a control block in memory.

The format of the control block is as follows.

Offset		
0	two-byte procedure number	
2	single-byte source station number	(&0000 for any station)
3	single-byte source network number	(high byte must be 0)
4	four-byte buffer start	
8	four-byte buffer end + 1	
C	four-byte reserved	reserved for event address
10		

Length of block &10 bytes.

**On exit:** C = 0 means open was successful, handle in Y.  
If C = 1 then the call failed to open and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved

**To call:**  
 LDX# @ECOOP%  
 COP  
 EQUB @OPECO% from BASIC  
 LDXIM ECOOP  
 OPSYS OPECO from MASM

# X = @ECORP% poll for reception

- Action:** After opening for reception, this call is issued every now and then to see whether a packet has been received.
- On entry:** X = @ECORP%  
Y = handle
- On exit:** If C = 0 then X = 0 means no reception yet, X = 1 means reception has occurred.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved
- To call:** LDX# @ECORP%  
COP  
EQUB @OPECO% from BASIC  
LDXIM SECORP  
OPSYS SOPECO from MASM

## X = @ECORR%    read receive block

**Action:** When reception occurs, the control information is placed in the copy of the control block stored in the Econet workspace. This call makes another copy of this block in the user workspace so its contents can be read, deletes the block in the Econet workspace and frees the handle.

**On entry:** X = @ECORR%  
BHA points to space for the copy of the control block.

**On exit:** If C = 0 then control block is copied, old block deleted, and handle is no longer valid.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved

**To call:**  
LDXW @ECORR%  
COP  
EQUB @OPECO% from BASIC  
LDXIM EECORR  
OPSYS EOPECO from MASM

# X = @ECORD% delete receive control block

- Action:** If after opening for reception, for some reason no reception occurs, then the copy of the control block stored in the Econet workspace can be deleted using this call.
- On entry:** X = @ECORD%  
Y = handle
- On exit:** If C = 0 then block deleted.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved
- To call:** LDX# @ECORD%  
COP  
EQUB @OPECO% from BASIC  
LDXIM £ECORD  
OPSYS £OPECO from MASM

# X = @ECOXB% allocate extra RAM for control blocks

**Action:** This call will expand the Econet workspace to allow more control blocks to be stored.

**On entry:** No requirements.

**On exit:** If C = 0 then extra RAM successfully allocated.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved

**To call:** LDX# @ECOXB%  
COP  
EQUB @OPECO% from BASIC  
LDXIM ECOXB  
OPSYS OPECO from MASM

# X = @ECONB%    read number of free receive blocks

**Action:** Returns in X the number of receive control blocks which remain to be set up in the Econet workspace.

**On entry:** No requirements.

**On exit:** If C = 0 then X = number of free control blocks.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.  
D preserved

**To call:**  
LDX# @ECONB%  
COP  
EQUB @OPECO% from BASIC  
LDXIM £ECONB  
OPSYS £OPECO from MASM

# OPNET high-level Econet routines

**Action:** This call performs several tasks depending upon the reason code in X.

**On entry:** X = reason code.

The reason codes are as follows:

X = @NETOP%	Do a FileStore operation.
X = @NETXH%	Convert internal to external handle.
X = @NETRF%	Read current FileStore station number.
X = @NETWF%	Write FileStore station number.
X = @NETRR%	Read retry settings.
X = @NETWR%	Write retry settings.
X = @NETRC%	Read context handles.
X = @NETWC%	Write context handles.

Other registers contain or point to arguments.

**On exit:** C = 0 means that the call was successful.

If C = 1 then the call failed and the error code is in X.

If C = 1 and X = 0 then the Econet module is not present.

If C = 1 and X = 1 then BHA points to an error message.

D preserved

**To call:**

COP

EQUB @OPNET% from BASIC

OPSYS £OPNET from MASM

This call must be made in 16-bit XY mode - handles are 16 bits.

# X = @NETOP% write to / read from a FileStore

**Action:** Writes a block of data to a FileStore and reads a block back to the same place in memory.

**On entry:** X = @NETOP%  
BHA points to a control block in memory.

The format of the control block is as follows.

Offset		
0	two-byte size of data to send to FileStore	(Note: BBC had size + 2)
2	two-byte maximum size of reply from FileStore	(Note: BBC was unlimited)
4	single-byte port number	(Reply from FileStore)
5	single-byte function code	(Reply from FileStore)
6	three single-byte handles User root directory handle Currently selected directory handle Currently selected library handle	(8-bit handles from FileStore)
9	Data	(Rest of block)

Size of data from offset 9 to end must be stored at offsets 0 and 1, LSB first.

**On exit:** If C = 0 then transaction occurred.  
However, the FileStore's reply may be an error message.  
If C = 1 then an error occurred in which it was not possible to contact the FileStore, eg "Not listening".

Results are written back over bytes 4 to n of the control block, and bytes at offsets 2 and 3 are updated to show the size of the reply.

**To call:**  
 LDX# @NETOP%  
 COP  
 EQUB @OPNET% from BASIC  
 LDIXIM ENETOP  
 OPSYS EQPNET from MASM

**X = @NETXH%** convert internal to external handle

**Action:** Communicator internally uses 16-bit handles. The file server uses 8-bit handles. This call will convert internal handles to ones which the FileStore can use.

**On entry:** X = @NETXH%  
Y = internal (16-bit handle).

**On exit:** If C = 0 then A = external (8-bit) handle.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.

**To call:** LDXW @NETXH%  
COP  
EQUB @OPNET% from BASIC  
LDXIM LNETXH  
OPSYS LOPNET from MASM

# X = @NETRF%    read current FileStore station number

**Action:**    Reads the station number and network number of the currently selected FileStore.

**On entry:**    X = @NETRF%

**On exit:**    If C = 0 then A = station number and H = network number.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.

**To call:**  
LDX# @NETRF%  
COP  
EQUB @OPNET% from BASIC  
LDXIM @NETRF  
OPSYS @OPNET from MASM

**X = @NETWF%** write FileStore station number

**Action:** Same as typing `*Is n.s` where n is the network number and s is the station number. s is passed in A and n is passed in H.

**On entry:** X = @NETWF%  
A = station number  
H = network number

**On exit:** If C = 0 then station number is set.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.

**To call:** LDXW @NETWF%  
COP  
EQUB @OPNET% from BASIC  
LDXIM £NETWF  
OPSYS £OPNET from MASM

# X = @NETRR% read retry settings

**Action:** Reads the current retry settings into a control block.

**On entry:** X = @NETRR%

BHA points to empty &0A byte control block.

**On exit:** If C = 0 then the control block is updated.

If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.

The format of the control block on exit is as follows.

**Offset**

0	Number of Tx retries for normal transmission	(Number of times to transmit)
2	Delay between Tx retries	(In centiseconds)
4	Number of Tx retries for machine peek	(Number of times to transmit)
6	Delay between Tx retries	(In centiseconds)
8	Maximum Rx wait	(In centiseconds. 0 to wait forever)

A

Length of block &0A bytes.

**To call:**

LDX# @NETRR%  
COP  
EQUB @OPNET% from BASIC  
  
LDXIM LNETRR  
OPSYS LOPNET from MASM

# X = @NETWR% write retry settings

**Action:** Writes the retry settings contained in the control block.

**On entry:** X = @NETWR%

BHA points to a control block in memory.

The format of the control block is as follows.

Offset

0	Number of Tx retries for normal transmission	(Number of times to transmit)
2	Delay between Tx retries	(In centiseconds)
4	Number of Tx retries for machine peck	(Number of times to transmit)
6	Delay between Tx retries	(In centiseconds)
8	Maximum Rx wait	(In centiseconds. 0 to wait forever)

A

Length of block &OA bytes.

**On exit:** If C = 0 then the settings have been updated.

If C = 1 then an error occurred and the error code is in X.

BHA points to zero-terminated error message.

**To call:** LDX# @NETWR%

COP

EQUB @OPNET% from BASIC

LDXIM ENETWR

OPSYS COPNET from MASM

# X = @NETRC% read context handles

**Action:** Reads the 16-bit internal format handles of the user root directory, the currently selected directory, and the currently selected library into a control block.

**On entry:** X = @NETRC%  
BHA points to an empty 6 byte control block

**On exit:** If C = 0 then context read.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.

The format of the control block on exit is as follows.

Offset  
0 URD handle  
2 CSD handle  
4 Library handle  
6

Length of block &06 bytes.

**To call:** LDX# @NETRC%  
COP  
EQUB @OPNET% from BASIC  
LDXIM \$NETRC  
OPSYS \$OPNET from MASM

# X = @NETWC% write context handles

**Action:** Writes the 16-bit internal format handles of the user root directory, the currently selected directory, and the currently selected library from a control block.

**On entry:** X = @NETWC%  
BHA points to a control block in memory.

The format of the control block is as follows.

Offset	
0	URD handle
2	CSD handle
4	Library handle
6	

Length of block &06 bytes.

**On exit:** If C = 0 then context handles updated.  
If C = 1 then an error occurred and the error code is in X.  
BHA points to zero-terminated error message.

**To call:** LDX# @NETWC%  
COP  
EQUB @OPNET% from BASIC  
LDXIM £NETWC  
OPSYS £OPNET from MASM

## 6.9 RAM:

RAM: is the device driver through which access is gained to the internal RAM filing systems. There are two types of RAM filing system - CMOS non-volatile RAM and dynamic RAM.

### 6.9.1 32K CMOS non-volatile memory

Filing commands have the format:

**RAM:filename**

Where RAM: specifies an internal filing system and the prefix ! selects the CMOS filing system.

### 6.9.2 32K dynamic RAM

Filing commands have the format:

**RAM:filename**

Note that RAM: files beginning with a "!" character are CMOS files (see previous section). RAM: filenames are deemed to be dynamic RAM files. RAM: filenames can include the character ":" in their names; a directory structure however, is not supported.

A subset of the standard operations is available in the RAM filing systems:

- \*CAT
- \*LOAD
- \*SAVE
- \*EXEC
- \*DELETE
- \*RENAME

From BASIC the OPEN primitives OPENIN, OPENOUT and OPENUP are supported.

For example,

**A% = OPENIN("RAM:!testfile")**

Other file operators such as BPUT#, PTR# etc are all supported. Note though that RAM: files do not have any attributes, load and exec addresses or date stamp associated with them.

The following conventions are adopted for RAM: files used by the system:

<b>lc.xyz</b>	The configuration file for device xyz:
<b>ls.abc</b>	Screen information stored by the terminal software and accessed by the Carousel task.
<b>lp.directory</b>	File containing the directory information for the Phone task.
<b>c.main</b>	DRAM menu configuration file. If it exists, it is used in preference to !c.main (the CMOS configuration file).

## 6.10 CARD: the memory card filing system

### 6.10.1 Introduction

The card filing system is currently only available on the SpectarII Briefcase Communicator.

The card filing system allows the use of up to eight ASTRON Data Cards at one time, providing a flexible method of portable mass data storage and retrieval.

### 6.10.2 Description

The card filing system provides facilities similar to those available on the Econet filing system.

Access to the card filing system is via standard BBC BASIC file system commands or low-level device-driver calls.

#### *Naming conventions*

File names and Card names are restricted to a maximum of ten alpha-numeric characters. Names must begin with an alphabetic character and may contain the period character (".").

Wildcard characters start ("\*") and hash ("#") may be used when selecting filenames.

From the command line, access to card files follows the naming convention:

**CARD:<cardname><filename>**

where

**CARD** specifies the card filing system

**cardname** is an optional name specifying card to access

**filename** is an optional name specifying the file to access.

### 6.10.3 Card filing system commands

The card filing system supports the standard BASIC and OS file system commands. Support is provided for most of the standard device-driver low-level reason codes.

#### *Card supported operating system commands*

The card filing system supports the following Operating System commands:

*LOAD	load a file into memory
*SAVE	save an area of memory into a file
*EX	show extended file information
*DELETE	delete a file
*RENAME	rename a file

The following OS commands are card filing system specific:

*ACCESS	set access file access options
*FORMAT	format cards for use
*FREE	return free space information on CARDS
*PROTECT	make RAM cards read only

#### *Card supported BASIC file operations*

The card filing system supports the usual BASIC file operation commands:

OPENIN	open existing file for input
OPENOUT	create and open file for output
OPENUP	open existing file for random access
CLOSE#	close a file
BPUT#	place bytes in a file
BGET#	get a byte from a file
=EXT#	return the length of a file
=PTR#	return pointer to the current write position in a file
PTR#=	set write position in a file
EOF#	return current End Of File condition for a file

After any of the BASIC file operations are performed, an error state is returned in the variable @ERC\$rm . If @ERC% is true, an error message can be found at the address @ERMSG\$rm .

#### *Card supported device driver reason codes*

The card file system supports all of the standard device driver reason codes, although some of them are not implemented.

The card filing system implements the following standard device driver reason codes:

DVOPN	open a file
DVCLS	close a file
DVBGT	get a byte from a file
DVBPT	put a byte in a file
DVEOF	test for EOF on a file
DVBGB	get a specified number of bytes from a file
DVBPB	put a block of bytes into a file
DVLOD	load a file into memory
DVSAY	save a section of memory into a file
DVRLE	read associated information on a file
DVRAT	read file attributes
DVWAT	write file attributes
DVRSP	return the sequential pointer for a file
DVWSP	write the sequential pointer for a file
DVRPL	read the physical length of a file
DVRLL	read the logical length of a file
DVRFN	return cards or files present
DVDEL	delete a file
DVREN	rename a file

#### **6.10.4 Card filing system functional interface**

The card filing system presents a standard device driver functional interface to the outside world. Functions are invoked via the usual device driver reason codes as listed above.

The standard method by which errors are returned is for the device driver function to exit with the carry flag set, an error number in X and a null terminated error message referenced by BHA.

# DVOPN open a device

**Action:** Tries to open the specified file in the specified mode, (INPUT, OUTPUT or UPDATE). If successful a 16 bit handle is returned in Y.

**On entry:** BHA -> filename (terminated by &0D)

Y = open type

&40 - INPUT

&80 - OUTPUT

&C0 - UPDATE (INPUT/OUTPUT)

**On exit:** C = 0

Y = 16 bit file handle

C = 1

X = error code

BHA -> error message

## DVCLS close a file

**Action:** This function is used to close an opened file. The procedure will attempt to close the file with the handle supplied in Y. After closing the file the handle cannot be used for any future accesses to the file.

**On entry:** Y = 16 bit file handle

**On exit:** C = 0 file closed

C = 1

X = error code

BHA -> error message

# DVBGT get a byte from a file

**Action:** This function gets a byte value from the specified file handle. The function reads a byte from the sequential pointer within the file specified by the file handle in Y. The file pointer is incremented to point to the next byte.

**On entry:** Y = 16 bit file handle

**On exit:** C = 0  
A = byte value

C = 1  
X = error code  
BHA -> error message

## DVBPT put a byte in a file

**Action** This function is used to put a byte value to the specified file. The function will write the byte value supplied in A to the file specified by the 16 bit handle in Y, if the file is opened for OUTPUT. After the write, the sequential pointer value for the file is incremented to the next character write position in the file.

**On entry:** A = byte value to be written  
Y = 16 bit file handle

**On exit:** C = 0 then byte was written OK  
C = 1  
X = error code  
BHA -> error message

# DVEOF test for EOF on a file

- Action:** This function returns the current End Of File state for the file specified by the 16 bit handle in Y.
- On entry:** Y = 16 bit file handle
- On exit:**
- C = 0
  - A = 0 then not EOF
  - or
  - A = 1 then EOF
  - C = 1
  - X = error code
  - BHA -> error message

## DVBGB get a specified number of bytes from a file

**Action:** This procedure reads a block of bytes from the file handle, supplied in Y, into the buffer associated with the control block referred to by BHA. The sequential file pointer is incremented to the byte after the block read.

**On entry:** Y = 16 bit handle  
BHA -> control block  
CB+0 = pointer to buffer  
CB+4 = number of bytes

**On exit:** C = 0 read successful, buffer contains data  
C = 1 read failed  
X = error code  
BHA -> error message

# DVBPB put a block of bytes into a file

- Action** This procedure writes a block of bytes supplied in a buffer referred to by the control block pointed to by BHA, to the file specified by the 16 bit handle in Y. The sequential file pointer is incremented to the byte after the block written.
- On entry:**
- Y = 16 bit file handle
  - BHA -> control block
    - CB+0 = pointer to buffer
    - CB+4 = number of bytes
- On exit:**
- C = 0 buffer contents were written to the file OK
  - C = 1 write failed
  - X = error code
  - BHA -> error message

# DVLOD

load a file into memory

- Action** This procedure loads a file into memory. The file name and load address are supplied in a control block referred to by BHA. The sequential file pointer is incremented to the byte after the block written.
- On entry:** BHA -> control block  
CB+00 = pointer to filename  
CB+04 = load address
- On exit:** C = 0 file loaded OK  
X = 0 address valid  
or  
X = 2 address invalid  
C = 1 failed to load file  
X = error number  
BHA -> error message

# DVSAV save a section of memory into a file

**Action:** This procedure saves a section of memory to a file. The filename, start address and length of the data area are supplied in the control block referenced by BHA.

**On entry:** BHA -> control block  
CB+00 = pointer to filename  
CB+0C = data start  
CB+10 = data length

**On exit:** C = 0 data saved OK  
X = 0

**On exit:** C = 1 failed to save data  
X = error number  
BHA -> error message

# DVRLE read associated information on a file

**Action:** this function returns information about the file named in the parameter block referenced by BHA.

**On entry:** BHA -> control block  
CB+00 = pointer to filename

**On exit:** C = 0  
XHA = file length (32 bits)  
C = 1 error  
X = error code  
BHA -> error message

# DVRAT

**Action:** This procedure reads the given attributes (32 bits) for the specified file. The filename is supplied in the control block referenced by BHA. The attributes are returned in a four byte area in the same control block. No wildcards are allowed in the filename.

**On entry:** BHA -> control block  
CB+00 = pointer to filename  
CB+04 = attributes area  
within the longword byte 3 should always be zero  
bit 16 - ON = LOCKED, OFF = UNLOCKED  
bit 17 - ON = NEVER SHOW FILE  
bit 20 - ON = HIDE FILE, OFF = SHOW FILE  
the rest of the bits are undefined

**On exit:**      C = 0 OK  
                   CB+4 = file attributes  
                   C = 1  
                   X = error code  
                   BHA -> error message

## DVWAT write file attributes

**Action:** This function writes the given attributes to the specified filename. The filename and attributes are supplied in the control block referenced by BHA.

**On entry:** BHA -> control block  
CB+00 = pointer to filename  
CB+04 = attributes area  
within the longword byte 3 should always be zero  
bit 16 - ON = LOCKED, OFF = UNLOCKED  
bit 17 - ON = NEVER SHOW FILE  
bit 20 - ON = HIDE FILE, OFF = SHOW FILE  
the rest of the bits are undefined

**On exit** C = 0 attributes written OK  
C = 1 failed to write attributes  
X = error code  
BHA -> error message

# DVRSP return the sequential pointer for a file

**Action:** This function returns the current value of the sequential file specified by the 16 bit handle in Y. The file pointer is returned in the control block referred to by BHA.

**On entry:** Y = 16 bit file handle  
BHA -> control block  
CB+00 = pir data area

**On exit:** C = 0 pointer in control block OK  
C = 1 failed to transfer pointer to control block  
X = error code  
BHA -> error message

# DVWSP write the sequential pointer for a file

**Action:** This function sets the sequential pointer for a specified file. The file is specified by a 16 bit handle in Y and the pointer value is supplied in a control block referenced by BHA.

**On entry:** Y = 16 bit file handle

BHA -> control block

CB+00 = new sequential pointer

**On exit:** C = 0 pointer written OK

C = 1

X = error code

BHA -> error message

## DVRPL read the physical length of a file

**Action:** This function returns the physical length of a specified file. The file name is referenced by a pointer in the control block passed in BHA. The length is returned in the same control block.

**On entry:** BHA -> control block  
CB+00 = file name pointer  
CB+04 = length of data area

**On exit:** C = 0 length in control block OK  
C = 1  
X = error code  
BHA -> error message

# DVRLL read the logical length of a file

**Action:** This function returns the logical length of the specified file (specified by the 16 bit handle in Y). The logical length is returned in the control block supplied by the caller and referenced by BHA.

**On entry:** Y = 16 bit file handle  
BHA > control block  
CB+00 = length data area

**On exit:** C = 0 logical length in control block OK  
C = 1  
X = error code  
BHA > error message

# DVRFN return cards or files present

**Action:** This function returns a list of the files associated with a specified card. If a card is not specified, (i.e. card name of &0D is given) a list of cards installed in the system is returned. The file or card names are returned in a block of memory provided by the caller. The caller can specify the format of the information and the maximum number of names to be returned per call. Parameters are passed via a control block referred to by BHA. Each name returned is separated by zero (&00). The number of names read is returned in X. The control block contains 16 bytes of file system work space which should be zeroed for the first call. If there are names remaining, the function may be invoked again as long as the contents of the control block are not disturbed.

**On entry:** BHA -> control block  
CB+00 = pointer to CARD name  
CB+04 = pointer to buffer  
CB+08 = type of information  
    0 - file name only (16 chars)  
    1 - short info (20 chars)  
    2 - full info (80 chars)  
CB+0A = number of entries  
CB+0C = 16 bytes of file system workspace

**On exit:** C = 0 buffer contains name data  
X = number of entries read  
C = 1  
X = error code  
BHA -> error message

# DVDEL delete a file

**Action** This procedure deletes the named file. A pointer to the filename is passed in BHA.

**On entry:** BHA -> file name

**On exit:** C = 0 file deleted OK

C = 1

X = error code

BHA -> error message

## DVREN rename a file

**Action:** This function renames the specified file. Pointers to the file name and its replacement are passed in a control block referenced by BHA.

**On entry:** BHA -> control block  
CB+00 = old name pointer  
CB+04 = new name pointer

**On exit** C = 0 file renamed OK  
C = 1  
X = error code  
BHA -> error message

*Chapter 6*

The following reason codes are supported but not implemented by the card filing system

**DVRST** reset device

**DVCGT** get control byte

**DVCPT** write a control byte

**DVWLE** write associated file information

**DVWLL** write logical length to a file

**DVRCH** read catalogue header

## 6.11 Autoboot facilities

### 6.11.1 Boot files at logon

Whenever a user logs on to a FileStore, either by a 'I am command from BASIC, VIEW etc or by pressing the logon button in the menu, an autoboot may be invoked. A boot option is kept by the FileStore for each user, and after the logon this is inspected to cause the following effects (effect on a BBC micro shown for reference):

Option	Communicator	BBC micro
0	No action	No action
1	Reserved	*LOAD !BOOT
2	*RUN !CBOOT	*RUN !BOOT
3	*EXEC ICBOOT	*EXEC !BOOT

The option can be inspected by typing \*CAT - the option is displayed at the top right of the catalogue display. Note that this has nothing to do with the directory being catalogued.

There is one boot option per user per FileStore. The displayed value is that which applied when you last logged on to that terminal.

There is not an option on each directory.

Note that the option is not held locally in Communicator, but in the PASSWORD FILE on the FileStore. If floppy-based FileStores are in use, there may be many password files on different discs.

The option can be changed by entering a \*OPT 4,n command, where "n" represents the required option number. This alters the boot option for the user/FileStore/disc combination currently in use.

### 6.11.2 Autostart at power-on

The menu program has a configured setting for autoboot user name. At startup, the menu sets up the screen with a suitable window in case the boot sequence is interactive, and executes 'I am <username>'. This will cause execution of the !CBOOT file if it is enabled for that user (\*OPT 4 etc). \*EXEC (option 3) is not a sensible boot option here, as the characters will be read by the menu, and ignored.

After logging on, the menu reads its Configure file. If a startup task has been defined in this file, such as "phone", then the phone front screen will be displayed instead of the menu. It is possible, of course, to get back to the menu by pressing STOP.

As an extra facility, the menu will attempt to read its Configure file from the volatile RAM filing system, file name RAM:c.main, reverting to the CMOS filing system if not found.

This means that the boot program can control the configuration of the task menu with three levels of severity:

- it can copy the CMOS file to DRAM, appending its startup information
- it can put a fixed configuration in DRAM
- it can permanently modify the CMOS file.

The autoboot can be configured not to happen by specifying a null user name.

## **6.12 PRINTER:**

### **6.12.1 Introduction**

The PRINTER: driver module provides output channels to the current printer output device. This may be either RS423: CENTRONICS: or NETPRINT:. PRINTER: imposes no limitations on the number of channels which may be open simultaneously. However, RS423: and CENTRONICS: are limited to one channel each, and currently so is NETPRINT:.

PRINTER: supports data output only; attempts to BGET from the channel will fail, even when output is via RS423:. To save space, RS423: will be opened with the minimum input buffer size.

PRINTER: does not provide direct access to the commands of the output device. Some commands (eg the RS423: data format, data rate, set input buffer size, and send break commands) will not be accessed at all. In the case of RS423: data rate and format commands, you will use the Configure mechanism to set default RS423: settings appropriate for the printer in use. Other output device commands will be accessed indirectly via the corresponding PRINTER: commands, eg set output buffer size, set/reset quit mode. Execution of these commands will consist entirely of passing the corresponding command on to the relevant output driver.

### **6.12.2 Device driver commands**

Commands are as follows:

#### **F**

Pass the corresponding F command to the output device, to flush the output buffer.

#### **Gnone/linefeed/return**

Defines the printer ignore character. All characters are compared with the ignore character stored in direct page. This contains either the ASCII code for <LF> or <CR>, or zero if no characters are to be ignored.

#### **O<hex size>**

Pass the corresponding O command to the output device.

#### **Qon/off**

Pass the corresponding Q command to the output device.

#### **Tcentronics/rs423/netprint**

Select output device type, and ensure device is open. If the requested device is already selected and opened by PRINTER:, this command has no effect; otherwise, the requested device is opened. Once the requested device is opened successfully, it becomes the currently selected output device, and any previous output device is closed.

#### **? commands**

The G Q O and T commands maintain status entries. The current status of any of these commands can be interrogated using the corresponding ? command. The status is returned via the CGET buffer.

The following is an example from BASIC, assuming B% is the control handle for PRINTER:

**100 BPUT# B%, "?T" 110 PRINT GET\$# B%**

This example will print the current T command status, for example, "Tcentronics".

## **6.13 CENTRONICS:**

### **6.13.1 Introduction**

The Centronics port is a uni-directional, parallel data port. It has 8 data output lines, and two handshaking lines: these are "data ready" output and "data taken" input. The port is controlled by the 6522 VIA.

### **6.13.2 Device driver commands**

Commands are as follows:

#### **Qon/off**

Switch quit mode on or off. This determines whether the driver will return immediately with carry set if attempting to BPUT to a full data buffer, or will loop until space is available in the buffer.

#### **O<hex size>**

Set output buffer size. This command will have no effect if there is already data in the buffer, or if the requested buffer size cannot be allocated.

#### **X<hex val>**

Direct write to hardware. Mainly for debugging purposes. The 3 byte value is made up as follows:

- |            |                 |
|------------|-----------------|
| low byte:  | EOR mask        |
| mid byte:  | Register number |
| high byte: | AND mask        |

Can write to any 6522 register.

#### **?Q**

Return current mode in CGET buffer.

#### **?O**

Return current buffer size in CGET buffer.

## 6.14 NETPRINT:

### 6.14.1 Introduction

Some means needs to exist whereby the generalised PRINTER: module can select a printer from the network. This is performed through the device driver NETPRINT:.

### 6.14.2 Software

Only one channel may be open to the NETPRINT: device at any one time. Further attempts to open it will result in the OPEN request being refused. The device should only be opened for output.

There are four configurable parameters which are set up using the OPEN string:

**net number** which specifies on which net (or any net) to find the printer.

**station number** which specifies on which station (or any station) on the selected net to find the printer.

**printer string** which identifies the type of printer (or any type) to be used.

**buffer flush character** which specifies which characters will cause immediate output of all buffered characters to the printer (including the buffer flush character itself).

The wild card for the net number is 255. 0 and 255 are wild cards for the station number. The printer type wild card is "print" or a null string. Net and station numbers may not be greater than 255.

The printer type string should not contain upper-case letters, and only the first 6 characters are used. Thus a user can specify, for example, any printer of type "LASER" on net 23 by specifying 23, 255 and "LASER" respectively.

The buffer flush character may be "none", "return", "formfeed", or "return/formfeed" (which will use either CR or FF). Formfeed is useful for printers which execute slow formfeeds. The buffer flush character is the only parameter which can be changed after the initial OPEN has occurred.

The netprint module uses the standard device driver interface. Calls to this module should therefore always be made with a reason code in X, a 16-bit handle in Y (except for OPEN, when it should contain the open type, &80, for OPENOUT), and the byte to be PUT (if any) in A. All the reason codes given in the introduction are supported.

Remember that a single channel has separate data and control streams; these are indicated by separate reason codes to the device driver, but are achieved by using odd or even handles from BASIC etc. The MOS notes the value of bit 0 of the handle, and selects the appropriate reason code. Thus from BASIC, one might type:

```
A% = OPENOUT "NETPRINT: N5;S255;Tlaser"
IF A%<0 THEN PRINT $@ERMSG% : STOP
B% = A% + 1
BPUT# B%, "Fnone"
BPUT# A%, "Hello World"
CLOSE# A%
```

:REM A% will always be even
 :REM print error message and end
 :REM Select net 5, any station, type laser
 :REM B% is now the control handle
 :REM select no buffer flush character
 :REM write to data channel

Commands are as follows:

**F<none|return|formfeed|return/formfeed>**

Define character(s) which upon receipt will cause the buffer, including that character, to be flushed immediately.

**N<dec val>**

Set printer net number, 0 <= N <= 255 (wildcard).

*Chapter 6*

**S<dec val>**

Set printer station number, 0 (wildcard) <= S <= 255 (wildcard).

**T<string>**

Set printer type. Only the first 6 characters of the string are significant, no upper-case letters may be included, and "print", "print " or null string are the wildcards.

Note: the N S and T parameters are frozen after the operating system has issued the ; command. Therefore they have to be set using the OPEN string. Any attempt to change these three parameters afterwards will be ignored.

## 6.15 TEXT: the front-end printer driver

### 6.15.1 Introduction

The TEXT: driver provides a front-end service to the printer driver.

### 6.15.2 Description

TEXT: acts as a filter for output to the printer driver. It performs the requisite character and string transformations on data to be printed to meet the input requirements of the selected printer device. This means that applications can generate output independent of the actual printer being driven.

It provides a screen dump facility, enabling the user or applications program to output the text contents of the screen to the currently selected printer.

The TEXT: driver also provides a facility to highlight printed output by selecting a highlighting mode such as bold or *italic*, as well as providing a function to set microspacing offsets (gaps between characters) on printers which support the facility.

Only one user at a time can be supported, so it is important that an application closes TEXT: when its services are no longer required.

#### *Printer types*

TEXT: supports three standard printer types, the Epson FX80, the Brother M1409 and the Diablo 630.

In addition, TEXT: supports the printer type **USER**, which allows users to set up their own printer driver (from the Configure module).

### 6.15.3 TEXT: reason codes

The TEXT: device driver supports a subset of the standard device driver reason codes and an additional three private reason codes.

#### *Standard reason codes*

The TEXT: driver supports the following standard reason codes:

DVOPN	open the TEXT: driver
DVCLS	close the TEXT: driver
DVBPT	output a character to the TEXT: driver

#### *Private reason codes*

The TEXT: driver supports the following private reason codes:

TXHLT	select a highlight mode
TXHMI	perform microspacing
TXDMP	perform a textual screen dump

#### **6.15.4 The TEXT: device driver functional interface**

The TEXT: device driver presents a semi-standard device driver functional interface to the outside world.

The standard method by which errors are returned is for the device driver function to exit with the carry flag set, an error number in X and a null terminated error message referenced by BHA.

The TEXT: driver can return the following error codes:

bpterr	(03)	printer failed during bput
prbusy	(05)	couldn't open printer
prclos	(07)	couldn't close printer
ldpfail	(11)	couldn't allocate local direct page
badrsn	(13)	bad reason code
trfull	(15)	character translation table full
errsvr	(17)	context save/restore failed
badhno	(19)	bad highlight number
badhgh	(21)	bad highlight string
badbar	(23)	bad sub-string separator
comitm	(25)	bad item in command string
comsep	(27)	bad separator in command string
brange	(29)	byte item out of range 0-255
comlmg	(31)	command string too long
control	(33)	control code encountered within ASCII item
badhmi	(35)	error in HMI string

Error numbers 21 to 35 refer to errors in strings set up in Configure which define the highlight commands etc. These error codes will be returned on OPENING the driver.

## DVOPN open the TEXT: driver

**Action:** Opens the TEXT: device driver. The TEXT: driver must be opened in OUTPUT mode. This operation must be performed before a file can be printed. The function is called with a pointer to the drivers name in BHA and OPENOUT (&80) in register Y.

**On entry:** BHA -> "TEXT"  
Y = &80 (OPENOUT)

**On exit:** C = 0 open succeeded  
Y = 16 bit handle  
C = 1 open failed  
X = error code  
BHA -> error message

## DVCLS close the TEXT: driver

**Action** Closes the TEXT: driver. This function should be called when an application no longer requires the services of the TEXT: driver. This function is invoked with the TEXT: driver's associated 16 bit handle in Y.

**On entry:** Y = 16 bit handle for text

**On exit:** C = 0 TEXT: was closed OK

C = 1 failed to close TEXT: properly

X = error code

BHA -> error message

## DVBPT    output a character to the TEXT: driver

**Action:** This function writes the byte value supplied in A to the printer driver, performing any necessary translations. The function is called with the TEXT: driver's associated 16 bit handle in Y.

**On entry:**    Y = text's handle  
                  A = character (0-255)

**On exit:**    C = 0 success  
                  C = 1 failed to output byte  
                  X = error code  
                  BHA -> error message

## TXHLT select highlight mode

**Action:** This function sets the current method of highlighting text. Called with the method in A and text's handle in Y.

**On entry:** Y = 16 bit handle  
A = highlight number  
    1 - underlined  
    2 - bold  
    3 - superscript  
    4 - subscript  
    5 - italic  
    6,7,8 - reserved

**On exit:** C = 0 highlight mode selected OK  
C = 1 failed to select highlight mode  
X = error code  
BHA -> error message

## **TXHMI** perform microspacing

**Action:** This function sets microspacing offsets on the currently selected printer if supported. Called with task's handle in Y and required spacing in A. Spacing is set in units of 120th of an inch.

**On entry:** Y = 16 bit TEXT: handle  
A = required spacing

**On exit:** C = 0 spacing set OK  
C = 1 failed to set spacing  
X = error code  
BHA -> error string

## **TXDMP** perform a textual screen dump

**Action:** This function dumps current screen text to the currently selected printer. Called with text's 16 bit handle in Y.

**On entry:** Y = 16 bit text handle

**On exit:** C = 0 dump performed OK

C = 1 dump failed

X = error code

BHA -> error code

## **6.16 GRAPHICS:**

### **6.16.1 Introduction**

The GRAPHICS: driver provides the facility to dump the contents of the screen to the currently selected printer.

### **6.16.2 Description**

The GRAPHICS: module implements a very restricted form of driver. In fact it only exists as such in order to inter-work with the Configure utility.

The screen dump is performed every time the GRAPHICS: driver is opened.

To perform a dump from BASIC, type:

```
CLOSE#OPENOUT"GRAPHICS::"  
IF @ERC%<>0 THEN P. $@ERMSG%
```

### **6.16.3 Printer types**

The GRAPHICS: driver is printer independent. Printers are driven via the Configure utility. The graphics menu displayed by Configure allows the user to select one of two pre-configured printers or alternatively the user can specify a printer configuration of their own. GRAPHICS: supports two standard printer types, the Epson FX80 and the Brother M1409.

## 6.17 Introduction to the terminal session software

### 6.17.1 Introduction

The sections that follow describe the various modules which make up the Terminal Session Software. This section attempts to illustrate the role of these modules within the framework of terminal session handling and the overall functionality provided.

### 6.17.2 Description

The terminal session software provides a terminal session service to the user. Facilities are provided for the emulation of different terminal types, keyboard or file input, direct or dialup connections, a phone book with auto-dial capability and multiple terminal sessions.

The terminal session software consists of the following modules:

<b>Phone</b>	Implements the telephone directory service along with the attendant user-interface.
<b>TShell</b>	The control program for an individual terminal session
<b>Keypage</b>	Handles the core terminal functions
<b>Emulation Modules</b>	These modules carry out the necessary data stream transformation to enable Keypage to emulate specific terminal types.
<b>Low-Level Drivers</b>	These modules handle data I/O on specific external port I/O devices. There is always a line driver and there may be an optional printer driver.

The user invokes a session via the Phone module.

The Phone module invokes a TShell process to run the terminal session.

TShell opens the line driver, Keypage and the emulation module and establishes a call. TShell runs the terminal session via a set of services provided by the Keypage module.

Keypage handles input character streams from files and strings supplied by TShell and from the line driver. TShell is responsible for handling any user interaction with the terminal session.

Keypage outputs data, after performing any required transformations, to either the screen, the line driver or both. In addition Keypage can be instructed to save a copy of the output to a session spooling buffer.

Keypage can be instructed to make a hard copy of the progress of a terminal session to a printer driver specified by TShell.

The user can have more than one terminal session active at one time, the number being limited by the available external I/O channels, (two in the case of the current system). Separate instances of the Keypage, TShell and the associated Emulation modules are created for each session.

TShell is a coroutine instance of BASIC running the TShell program, whereas Keypage and the emulation modules are device drivers.

### 6.17.3 The Phone module

The phone module provides the top-level user interface to the terminal session software. The user can add phone-book entries or invoke terminal sessions from the phone module blue screen menus.

When the user invokes a terminal session, Phone checks to see if the session is to be conducted over the Modem port. If it is, Phone checks to see if a phone number has been specified. If it has, Phone dials the requested number, otherwise the user is prompted for it.

After the call has been established, Phone initiates a TShell process and passes control to it.

The Phone module is not covered further in the *Systems Manual*. See the OEM Developer's Manual for more information.

## 6.17.4 TShell - the terminal shell

### *TShell's responsibilities*

TShell provides the blue-screen user interface during the terminal session, allowing the user, for example, to save a terminal screen, or terminate the call.

TShell controls the basic operation of the terminal session software during the course of the session.

It invokes the rest of the modules required for the session, (i.e. Keypage, the required emulation module and the required line driver).

In the case of modem and ecpmodem, TShell is responsible for establishing the call. (i.e dialling the number).

TShell is also responsible for executing any auto-logon sequence specified in the call's directory.

### *TShell's actions*

After initialisation, TShell passes a pointer to a buffer to Phone. Phone places a (possibly modified) copy of the relevant directory entry for the call being initiated into the buffer before returning it to TShell.

TShell checks that the specified emulation module exists before initiating the call.

TShell examines the directory entry to determine which I/O device is to be used and opens it.

After a call has been successfully established across the modem port, or in the case of RS423 immediately after the driver has been opened, TShell opens the required emulation module for the session and then opens Keypage, passing it the handles of the line driver and emulation module.

TShell invokes Keypage which handles the terminal emulation. Control is passed back to TShell when the user presses a function key, (except F6), or when the call terminates, (e.g. carrier lost on the modem). TShell notes the event which caused Keypage to terminate and acts on it.

If requested by the user, TShell can open a printer driver and pass its handle to Keypage requesting a hard copy of the terminal session. This can occur at any time during the session.

All data to TShell originating from the line or the keyboard during the session, arrive via the Keypage module. All data to be output to the screen or on the line from TShell during the session are also routed via Keypage.

When the session is terminated, TShell closes down the Keypage, emulation and line driver processes associated with it.

## 6.17.5 Keypage

Keypage takes input data from the keyboard, a specified file or string and sends them to the screen and/or line and/or printer via the assigned emulation module, as instructed by TShell.

Keypage reads data from the assigned line driver, (either RS423, MODEM or ECPMODEM), and sends them via the assigned emulation module to the screen and/or printer.

After opening and initialising Keypage, TShell can instruct it to run. Whilst running, Keypage handles terminal emulation using the Emulation module assigned by TShell. Keypage directs byte streams from various sources to the emulation module which carries out any required data translation and routes output to the screen and/or line.

If a function key or a CTRL-function key is pressed when Keypage is running, Keypage does not pass it to the emulation module. In the cases of all function keys except F6, Keypage suspends and passes control and the function key back to TShell. Tshell is responsible for deciding what to do with it. Function key F6 is a special case. F6 followed by a letter a to p causes Keypage to send an associated string, specified in the Configure module, to the emulation module.

Keypage provides functions to transmit a specified file to the emulation module as though it had been typed in from the keyboard.

Keypage provides a function to send a supplied string to the emulation module. This function is used by TShell to send auto-logon data.

Keypage provides functions to implement session spooling. This involves copying all screen output into a spool buffer, supplied by TShell. On instruction from TShell, Keypage copies the contents of the buffer to a file. This facility enables users to record the screen output from a session for future reference.

### **6.17.6 The emulation modules**

The Emulation Modules are provided to customise the actions of Keypage to make it conform to a specific terminal type.

Keypage knows nothing about the nature of the Emulation module. It is assigned by TShell. Keypage communicates with all emulation modules in the same way, making it terminal independent.

Emulation modules provide a similar interface to normal device drivers and share certain calls.

Emulation modules provides two special services to Keypage:

1. process a character from the keyboard
2. process a character from the line

Both routines process a single character, supplied by Keypage, and return a pointer to a block of memory. The returned block contains data to be sent to the screen (if any) and data to be sent to the line (if any). It is the responsibility of Keypage to output any data to the screen and/or line driver.

An emulation module exists for each terminal Communicator is capable of supporting. Currently these are:

**TELETYPE, BBC, ANSI/VT100 and VIDEOTEX.**

### **6.17.7 The low-level drivers**

The low-level components of the session spooling software provide the access to Communicator's external I/O devices

There are two external I/O devices used for terminal sessions:

- |                       |  |
|-----------------------|--|
| <b>The Modem</b>      | This is supported by the MODEM driver and the ECPMODEM driver. |
| <b>The RS423 port</b> | This is supported by the RS423 driver.                         |

Keypage has no knowledge of the line driver it is using.

In addition to the line drivers, Keypage may use a printer driver. The printer driver is responsible for supporting Communicator's external printer port. Keypage accesses the printer driver when instructed to produce hard copy of a terminal session by TShell.

## 6.18 KEYPAGE: the terminal kernel

### 6.18.1 Introduction

There is only one Communicator terminal program, which consists of two modules - the terminal shell called TSHELL and the terminal kernel called KEYPAGE.

### 6.18.2 Description

Keypage is customised for varying terminal emulations such as VT100 and videotex through emulation modules. See the next section.

In addition to terminal emulation, the user can direct that a copy of all screen output be saved in a spool buffer. The contents of the spool buffer can be saved to a user specified file, permitting long-term storage and retrieval of screen information.

The KEYPAGE: module has 16 f6 key commands associated with it. Commands a to p are configurable and each has a string argument - this is the string to be sent when the user presses function key f6 and a - p. The commands are set up by the user via the Configure module and the commands are sent by the MOS when KEYPAGE is opened. This is only done once. Changing the commands' associated strings has no effect until KEYPAGE is opened again.

#### *The terminal shell*

The terminal shell handles all operations where errors can occur, such as

##### a) Interaction with the user

- getting file names
- issuing error messages

##### b) Opening files and device drivers

- opening channel to emulator
- opening channel to line
- opening channel to printer
- opening channel to kernel
- send emulator handle to kernel
- send line handle to kernel
- send printer handle to kernel
- send file handle to kernel

**The terminal kernel**

The terminal kernel handles I/O, ie

- It reads data from the keyboard or a file or a string and sends it via the current emulator to the screen and/or printer and/or line (RS423 or MODEM), as instructed by the terminal shell. No f keys or CTRL f keys are sent through the emulators, but they are sent to the terminal shell which then decides what to do. It may, for example, open a file or a printer driver, or may close all files and shut down the kernel. SHIFT f keys are sent through the emulators.
- It reads data from the line (RS423 or MODEM) and sends it via the current emulator to the screen and/or printer, as instructed by the terminal shell.

The KEYPAGE: driver need not be opened directly by the user - it is opened by TSHELL or Configure.

Keypage supports the following reason codes:

KNopen	open the terminal kernel and get a handle
KNclose	close the terminal kernel
KNline	select the line driver to be used by the kernel
KNeml	select terminal emulator to be used by the kernel
KNprint	select simultaneous printing
KNstring	force terminal kernel to transmit a string
KNfile	force terminal kernel to transmit a file
KNpage	force terminal to transmit a teletext file
KNrun	initial entry point to the terminal kernel
KNsclaim	claim memory buffers and set spooling ON
KNsreset	reset memory buffers & reset spool-buffer-full state
KNssstate	set specified spool state to ON/OFF
KNssave	save spool buffer contents to specified file handle
KNsclose	release session spool buffers and turn spooling OFF
KNsread	return current spool status

### 6.18.3 Keypage operation

Keypage is normally accessed via TShell.

#### *Opening and initialising Keypage*

Before a terminal session can commence, an instance of Keypage must be created for it. A Keypage Driver is obtained by opening Keypage and obtaining a handle for an individual Keypage incarnation. This is achieved by calling the standard driver function, OPOPN, supplying the parameter string "KEYPAGE:" or referencing the Keypage module directly with the KNopen reason code. Either method results in a 16 bit handle being returned to the caller.

Prior to invoking Keypage, the controlling coroutine, (usually TShell), must supply it with the handles of the line and emulation module drivers which are to be used during the terminal session. The functions KNline and KNemul are provided for this purpose.

Once opened and initialised, Keypage can be invoked to perform the required terminal handling tasks.

#### *Normal terminal handling*

To perform terminal handling, TShell invokes Keypage via the KNrun reason code. Control is passed to Keypage which processes the input character streams from the keyboard and line driver, performs the required character translation and output the results on the specified output streams, (screen, line driver and sometimes printer driver).

Keypage uses the supplied emulation module to perform the required transformation on the input streams. Input stream data is passed on a single character basis to the emulation module. The character is processed and a pointer to a result block is returned to Keypage in BHA. The result block contains character streams for Keypage to send to the screen and the line driver. The character streams may be zero length, (i.e. send no data).

Keypage runs until a termination event occurs. A termination event can be caused by the user pressing a function key or the call terminating (e.g. loss of carrier on the modem). All function keys except F6 cause a termination event. All CTRL-function keys including CTRL-F6 cause a termination event. When a termination event occurs, Keypage suspends itself and returns control to TShell along with a termination code. If termination was caused by a function key, the keycode is returned, otherwise the termination code represents a Keypage error number. It is TShell's responsibility to handle error and function key termination events.

#### *File transmission*

Keypage provides two functions for file transmission, KNfile and KNpage. KNfile is used to transmit normal text files and KNpage is used to transmit Teletex pages. The functions are invoked with a handle referencing the file to be transmitted. Keypage take input data from the file as if it were originating from the keyboard.

Keypage exits from file transmission when a call error occurs, a function key is pressed or the file terminates. Keypage suspends itself and passes a termination code back to TShell.

#### *String transmission*

Keypage can be made to process a specific string by invoking it with the KNstring reason code and a pointer to the string in BHA.

Keypage treats the string as though it originates from the keyboard. When the string has been transmitted, it carries on with normal terminal handling, (ie, as though it had been invoked with KNrun).

#### *Session printing*

Keypage can provide a hard copy of the terminal session. To invoke this function, Keypage should be called with the KNprint reason code, a handle to a printer driver in Y and a non-zero value in A.

To switch printing off, the same procedure should be followed but with A set to zero.

***Session spooling***

Session spooling permits all screen output to be copied to a spool file.

Keypage provides a number of functions to implement session spooling.

To start session spooling, TShell invokes Keypage with the reason code **KNsclaim**. This instructs Keypage to claim a spool buffer from the memory manager and set spooling to ON. From this point on all data sent by Keypage to the screen will also be copied to the spool buffer.

If Keypage is running and the spool buffer becomes full, it suspends itself and passes control back to TShell, along with a buffer-full termination code. TShell will then, if it so wishes, invoke Keypage with the reason code **KNssave** along with a 16 bit file handle in HA. This causes Keypage to append the contents of the spool buffer onto the specified file.

The Keypage reason code **KNsstate** is used by TShell to turn session spooling ON and OFF.

The reason code **KNsreset** causes Keypage to reset the spool-buffer-full state and the spool buffer pointer.

The **KNsread** reason code causes Keypage to return information on the size of the spool buffer, the number of characters spooled to the buffer so far, and the current spool state.

To end session spooling, TShell calls Keypage with the reason code **KSsclose**. Keypage turns session spooling OFF and releases the spool buffer back to free memory.

#### 6.18.4 Keypage termination return codes

When Keypage is running and it receives a termination event, it suspends itself and returns control to TShell along with a 16 bit termination word in A. If the lower byte of the word is non-zero, the code returned is the value of the function key that caused the termination event. If the lower byte of the word is zero, the code returned reflects the error condition that caused the termination event.

Keypage returns the following error code values:

&0000	End of file. Returned when Keypage has finished processing an input file after being invoked with KNfile or KNpage.
&0100	Lost carrier. Returned when Keypage is notified by the line driver that carrier has been lost.
&0200	Timeout. Returned when Keypage is notified by the line driver that a modem timeout has occurred.
&0300	Modem Error. Returned when the line driver informs Keypage that a modem error has occurred.
&0400	Buffer Full. Returned by Keypage when the session spooling buffer becomes full.

### 6.18.5 Application Escape

Application escape provides access for user developed applications to the built-in system phone & terminal facilities. This enables programmers to incorporate their own applications into on-line terminal sessions.

Application escape is triggered either by the F8 function key being pressed, ("Leave" from the terminal module), or on answering a call whilst in auto-answer mode. When either of these events occur TShell searches for the module "PhoneEsc". PhoneEsc contains the users application code. If TShell fails to find a module PhoneEsc, it continues as if an invalid function key has been pressed. If PhoneEsc is present, it is invoked by TShell and passes the following information:

- The handle for the communications interface
- The handle for the emulation module
- The handle for the printer driver

PhoneEsc takes over operation of the call from TShell if it so wishes. When PhoneEsc returns control, TShell may continue the call as if nothing has occurred or close it down if necessary. PhoneEsc is responsible for leaving the screen in valid state on exit.

If Keypage is sending a file when F8 is pressed, PhoneEsc will not be invoked.

If Keypage has session spooling turned ON, capture of data will be suspended until control is returned to TShell.

If the user presses F4, (end call), or any Tshell error condition occurs, then any active PhoneEsc coroutine will be killed.

#### *Invoking PhoneEsc*

When the PhoneEsc module is invoked, it is passed a pointer to a control block in BHA. The control block takes the format:

CB+00	= status
CB+01	= number of handles
CB+02	= text pointer
CB+05	= line handle
CB+07	= emulation handle
CB+09	= printer handle

**status** is a one byte field indicating who invoked PhoneEsc.

0 = invoked by user  
1 = invoked by auto-answer

**number of handles** is a one byte field containing the number of handles in the control block and in this version of the software is always three.

**text pointer** is a three byte field containing a pointer to a return terminated null message. PhoneEsc can return a pointer to a message which will be printed out when TShell closes down PhoneEsc and the call.

**line handle** is a two byte field containing the 16-bit handle of the current line driver.

**emulation handle** is a two byte field containing the 16-bit handle of the current emulation module.

**printer handle** is a two byte field containing the 16-bit handle of the current printer driver.

***Exiting PhoneEsc***

On exit from PhoneEsc control is passed back to TShell. If the C flag is SET on return, TShell assumes that an error has occurred, and BHA is taken to point to a return terminated error message.

If the C flag is CLEAR on return, TShell assumes that BHA points to a return control block. The control block takes the format:

CB+00	= status
CB+01	= number of handles
CB+02	= text pointer
CB+05	= line handle
CB+07	= emulation handle
CB+09	= printer handle
<b>status</b>	is a one byte field indicating the actions which TShell should take. 0 = kill PhoneEsc coroutine and continue 1 = don't kill PhoneEsc, but continue 2 = kill PhoneEsc and close down TShell/Keypage
<b>number of handles</b>	A one byte field which should equal the number of handles on entry. If TShell is returned an incorrect number of handles PhoneEsc is killed and TShell and Keypage are closed down.
<b>text pointer</b>	May have been modified by PhoneEsc.
<b>line handle</b> <b>emulation handle</b> <b>printer handle</b>	These may have been changed by PhoneEsc. If the line or emulation handle is zero, TShell will close down the call. If any of the handles are odd, they are taken to be invalid, TShell will kill PhoneEsc and close itself and Keypage down. If the handles are valid, they will replace the handles passed to PhoneEsc on entry.

***Error messages***

If TShell closes down for any reason whilst handling an Application Escape, the user will be informed as follows:

Data call terminated due to Application Escape closing down <message returned by PhoneEsc>

PhoneEsc requests to be killed and to close down TShell.

Data call terminated due to an error in the Application Escape  
(Error code: &xx)  
<message returned by PhoneEsc>

TShell receives invalid data from PhoneEsc and decides to close down.

Error code is returned to allow developers to track down problems with their code. Errors are bit-flagged so that the code can indicate multiple errors.

<b>bit0</b>	An invalid status value was returned.
<b>bit1</b>	The parameter reporting the number of handles had an incorrect value. For the current implementation, this value should be the same number on exit as on entry and should be three.
<b>bit2</b>	An invalid line handle was returned (either 0 or odd).
<b>bit3</b>	An invalid emulation handle was returned (either 0 or odd).
<b>bit4</b>	An invalid printer handle was returned, i.e. odd.
<b>bits5,6,7</b>	reserved for future use.

## *Chapter 6*

### ***Responsibilities of the PhoneEsc module***

When the PhoneEsc module returns control to TShell it must ensure that:

1. Tshell's screen and context have been preserved.
2. Any memory allocated by PhoneEsc is deallocated.
3. Any device drivers or files opened by PhoneEsc are closed.

All of this functionality should be placed in a subroutine which can be invoked via the coroutine reason code ENKILL. As this code will be invoked as a subroutine by the MOS, it should be terminated by an RTL instruction.

### **6.18.6 Keypage functions**

The reason codes described on the following pages are supported.

Note: all entries should be called in word mode for both XY and AM.

# KNopen

To open the terminal kernel and get a handle

On entry: X = @DVOPN%

Y = opentype (&40 &80 or &C0)

On exit: Y = handle

Y = 0 failed to open

# KNclose

To close the kernel

On entry: Y = @DVCLS%

On exit: C = 1 failed to close

# KNline

Required for every terminal - must be called before main entry

**On entry:** Y = handle of open terminal kernel

X = &80

A = handle of line driver (eg RS423 or MODEM) to be used by kernel

**On exit:** C = 1 error

# KNeml

Required for every terminal - must be called before main entry

**On entry:** Y = handle of open terminal kernel  
X = &82  
A = handle of emulator to be used by kernel

**On exit:** C = 1 error

# KNprint

Only used if simultaneous print needed - may be called any time

**On entry:** Y = handle of open terminal kernel

X = &8A

A = handle of printer channel to be used for simultaneous print

A = 0 terminate simultaneous printing (channel closed by caller)

**On exit:** C = 1 error

# KNstring

To get terminal kernel to transmit a string

**On entry:** Y = handle of open terminal kernel

X = &86

BHA = pointer to string to send.

**On exit:** only exits if user presses an f key which TSHELL needs to process, or if the carrier disappears, or if one of a number of other errors occurs

Y = undefined

X = undefined

A = code of character which caused us to exit (f key) or an error code

This entry point is provided only for use when the kernel is first started up, and a "LOGON" string is to be transmitted.

String is transmitted with a limited amount of GSREAD-type escape processing:

<char>	is sent as ctrl/<char>
!	is sent as !
?	is sent as <delete>

# KNfile

To get terminal kernel to transmit a file

(Should not be called if KEYPAGE is already sending a file or a string.)

**On entry:** Y = handle of open terminal kernel

X = &88

A = handle of file (already opened for input by caller)

**On exit:** only exits if user presses an f key which TSHELL needs to process, or if the carrier disappears, or if one of a number of other errors occurs.

In particular, it exits when the file has been sent, with an error code meaning "file finished".

Y = undefined

X = undefined

A = code of character which caused us to exit (f key) an error code

This entry point is provided for use when the kernel has exited because user pressed an f key requesting file transmission. If the file was successfully opened by TSHELL, then this entry is called to send it.

# KNpage

KNpage works as KNfile except that a flag is set such that:

1. any top-bit-set character  $x$  is converted to  $\&1B, (x \text{ AND } \&1F) \text{ OR } \&40$ . This is the ASCII code for  $<\text{ESCAPE}>$  followed by a number in the range  $\&40$  to  $\&5F$ . This converts Teletext colour codes to ESCAPE sequences.
2. transmission of the file is terminated after a maximum of 22 lines (a line is 40 characters, or less than 40 characters plus a carriage return, whichever comes first).

(KNpage should not be called if KEYPAGE is already sending a file or a string.)

**On entry:** Y = handle of open terminal kernel

X = &8C

A = handle of file (already opened for input by caller)

**On exit:** only exits if user presses an f key which TSHELL needs to process, or if the carrier disappears, or if one of a number of other errors occurs, for example, "file finished".

Y = undefined

X = undefined

A = code of character which caused us to exit (f key) or an error code

This entry point is provided for use when the kernel has exited because user pressed an f key requesting file transmission. If the file was successfully opened by TSHELL, then this entry is called to send it.

# KNrun

Main kernel entry point

**On entry:** Y = handle of open terminal kernel

X = &84

A = not important

**On exit:** only exits if user presses an f key which TSHELL needs to process, or if the carrier disappears, or if one of a number of other errors occurs, for example, "file finished".

Y = undefined

X = undefined

A = code of character which caused us to exit (f key) or an error code

This entry point is provided for use when TKernel has exited because user pressed an f key and TSHELL wishes the kernel to carry on where it left off. It is also used as the initial entry point on start up where no logon string is to be sent.

## KNsclaim session spooling code

Claim memory buffers and set spooling ON.

**On entry:** X = &8E

**On exit:** Y = undefined  
X = undefined

C = 0 buffers allocated  
C = 1 failed to claim memory buffers

## **KNsreset session spooling code**

Reset memory buffer pointers and reset spool-buffer-full state.  
Called by KNsclaim.

**On entry:** No conditions

**On exit:**    C = 0 done  
              C = 1 failed - no open spool buffer

## **KNsstate session spooling code**

Set specified spool state to ON/OFF.  
Called by KNsclaim.

**On entry:** HA = 0 turn state OFF  
HA < 0 turn state ON

**On exit:** C = 0 done  
C = 1 failed - no open spool buffer

## **KNSSAVE session spooling code**

Save buffer contents to a specified file handle.

**On entry:** HA = 16 bit file handle

**On exit:** C = 0 done - buffer written to handle

C = 1 failed

X = reason code

## **KNsclose session spooling code**

Release memory session spool buffers and turn spooling OFF.

**On entry:** no conditions

**On exit:** C = 0 done - all memory released

C = 1 failed

X = reason code

## **KNsread session spooling code**

Read current spool status.

**On entry:** No conditions

**On exit:** C = 0

HA = characters spooled so far

X = size of spool buffer

Y = current status word

C = 1 no current active spool

## **6.19 Terminal Emulation Modules**

### **6.19.1 Purpose**

There is only one Communicator terminal program, consisting of two modules, the terminal shell and the terminal kernel, called KEYPAGE (see the previous section). The terminal program is customised for different terminal emulations, such as VT100 and videotex, by means of emulation modules.

These modules handle the flow of characters between the Communicator keyboard, the Communicator screen and the communications link. However, they are not responsible by themselves for I/O; they simply create the necessary character streams and leave the terminal shell to handle the I/O.

The terminal shell searches for emulation modules at run time and does not need to know what modules will be available. This means that new emulations can be written without disturbing the terminal shell code.

Communicator software has four basic emulation modules - teletype, BBC, ANSI/VT100 and videotex.

### **6.19.2 Usage**

Emulation modules are connected with the terminal shell through their name. The emulation name is taken from the directory entry for the service. The names of emulation modules take the form of the character E. followed by the emulation name. The four Communicator modules are therefore named E.TTY, E.VT100, E.VIDEOTEX and E.BBC. No other modules in the Communicator should have names starting with E.

An emulation module will be activated when a call is made to a directory entry that requires that module. The module will be deactivated when that call is cleared.

Emulation modules are written in a similar way to Communicator device drivers and share certain calls.

The directory entry may contain just the emulation type, eg TTY, or may have the type followed by a string of options, eg TTY:N+;E-. The option characters are described under each emulation type below. They are sent to the emulation control stream after the emulation has been opened but before any data characters are processed.

### **6.19.3 Software Interface**

An emulation module is written as a Communicator device driver and implements calls with the following reason codes:

- ENINIT** overall initialisation call
- ENKILL** overall closedown call
- DVOPEN** open call - create an instance of the emulation module
- DVCLS** close call - destroy an instance of the emulation module
- DVCGT** read byte from control stream - read status of emulation module
- DVCPT** write byte to control stream - set operating modes for emulation module
- EMSERIN** process byte received from communications line
- EMKEYIN** process character typed at the keyboard

Note that the standard driver data stream read and write reason codes are not implemented. Calls with reason codes other than the ones listed above return with the carry flag set to indicate that an error has occurred.

#### 6.19.4 Software Call Details

This section describes the operation of the emulation module calls in detail.

##### ENINIT - overall initialisation call

Entry: X = ENINIT

Exit: C = 0      no error - module initialised and  
                  direct page allocated correctly  
or                C = 1      an error occurred  
                  X            error code

This should carry out any global initialisation the emulation module requires. It relates to the emulation module as a whole, not to any particular instance of the module. As the instances of the module will be independent, this call will normally just allocate global direct page for the emulation module. An error could be returned by this call in the unlikely event the allocation of global direct page memory failed.

None of the other reason code calls should succeed until ENINIT has succeeded. If ENINIT is called twice with no intervening ENKILL, it should be treated as an ENKILL followed by ENINIT.

##### ENKILL - overall closedown call

Entry: X = ENKILL

Exit: C = 0      no error - module killed  
or                C = 1      error - module had not been initialised  
                  X            error code

This call should undo whatever ENINIT does. In particular it should release any global direct page memory allocated. It relates to the emulation module as a whole, not to any particular instance of the module.

ENKILL should succeed even if there are instances of the module open. These should be implicitly closed and all memory associated with them released.

It is an error to call ENKILL for a module that has not had ENINIT called on it.

##### DVOPN - open call

Entry: X = DVOPN

Y = opentype (&40, &80 or &C0)  
BHA points to the "device" name

Exit: C = 0      driver opened without error  
                  Y = handle handle for this instance of the module  
or                C = 1      error - module not opened

This call creates a new instance of the emulation module. The module requests local direct page memory and a handle for this new instance from the OS using the standard DTALD routine. If any problems arise during the open, the handle and memory must be handed back to the OS using DTDCLS.

The emulation module may wish to allocate additional buffers for itself.

The remainder of the emulation module reason codes are called with the handle for the particular instance in the Y register.

The opentype has no significance for emulation modules.

**DVRST - reset**

**Entry:** X = DVRST  
           Y = handle

**Exit:** C = 0      no error - module reset  
 or        C = 1      error  
           X        error code

This reason code is specific to the handle in Y, not the entire module. It would be used by an application program to reset the module to the same state as it would be immediately after a DVOPN.

**DVCLS - close call**

**Entry:** X = DVCLS  
           Y = handle

**Exit:** C = 0      no error - module closed  
 or        C = 1      error - could not close driver  
           X        error code

This call closes an instance of the emulation module

The module must release any local direct page memory associated with the handle and then release the handle itself back to the OS.

**DVCGT - read byte from control stream**

**Entry:** X = DRVCGT  
           Y = handle

**Exit:** C = 0      no error - character read  
 A = character read  
 or        C = 1      error  
           X        error code

This call reads a byte from control stream. This call is not needed for all currently defined emulation types and always returns a CR (&0D) character.

**DVCPT - write byte to control stream**

**Entry:** X = DVCPT  
           Y = handle  
           A = character to write

**Exit:** C = 0      no error - character written  
 or        C = 1      error  
           X        error code

This call writes a byte to the emulation control stream. This sets operating modes for the emulation module. See individual module descriptions for defined control codes. New emulation modules should use the codes defined here if the function is identical. Otherwise, any character stream may be used.

The following mode commands are common to most emulations:

- N+** Set newline mode, ie transmit CRLF when user types CR.
- N-** Reset newline mode.
- L+** Set linefeed mode, ie perform CRLF when CR is received from communications line.
- L-** Reset linefeed mode.
- E+** Set echo mode, ie characters typed by the user are echoed to the screen. All printing characters should be echoed normally. It is up to the emulation code how it treats control characters. This has to be part of the emulation code, because the terminal shell has no knowledge of screen modes which would modify the display of the echoed characters (eg VT100 wraparound).  
Newline mode affects echo mode, in that CR typed should be echoed as CRLF if newline mode is on.
- E-** Turn off echo mode.
- R<string>** Some emulations may have the ability to respond with a character string to an ENQ or other character from the serial line. For example, a videotex emulation should respond to ENQ with the customer user id.  
<string> is a character string terminated by CR (&0D). The emulation should make its own copy of this.

Other operating modes are specific to particular emulations.

**EMSERIN - process byte from comms line**

**Entry:**    X = EMSERIN = &82  
              Y = handle  
              A = Character received from serial line

**Exit:**    C = 0       no error  
              BHA      pointer to result block  
 or           C = 1      error occurred  
              X          error code

This call and the keyboard input routine operate very similarly. They each process a single incoming character and return a pointer to a block of memory. This block of memory contains characters which are to be output to the screen and/or the communications line. The format of this block is given at the end of this section.

This call will return an error only if it encounters some insoluble internal problem (or for example an invalid handle). If it is presented with an unexpected sequence of characters (for example an invalid VT100 control sequence), it will do whatever it feels is most sensible without reporting an error.

The emulation module may reuse the result block when it is called again. However, the keyboard and serial line result blocks must always be separate areas of memory.

**EMKEYIN - process character from keyboard**

**Entry:**    X = EMKEYIN = &84  
              Y = handle  
              A = character received from keyboard

**Exit:**    C = 0       no error  
              BHA      pointer to result block  
 or           C = 1      error occurred  
              X          error code

This works identically to the serial line input call, with the exception that the input character has been typed on the Communicator keyboard rather than received from the serial line.

The emulation result block has the following format:

**High memory**

Line output data . . . .  
 Line output length  
 Screen output data . . . .  
 Screen output length

BHA → Offset of line data

**Low memory**

The "offset of line data", "screen output length" and "line output length" are all 16 bit words and can range from 0 to &FFFF. BHA points at the "offset of line data" word. BHA + "offset of line data" points at the "line output length" word.

The screen output data is a stream of characters to be output directly to the screen as soon as possible. The line output data is a stream of characters to be output to the communications line as soon as possible.

The serial line input and keyboard input calls are fairly independent in most emulations. There are exceptions to this - for example there are incoming VT100 control sequences which change the mode of the keyboard and therefore the characters that certain keys must transmit.

### **6.19.5 Teletype**

The Communicator teletype emulation is a literal emulation of the teletype specification.

Characters typed at the keyboard are transmitted onward to the serial line with no modification or filtering.

Only a restricted set of characters is accepted from the serial line for display on the screen. This set is:

BEL	&07	beeps
BS	&08	moves cursor backwards
LF	&0A	moves cursor down
CR	&0D	returns cursor to left edge of screen

Printable characters are &20 to &7E inclusive.

Note that the DEL character CHR(127) is not displayed on the screen. This is because many communications services which are expecting to be communicating with teletype devices use DEL as padding at the beginning of lines.

Teletype emulation implements echo, newline and linefeed modes. It has no responses.

See the table at the end of this section for keyboard mapping.

As with current emulation modules, the only error code ever returned in X is zero, the MOS default code.

### **6.19.6 BBC**

This emulation is the simplest to implement. All keyboard characters are passed on unchanged. Incoming characters from the serial line are passed to the screen unchanged. Note that this is not safe, in the sense that the remote service could create unexpected changes in the behaviour of Communicator.

The BBC emulation implements echo, newline and linefeed modes. It has no responses.

### **6.19.7 VT100**

The Communicator VT100 emulation supports the following VT100 features:

- 80 x 24 screen size
- Carriage return, line feed, bell, reverse line feed, tab
- Cursor up, down, right, left, home
- Direct cursor address
- Select G0/G1 character set
- Index, newline
- Save/restore cursor/attributes
- Erase from beginning of line
- Erase cursor line
- Erase from beginning of screen
- Erase screen
- Erase to end of screen
- Erase to end of line
- Reverse video
- UK, US and graphics character sets (partial)
- Set scrolling region
- Set/clear tab stops
- line feed/newline mode
- Application cursor key mode
- Relative origin mode
- Wraparound on/off
- Report cursor position
- Report status
- What are you?
- Reset terminal
- Local echo mode
- Erase character

The VT100 emulation implements echo, newline and linefeed modes. However, linefeed mode operates differently from normal, in that LF is translated to CRLF. It has two additional modes, settable through the control stream, which are:

- |    |  |
|----|--|
| W+ | set wraparound mode  |
| W- | resets wraparound mode - default   |
| #+ | use US character set (incoming &23 will be displayed as hash)            |
| #- | use UK character set (incoming &23 will be displayed as pound) - default |

### **6.19.8 Videotex**

This emulation places the Communicator screen in mode 7 on initialisation.

Keyboard characters are forwarded to the serial line unaltered, with the exception of the # and pound keys, which are recoded to their UK videotex equivalents (&5F and &23 respectively).

Incoming data from the serial line is "vetted" to ensure that it contains only valid videotex control sequences and then passed on to the mode 7 screen. In conjunction with the mode 7 screen driver, this gives full videotex terminal emulation.

The videotex emulation implements echo, newline and linefeed modes. It transmits in response to ENQ (&05).

### 6.19.9 Keyboard mapping table

This table describes how the special keys on the Communicator behave under the various emulations. Keys which create no output are marked with a dash (-). Notes are marked with the corresponding number.

Key	Teletype	BBC	VT100	Videotex
TAB	&09	&09	&09	&09
HOME/%	&1E	&1E	*3	&1E
INSERT/	-	-	-	-
COPY/EE	-	-	*3	-
Up arrow	&0B	&0B	Note 1	&0B
Left arrow	&08	&08	Note 1	&08
Right arrow	&09	&09	Note 1	&09
DEL/CE	&7F	&7F	&7F	&7F
Down arrow	&0A	&0A	Note 1	&0A

Note: VT100 cursor keys can transmit different codes depending on the VT100 "Cursor Key Mode". The values are:

#### Cursor key mode

Key	reset	set
Up	ESC [ A	ESC O A
Down	ESC [ B	ESC O B
Right	ESC [ C	ESC O C
Left	ESC [ D	ESC O D

Note that the middle character in the "set" mode is the alpha character O, not a zero.

Note: the HOME and COPY keys are used to simulate the use of the VT100 numeric keypad keys - (dash) and , (comma). They therefore transmit the following codes:

Key	Keypad numeric mode	Keypad application mode
HOME/% -	&2D	ESC O m
COPY/EE ,	&2C	ESC O l

Note 4: the VT100 PF keys are simulated using SHIFT Fx as follows:

VT100 key	Communicator key	Transmitted code
PF1	SHIFT F1	ESC O P
PF2	SHIFT F2	ESC O Q
PF3	SHIFT F3	ESC O R
PF4	SHIFT F4	ESC O S

In summary, the Communicator keys map onto VT100 numeric pad keys as follows:

Communicator	VT100 numeric keypad key
*/	. (period)
#=	ENTER
HOME/%	- (dash)
COPY/EE	, (comma)
SHIFT F1	PF1
SHIFT F2	PF2
SHIFT F3	PF3
SHIFT F4	PF4

## 6.20 RS423:

### 6.20.1 Introduction

The RS423 port is a bi-directional, asynchronous, serial communications port. The port uses a 6-way connector, with the following connections (pins numbered from the left when looking at the rear of the machine):

Pin 1	Transmit Data TXD	Output
Pin 2	Request to Send	RTS Output
Pin 3	Receive data RXD	Input
Pin 4	Clear to Send CTS	Input
Pin 5	Signal Ground (0V)	GND
Pin 6	+5V VCC	

The TXD line may be in one of two states (voltages) known as mark and space (corresponding to bit values 1 and 0). When no transmission is occurring, the line is held in the mark state. Each character is sent as a series of bits, which are sent at a predetermined rate (the Baud rate). A character always starts with a start bit (space), followed by the data bits, which are sent least significant bit first. The number of data bits in a character must be preset, and may range from 5 to 8. The data bits are followed by an optional parity bit, and a preset number of stop bits (marks). In addition to data transmission, the TXD line may transmit a special "break" signal; this consists of a continuous space-level output, and is therefore distinct from any data transmission.

Data reception on the RXD line is in the same format, and is independent of transmission.

The CTS input indicates when the remote machine is ready to accept data; CTS checking is built into hardware, which cannot transmit without CTS being active. If for any reason the remote machine cannot provide a suitable CTS signal, then it must be provided in the cabling.

The name RTS for the signal on pin 2 is confusing, as the meaning of this signal is protocol dependent - it is often used to mean "Ready to Receive"; see the section on transmission protocols below.

Communication is interrupt-driven; the hardware generates interrupts when its receive register becomes full (ie when a character has been received) and when its transmit register becomes empty (ie the hardware is ready to transmit another character.)

### **6.20.2 Hardware**

Communicator uses a 2641 ACI (see the Mullard components catalogue, volume 4 part 9, 1985). This chip provides its own timing signals, and also controls the RTS output directly. The 2641 is addressed in bank &43. Registers are as follows:

Address	Read	Write
430000	Receive data	Transmit data
430001	Status	-
430002	Mode (1 or 2)	Mode (1 or 2)
430003	Command	Command

Note that read/write operations to the mode registers are controlled by an internal pointer. This is reset to point to Mode 1 at power up or reset. It may also be reset by reading the command register. Thereafter, it alternates between the 2 registers after each read or write to them.

Cabling is as described in the Communicator service manual, at least for CTS/RTS protocol. Other protocols (see below) may require different cables (eg with RTS output looped back to CTS input) to ensure that the CTS input is held low. The hardware cannot transmit without CTS low, irrespective of protocol selected.

### 6.20.3 Device driver commands

Take care when issuing commands which change the data format, rate or protocol (D, H, P, S, or W); any data received before the correct format is set will be suspect, and any changes made will apply immediately, even to data already in the Tx buffer. If in doubt, flush the buffers as well, or better still issue these commands only from the Configure page or as options following the OPEN command, eg:

**A% = OPENIN "RS423: D9600"**

as receipt and transmission are not enabled until the OPEN string is fully processed.

Commands are as follows:

#### B<Decimal value> - Send Break

Causes a break (ie a continuous space-level signal) to be sent. The parameter specifies the duration of the break, in units of 100 mS. The break will be sent immediately, taking priority over any pending data for transmission. (To avoid data loss, break is sent from the next TxEl; therefore break cannot be sent in the absence of a CTS signal). The Tx buffer is flushed when break is sent. Rx interrupts remain enabled throughout the break.

A flag is set to indicate that break is being sent; its effect is hardware dependent. Tx interrupts must remain enabled throughout the break, and setting the flag prevents the Tx interrupt service routine from disabling transmission.

On completion of the break, transmission is enabled; as the Tx buffer has been flushed, the first Tx interrupt will normally shut down transmission. We must wait at least 1 bit time after cancelling the break before writing to the tx holding register. This is ensured by a short wait loop in the 2641 break code.

#### D<baud rate> - Set Data Rate

50|75|110|134.5|150|300|600|1200|1800|2000|2400|3600|4800|7200|9600|19200

The baud rate is controlled by an internal clock within the 2641. The above list represents all available rates on this chip.

#### Frx|tx|both - Flush buffers

Allows the user to flush either or both buffers.

#### Hnone|rts|cts|xon|xoff|dtr - Select Handshaking Protocol

This selects the protocol to be used. Note that rts/cts and dtr are different only in name.

#### I<hex size> - Set Input buffer size

This command attempts to allocate a new input (receive) buffer of the requested size. A new buffer cannot be allocated if a buffer already exists and is not empty.

The existing buffer (if any) is first killed, and then a buffer of the new size is allocated. If this fails, a new buffer of the original size is re-allocated.

Legal buffer sizes for the current buffer handlers range from 1 to &FFFF bytes, which must lie within a single bank. Note, however that the lower limit is increased to 4, because of the need for sensible levels (with hysteresis) for

sending XON and XOFF when in XON/XOFF protocol, and the upper limit is effectively &FF00, because the memory manager cannot allocate more bytes than this within a bank.

Remember that 2 input buffers are required: 1 for data and one for receive status. The I command therefore allocates 2 buffers of the size specified.

**O<hex size> - Set Output buffer size**

Analogous to the I command, but sets the output buffer size. Only a single output buffer is required.

**Pnone|odd|even - Select parity checking type**

Selects the parity type for both transmission and receipt.

**Qon|off - Set or Reset Quit mode**

By default, attempts to BPUT to a full buffer or BGET from an empty buffer will loop until either space/data becomes available or <ESC> is pressed. This is Qoff mode. In Qon mode, such attempts will return immediately, with carry set, and the user must decide what to do.

**S1|1.5|2 - Set number of stop bits**

Sets number of stop bits to be used as a terminator for each character.

**W5|6|7|8 - Set data Width**

Sets the number of data bits for transmission and receipt.

**X<hex value> - Direct write to hardware**

Provided as an aid to debugging, or for advanced users; this command allows all writeable hardware registers involved in the RS423 device to be modified directly. Takes a 3 byte hex parameter which is interpreted as follows:

B = AND mask for current register value

H = register number

A = EOR mask for current register value

Register numbers are as follows:

0 = 2641 command register

1 = 2641 Tx register

2 = 2641 Mode register 1

3 = 2641 Mode register 2

In general, the given register is read, ANDed with B, EORed with A and re-written. If B is zero, the effect is therefore to write the value in A to the specified register.

#### 6.20.4 Query commands

Commands which have an automatic status entry may be interrogated by writing "?Y" to the control channel (where Y is the command letter). The command letter, followed by the last parameter passed to that command will then be placed in the get status buffer. For example, if B% is the control handle, the last data rate set may be read by:

```
BPUT$ B%, "?D"  
PRINT GET$$ B%
```

The following commands currently support automatic status entries: D H I O P S W.

Other (non-automatic) query commands are used to obtain other types of status information from the driver. Currently only ?e is implemented.

?e writes the cumulative receive error status to the get status buffer, and then re-zeros that status. This indicates what, if any, receive errors occurred with the data characters read from the data buffer since the last ?e command (or the last receive buffer flush). Thus receive errors are maintained with the data bytes to which they refer, and may be checked as frequently as the application demands. ?e replies by writing "e" to the get status buffer, followed by none, any or all of the characters b, p, r, f, where:

- b indicates that a receive buffer overflow has occurred
- p indicates that a parity error has been detected
- r indicates that a receiver overrun has occurred
- f indicates that a framing error has occurred

### **6.20.5 Transmission protocols**

#### **CTS/RTS protocol**

The accepted use of this protocol differs from the literal meaning of the abbreviations (which would suggest a unidirectional protocol, in which RTS is raised when we have data to send).

RTS and CTS lines cross over in the cabling. Either side asserts RTS when it can accept data, and de-asserts it when its receive buffer is nearly full. Neither side will transmit when its CTS input (ie the RTS output from the remote machine) is OFF.

Hysteresis is built into the system by defining separate thresholds for turning RTS on and off. Currently, RTS is turned off when the receive buffer is 75% full, and turned on again when it is 50% empty.

RTS is always switched on when the channel is opened, and off when it is closed, independent of protocol.

#### **XON/XOFF protocol**

Either party in the link may transmit an XOFF character at any time. The computer receiving the XOFF must then cease transmission (except for protocol characters) until an XON is received. Thus each computer may be in either XON or XOFF state. On initialisation, the state must be XON. XON and XOFF should only be used as protocol characters; XOFFs will be filtered from the data stream before transmission, but spurious XONs will be allowed, as they are less disastrous.

XON and XOFF thresholds are defined as for CTS/RTS, and provide hysteresis.

#### **DTR**

DTR protocol is identical to CTS/RTS as described above; it simply uses a different name for the wires involved.

#### **No protocol**

There is no hand-shaking mechanism to control transmission; data transmission will always continue until the Tx buffer is empty. RTS is switched on when the channel is opened (to allow correct modem control) and will remain on until the channel is closed.

## 6.21 MODEM:

The modem driver module front-ends Communicator's internal modem. It supports the following features:

- Selectable baud rates (1200/75, 75/1200, 300/300);
- Tone and pulse dialling;
- Selectable dial pauses;
- Multiple V series interfaces (v23c, v23ce, v23l, v23le, v21o, v21a);
- Auto-answer (conforming to V25 auto-answer sequences);
- Selectable parity (odd, even, none);
- Flow control (XON/XOFF);
- Idle-line watchdog timeout.

The modem driver module uses the standard device driver interface. Calls to this module should therefore always be made with a reason code in X, a 16-bit handle in Y (except for OPEN), and the byte to be PUT (if any) in A. All the reason codes given in the introduction are supported.

The modem driver conforms to a finite-state model of modem behaviour. For more detail on this refer to the OEM Manual's Modem Applications Guide.

Remember that a single channel has separate data and control streams; these are indicated by separate reason codes to the device driver, but are achieved by using odd or even handles from BASIC etc. The MOS notes the value of bit 0 of the handle, and selects the appropriate reason code. Thus from BASIC, one might type:

```
A% = OPENUP "MODEM:" :REM A% will always be even
IF A% = 0 THEN PRINT $@ERMSG% :STOP:REM print error message and quit
B% = A% + 1 :REM B% is now the control handle
BPUT# B%, "B10" :REM send BREAK of 1s duration
BPUT# A%, "Hello World" :REM write to data channel
CLOSE# A%
```

### **6.21.1 Modem device driver commands**

The commands are as follows:

#### **A off|quiet|on - Auto-Answer**

This command controls the ring-detection monitor. The command may be issued at any time but takes effect only when the modem re-enters the idle state.

The on setting enables the ring detector; this sets a flag whenever ringing is detected, which may be polled via ?r, (see below), and also generates an audible signal.

The quiet setting disables the audible signal.

The off setting disables the ring detector.

#### **B <dec> - Send break**

Flush the transmit buffer, then send a break of <dec>\*100ms duration.

#### **C <string> - Call a number**

Dial a number using either tone, pulse or alt-pulse dialling. Characters allowed in string:

0123456789	Dial that digit
ABCD*#	Dial that digit (tone dialling only)
E	Send the external line access string (see 'E' command)
T	Select tone dialling
P	Select pulse dialling
-	Pause for 850ms (approximately)
:	Pause for 2s (approximately)
/	Pause for 4s (approximately)

If none of E, T or P is specified, the configured dialling method is used.

#### **D 1200/75, 75/1200, 300/303 - Set data rate**

Set the data rate used by the modem (options with slashes are rx/tx). Must be set in conjunction with the "M" command (see below).

#### **E <string> - External line access**

This command sets the external line access string. When an "E" is specified in a dial string, this string is substituted. The string may contain any characters which are legal in the "C" command, except for "E" itself. The string defaults to "9".

#### **F rx/tx/both - Flush buffer(s)**

Flush either the receive, the transmit or both receive and transmit buffers. All characters in the buffer(s) are thrown away.

**G off|1min|5min|15min|60min - Set watchdog timeout**

The watchdog causes idle lines to be dropped after the indicated period. A line is idle if it is not transmitting data and is not receiving valid data (i.e. data without framing or parity errors).

Once a call is established, the application should poll the modem state using the ?s command. If the state returns to idle before the call is terminated by the application then either the watchdog or the carrier monitor has 'fired', causing the line to drop. The application may determine which of these events has occurred using the ?d and ?g commands.

For auto-answer calls only, a watchdog is enforced of 50 seconds when the call is answered; if incoming carrier is not detected in this time, the call will be terminated. Once carrier is detected, the watchdog reverts to the setting selected by the application.

**H none, xon/xoff - Set handshake protocol**

Set which flow-control protocol should be used. When "none" is set, characters are sent and received freely. When "xon/xoff" is set, if the receive buffer has fewer than about 30 spaces left when a character is received, an XOFF character (&13) will be sent. If subsequently a character is removed from the receive buffer leaving more than about 60 spaces left, an XON character (&11) is sent.

If an XOFF character is received, the transmit process is halted until the reception of an XON character.

**I <hex> - Set size of input (receive) buffer**

Shuts down the current receive buffer and attempts to allocate a buffer of the specified size. If this fails, it attempts to allocate a buffer of the previous size. If this fails, it attempts to allocate a buffer of the default size (ie &100 bytes). If this fails, a fatal system error is generated.

If there are characters in the receive buffer when this command is received, the command is ignored.

This command should only be invoked when the modem is in the idle state. Buffer sizes should be established before 'C' or 'Koff' commands are issued , or after a 'Kon'.

**J 15ms|75ms|250ms|750ms|2500ms|voice - Set carrier detect time**

Sets the minimum time the carrier must be consistently detected by the modem hardware before the software accepts that it is "present". This provides improved accuracy of carrier detection, particularly when using noisy phone lines.

The optimum setting must be determined by experiment in each situation, and will depend on the quality of the available phone lines and the exact characteristics of the remote host.

The voice setting must be used for all voice calls. This disables carrier monitoring and avoids any danger of the modem dropping the line due to "carrier loss".

**K on/off - Hook**

Put the phone on or off hook. The phone is automatically taken off hook for a

call command ("C"), and put on hook when the channel is closed. This command makes it possible to dial a new number without closing the channel.

### L on/off - Loudspeaker

Turn the feedback to the internal loudspeaker on or off.

### M v23c, v23ce, v23t, v23te, v21o, v21a - Modem mode

Set the modem mode and V series to be used. Should be used in conjunction with the "D" command (see above). The command will only take effect when the "D" setting and the "M" setting are compatible. The valid combinations are as follows:-

D	M	Meaning
75/1200	v23c	V23 tones, full duplex "computer"
75/1200	v23ce	V23 tones, full duplex, equalised "computer"
1200/75	v23t	V23 tones, full duplex "terminal"
1200/75	v23te	V23 tones, full duplex, equalised "terminal"
300	v21o	V21 tones, full duplex, originate
300	v21a	V21 tones, full duplex, answer

### N pulse, tone - Set dialler type

Set the type of dialler used. This can be overridden in the dial string (see above).

### O <hex> - Set size of output (transmit) buffer

Shuts down the current transmit buffer and attempts to allocate a buffer of the specified size. If this fails, it attempts to allocate a buffer of the previous size. If this fails, it attempts to allocate a buffer of the default size (ie &100 bytes). If this fails, a fatal system error is generated.

If there are characters in the transmit buffer when this command is received, the command is ignored.

This command should only be invoked when the modem is in the d idle state. Buffer sizes should be established before 'C' or 'Koff' commands are issued , or after a 'Kon'.

### P odd/even/none - Set parity

Set the type of parity bit (if any). This command will only take effect when the settings of "P", "S" and "W" are compatible (see below).

### Q on/off - Set "quit" mode

This command controls what happens when BGET is called with an empty receive buffer or BPUT called with a full transmit buffer.

Initially the setting is "off". In this state, BGET will wait until a character is received or the user presses ESCAPE. If ESCAPE was pressed, a BRK is generated, otherwise the character is returned with carry clear. Similarly, BPUT will wait until there is room in the transmit buffer or the user presses ESCAPE, and again returns with carry clear or generates a BRK respectively.

If "quit" mode is turned "on", BGET will return immediately with carry set if there was no character in the receive buffer, and BPUT will return immediately with carry set if there is no room in the transmit buffer for the character. This allows the caller to continue with other tasks rather than having to wait.

## S 1/2 - Stop bits

Set the number of stop bits. This command will only take effect when the settings of "P", "S" and "W" are compatible (see below).

## T <str> - Send string of tone data

Sends DTMF data down the line. Characters allowed in string:-

0123456789ABCD*	Send tones for that digit
E	Send the external line access string
.	Pause for 850ms (approximately)
:	Pause for 2s (approximately)
/	Pause for 4s (approximately)

## W 7/8 - Word length

Set the number of data bits. This command will only take effect when the settings of "P", "S" and "W" are compatible. The valid combinations are as follows:

P	S	W
even	2	7
odd	2	7
even	1	7
odd	1	7
none	2	8
none	1	8
even	1	8
odd	1	8

## ? <command letter> - Read status

This reads a particular piece of status information as specified by the parameter. The information is read from the control stream after issuing this command.

Most statuses correspond with one of the other commands; in this class the following are currently provided (subject to modification):-

C	Last dialled string
D	Last specified data rate *
H	Handshake option
I	Input buffer size
K	Hook state
L	Loudspeaker state
M	Last modem mode set *
N	Dialler type
O	Output buffer size
P	Last parity setting *
Q	"Quit" state
S	Last stop bit setting *

**W Last word length setting \***

The options marked with a star always read back the last option set, although this may not have been set in the hardware yet, because the settings may not be compatible.

The other statuses are lower-case letters and are not in general connected with commands:

- d** - Read incoming carrier state
- e** - Read error status

This returns a string indicating which errors have occurred since last using ?e, as follows:

- b** - Receive buffer overrun
- p** - Parity error
- r** - Receiver overrun
- f** - Framing error

**p,r,f** only refer to bytes which have actually been read by the user since the last request for error status.

**b** will be set after the last character before the overrun has been read.

**g** - return the state of the watchdog

This command returns either gok or gfail according to whether the watchdog has 'fired' or not.

**r** - return ringing state

This command returns the status of the ring-detector and will cause either rtrue or rfalse to be placed in the CGET buffer.

If this command is issued when Aoff or when the modem is not idle, it will always return rfalse.

**s** - return modem state

Returns the current state of the modem as defined in the finite state model. Possible responses are:

- sidle** modem in idle state
- sdial** modem in dialling state
- sstart** modem in starting a call state
- sconnect** modem in connected state
- sanswer** modem in answering a call state

**t** - Read number of spaces in TX buffer (not yet implemented)

**x** - Read ACIA status register (not yet implemented)

**y** - Read modem status register (not yet implemented)

The status strings read consist of the command character, followed by the appropriate status. In the case of the command-related statuses, the returned status is in the same form as that in which it was specified.

For example after sending a "Lon" command (loudspeaker on), a ?L command is issued. The returned status string is "Lon".

## 6.22 ECPMODEM:

### 6.22.1 Introduction

In addition to the standard modem driver, Communicator is supplied with an Error Correcting Protocol modem driver implemented by the `ecpmodem` module. This facility allows users to communicate with services such as British Telecom's Vasscom.

The `ecpmodem` driver provides such facilities as:

- Selectable baud rates (1200/75, 75/1200, 300/300);
- Tone or pulse dialling support
- Selectable dial pauses
- Multiple V series interface support (`v23c`, `v23ce`, `v23t`, `v23te`, `v21o`, `v21a`)
- auto-answer (conforming to V25 auto-answer sequences)
- idle-line watchdog timeout

### 6.22.2 General description

The `ecpmodem` driver supports an asynchronous octet orientated communications protocol, providing facilities for data error-detection/retransmission and flow control. The protocol provides for full duplex operation with split-speed capability when required.

Transmit and receive data channels can operate in both character and block mode, independently of each other.

### 6.22.3 Character mode operation

In character mode any 8-bit code may be transmitted as data except the transmission control characters, (SOH, EOT, ACK, DLE and NAK). When operating in this mode, error-detection and retransmission is not supported.

### 6.22.4 Block mode operation

In block mode all data are embedded in message blocks bounded by control bytes and post-fixed with a 16-bit Cyclic Redundancy Checksum for data validation.

Block operation can take place in either Non-transparent or Transparent mode.

#### *Non-transparent block mode operation*

In this mode the data-link is maintained through the use of control character sequences. Any attempt by the application to transmit values corresponding to these control codes will result in protocol violation with attendant data loss.

#### *Transparent block mode operation*

This mode provides greater versatility in the range of data that can be transmitted. All data link control characters can be transmitted as data without implying any control meaning. Data link control characters transmitted by the `ecpmodem` driver are preceded by the DLE character. The receiver will only recognise characters as having a control significance if they immediately follow a DLE character in the data stream. On encountering a DLE character in the application's transmit data, the `ecpmodem` driver will prepend it with another DLE before sending it. The receiver recognises a DLE-DLE pair as having no control significance and passes up a single DLE character to the receiving application.

#### *Intermediate message blocks*

Message blocks can be subdivided into smaller intermediate message blocks for transmission. This provides for easier flow-control and on noisy lines increases data throughput. (it is less efficient to retransmit an entire message than a small part of it).

### **6.22.5 The driver interface**

The **ecpmodem** driver module uses the standard device driver interface. Calls to this module should therefore always be made with a reason code in X, a 16-bit handle in Y (except for OPEN), and the byte to be PUT (if any) in A. All the reason codes given in the introduction are supported.

The **ecpmodem** driver conforms to a finite-state model of modem behaviour. For more detail on this refer to the OEM Manual's Modem Applications Guide.

Remember that a single channel has separate data and control streams; these are indicated by separate reason codes to the device driver, but are achieved by using odd or even handles from BASIC etc. The MOS notes the value of bit 0 of the handle, and selects the appropriate reason code. Thus from BASIC, one might type:

```
A% = OPENUP "MODEM":  
IF A%<0 THEN PRINT $@ERMSG% : STOP:REM print error message and quit  
B% = A% + 1  
BPUT# A%, "Hello World"  
CLOSE# A%
```

### 6.22.6 Device driver commands

The commands are as follows:

#### A off|quiet|on - Auto-Answer

This command controls the ring-detection monitor. The command may be issued at any time but takes effect only when the modem re-enters the idle state.

The on setting enables the ring detector; this sets a flag whenever ringing is detected, which may be polled via ?r, (see below), and also generates an audible signal.

The quiet setting disables the audible signal.

The off setting disables the ring detector.

#### C <string> - Call a number

Dial a number using either tone, pulse or alt-pulse dialling. Characters allowed in string:

0123456789	Dial that digit
ABCD*#	Dial that digit (tone dialling only)
E	Send the external line access string (see 'E' command)
T	Select tone dialling
P	Select pulse dialling
-	Pause for 850ms (approximately)
:	Pause for 2s (approximately)
/	Pause for 4s (approximately)

If none of E, T or P is specified, the configured dialling method is used.

#### D 1200/75, 75/1200, 300 - Set data rate

Set the data rate used by the modem (options with slashes are rx/tx). Must be set in conjunction with the "M" command (see below).

#### E <string> - External line access

This command sets the external line access string. When an "E" is specified in a dial string, this string is substituted. The string may contain any characters which are legal in the "C" command, except for "E" itself. The string defaults to "9".

**F rx/tx/both - Flush buffer(s)**

Flush either the receive, the transmit or both receive and transmit buffers. All characters in the buffer(s) are thrown away.

**Frx** Flushes all released characters in the receive buffer, (i.e. all characters which are part of a message block whose checksum has been correctly received). Characters which are in a buffer, but are part of a message block or intermediate message block still only partly received, are not flushed.

**Ftx** Generates a "forward abort request" which will result in a transmit buffer being flushed and a forward abort generated next the modem requests a character from the buffer for transmission.

**Fboth** Implements both the above actions.

**G off|1min|5min|15min|60min - Set watchdog timeout**

The watchdog causes idle lines to be dropped after the indicated period. A line is idle if it is not transmitting data and is not receiving valid data (i.e. data without framing or parity errors).

Once a call is established, the application should poll the modem state using the ?s command. If the state returns to idle before the call is terminated by the application then either the watchdog or the carrier monitor has 'fired', causing the line to drop. The application may determine which of these events has occurred using the ?d and ?g commands.

For auto-answer calls only, the watchdog is enforced of 50 seconds when the call is answered; if incoming carrier is not detected in this time, the call will be terminated. Once carrier is detected, the watchdog reverts to the setting selected by the application.

**J 15ms|75ms|250ms|750ms|2500ms|voice - Set carrier detect time**

Sets the minimum time the carrier must be consistently detected by the modem hardware before the software accepts that it is "present". This provides improved accuracy of carrier detection, particularly when using noisy phone lines.

The optimum setting must be determined by experiment in each situation, and will depend on the quality of the available phone lines and the exact characteristics of the remote host.

The voice setting must be used for all voice calls. This disables carrier monitoring and avoids any danger of the modem dropping the line due to "carrier loss".

**K on/off - Hook**

Put the phone on or off hook. The phone is automatically taken off hook for a call command ("C"), and put on hook when the channel is closed. This command makes it possible to dial a new number without closing the channel.

**L on/off - Loudspeaker**

Turn the feedback to the internal loudspeaker on or off.

**M v23c, v23ce, v23t, v23te, v21o, v21a - Modem mode**

Set the modem mode and V series to be used. Should be used in conjunction with the "D" command (see above). The command will only take effect when the "D" setting and the "M" setting are compatible. The valid combinations are as follows:-

D	M	Meaning
75/1200	v23c	V23 tones, full duplex "computer"
75/1200	v23ce	V23 tones, full duplex, equalised "computer"
1200/75	v23t	V23 tones, full duplex "terminal"
1200/75	v23te	V23 tones, full duplex, equalised "terminal"
300	v21o	V21 tones, full duplex, originate
300	v21a	V21 tones, full duplex, answer

**N pulse, tone - Set dialler type**

Set the type of dialler used. This can be overridden in the dial string (see above).

**Q on/off - Set "quit" mode**

This command controls what happens when BGET is called with an empty receive buffer or BPUT called with a full transmit buffer.

Initially the setting is "off". In this state, BGET will wait until a character is received or the user presses ESCAPE. If ESCAPE was pressed a BRK is generated, otherwise the character is returned with carry clear. Similarly, BPUT will wait until there is room in the transmit buffer or the user presses ESCAPE, and again returns with carry clear or generates a BRK respectively.

If "quit" mode is turned "on", BGET will return immediately with carry set if there was no character in the receive buffer, and BPUT will return immediately with carry set if there is no room in the transmit buffer for the character. This allows the caller to continue with other tasks rather than having to wait.

**X n/t - Set transparency normal/transparent**

This command selects which block mode the ecpmodem driver will negotiate to transmit under.

In normal mode, it is forbidden for the application to attempt to send any characters that correspond to transmission control codes, (SOH, EOT, ACK, DLE, and NAK).

In transparent mode the application may send any 8-bit character.

## Y xxz - Set up negotiation parameters

This command allows the application to select the level of operation it wishes the cxdriver to negotiate for.

- xx** - selects the hex block mode, takes the values (0-3)  
0 = transmit and receive in character mode  
1 = transmit in block mode, receive in character mode  
2 = transmit in character mode, receive in block mode  
3 = transmit and receive in block mode
- y** - allow intermediate message blocks  
0 = do not allow intermediate message blocks  
1 = allow intermediate message blocks
- z** - receive buffer size (1-6)  
1 = 16 bytes maximum ITB block or 32 bytes maximum message block  
2 = 32 bytes maximum ITB block or 64 bytes maximum message block  
3 = 64 bytes maximum ITB block or 128 bytes maximum message block  
4 = 128 bytes maximum ITB block or 256 bytes maximum message block  
5 = 256 bytes maximum ITB block or 512 bytes maximum message block  
6 = 512 bytes maximum ITB block or 1024 bytes maximum message block

This command exists mainly for debugging and it is strongly recommended that "Y0316" should always be used - i.e. negotiating for the highest level of service that can be supported.

## ? <command letter> - Read status

This reads a particular piece of status information as specified by the parameter. The information is read from the control stream after issuing this command.

Most statuses correspond with one of the other commands; in this class the following are currently provided (subject to modification):-

C	Last dialled string
D	Last specified data rate *
K	Hook state
L	Loudspeaker state
M	Last modem mode set *
N	Dialler type
Q	"Quit" state

The options marked with a star always read back the last option set, although this may not have been set in the hardware yet, because the settings may not be compatible.

The other statuses are lower-case letters and are not in general connected with commands:

- b** - back channel protocol error status.  
Returns:
  - b ok** - no errors since last ?b enquiry
  - b error** - one or more protocol errors since last ?b enquiry
- d** - Read incoming carrier state
- f** - forward channel protocol error status  
Returns:
  - f ok** - no errors since last ?f enquiry
  - f error** - one or more protocol errors since last ?f enquiry

- g** - return the state of the watchdog

This command returns either gok or gfail according to whether the watchdog "fired" or not.

- n** - negotiation error status

This command returns the type of any negotiation errors that have occurred since the command was last invoked.

Returns:

n none - no errors have occurred since last ?n enquiry  
or any combination of:

n Tx-fail - negotiation transmission not acknowledged or  
negatively acknowledged for three attempts.

n Timeout - Didn't receive response to successfully  
transmitted negotiation request after 30 seconds.

n Invalid - Received invalid negotiation block.

n Unexpected - Received negotiation block when we weren't  
expecting one.

- p** - return the state of the protocol (i.e. have we negotiated successfully).

Returns:

p uninitialized - no successful negotiation has occurred

p pending - a negotiation is in progress

p completed - at least one negotiation has completed  
successfully (and none is pending).

- r** - return ringing state

This command returns the status of the ring-detector and will cause either rtrue or rfalse to be placed in the CGET buffer.

If this command is issued when Aoff or when the modem is not idle, it will always return rfalse.

- s** - return modem state

Returns the current state of the modem as defined in the finite state model.  
Possible responses are:

sidle	modem in idle state
sdial	modem in dialling state
sstart	modem in starting a call state
sconnect	modem in connected state
sanswer	modem in answering a call state

- t** - Read number of spaces in TX buffer (not yet implemented)

- x** - Read ACIA status register (not yet implemented)

- y** - Read modem status register (not yet implemented)

The status strings read consist of the command character, followed by the appropriate status. In the case of the command-related statuses, the returned status is in the same form as that in which it was specified.

For example after sending a "Lon" command (loudspeaker on), a ?L command is issued. The returned status string is "Lon".

**6.172**

## **7. Files**

### **7.1 Introduction**

The file control system provides functions to allow users and applications to access, read and modify the various filing systems available on Communicator.

Filing systems can be accessed via MOS star (\*) commands, BASIC file handling operations, MOS file handling COP calls and via low-level file driver reason codes.

The two standard filing systems available on Communicator are the RAM filing system and the NET filing system. There is an additional card filing system available on the Spectar-2 Briefcase Communicator.

### **7.1.1 Description**

The three filings systems available on Communicator are:

- |              |  |
|--------------|--|
| <b>NET:</b>  | The Econet Local Area Network filing system.   |
| <b>RAM:</b>  | The RAM based filing system.   |
| <b>CARD:</b> | The memory card based filing system available on the Spectar-2 briefcase Communicator. |

The various filing systems on Communicator can be accessed at four levels:

- By MOS star (\*) commands
- From BASIC file handling keywords.
- By file driver COP calls.
- By file module references with low-level file device reason codes.

The first three methods all map down onto the last one.

There is no concept of a currently selected file system, although MOS star (\*) commands default to the NET filing system if one is not specified in the command.

File system handlers are specialised forms of device drivers and are accessed from BASIC and ASSEMBLER in the same way as hardware drivers.

At the module level, file system drivers implement the extended device driver reason codes. See the Chapter on device drivers for more information. Not all of the available file systems support all of the possible reason codes.

From BASIC and assembler, files are opened and closed in the same way as device drivers. After a file has been opened it is referenced in following operations by the handle supplied by the open procedure.

### 7.1.2 MOS star (\*) file commands

There are eleven MOS star commands provided for user interaction with filing systems:

*SAVE	Save an area of memory to a file
*LOAD	Load a file into memory
*DELETE	Delete a file from the filing system
*RENAME	Rename a file
*INFO	Display information on a file
*EX	Display information on a filing system directory
*READFREE	Display information on available filing system freespace
*CDIR	Create a directory
*DIR	Enter a directory
*CAT	List the contents of a directory
*FORMAT	Format the filing system
*FREE	Return free space on filing system
*PROTECT	Make filing system read only

These commands default to the NET filing system. To access other filing systems with these commands, arguments should be preceded with the name of the required filing system.

For example:

\*CAT RAM:

Not all commands will work with all filing systems. The following table lists which commands can be used on individual filing systems. An "X" indicates that the command is supported.

	NET	RAM	CARD
SAVE	X	X	X
LOAD	X	X	X
DELETE	X	X	X
RENAME	X	X	X
INFO	X		
EX	X	X	X
ACCESS	X		X
READFREE	X		
CDIR	X		
DIR	X		
CAT	X	X	X
FORMAT			X
FREE			X
PROTECT			X

### **7.1.3 BASIC file commands**

In BASIC, files are accessed in the same way as devices. They are opened, read from, written to and closed. There are additional operations for manipulating file pointers and setting file attributes.

BASIC provides the following file handling commands:

OPENIN	Open existing file for input
OPENOUT	Create and open file for output
OPENUP	Open existing file for input and output
CLOSE#	Close a file
BPUT#	Write a byte to a file
BGET#	Return the next byte from a file
EOF#	Return current End Of File condition for a file
=EXT#	Return length of a file
EXT#=	Set length of a file
=PTR#	Return pointer to the current write position in a file
PTR#=	Set current write pointer in a file

Not all commands will work with all filing systems. The following table lists which commands can be used on individual filing systems. An "X" indicates that the command is supported.

	NET	RAM	CARD
OPENIN	X	X	X
OPENOUT	X	X	X
OPENUP	X	X	X
CLOSE#	X	X	X
BPUT#	X	X	X
BGET#	X	X	X
EOF#	X	X	X
=EXT#	X	X	X
EXT#=	X	X	
=PTR#	X	X	X
PTR#=	X	X	X

### 7.1.4 Filing system COP calls

The MOS provides the following COP calls for accessing the filing system:

OPOPN	Open a file
OPEND	Test for end of file
OPCLS	Close a file
OPGBT	Get a byte from a file
OPPB	Put a byte to a file
OPBGB	Get a block of bytes from a file
OPBPB	Put a block of bytes to a file
OPLOD	Load a file into memory
OPSAV	Save an area of memory to a file
OPRLE	Read load address, execute address and length of a file
OPWLE	Write load address and execute address of a file
OPRAT	Read a file's attributes
OPWAT	Write a file's attributes
OPRSP	Read a sequential file pointer
OPWSP	Write a sequential file pointer
OPRPL	Read physical length of a file
OPRLL	Write logical length of a file
OPWLL	Write logical length of a file
OPRCH	Read catalogue header and information
OPRFN	Read file/object information from a directory
OPDEL	Delete a named object
OPREN	Rename an object

*Chapter 7*

Not all calls will work with all filing systems. The following table lists which calls can be used on individual filing systems. An "X" indicates that the call is supported.

	NET	RAM	CARD
OPOPN	X	X	X
OPEND	X	X	X
OPCLS	X	X	X
OPGBT	X	X	X
OPPBFT	X	X	X
OPBGB	X		X
OPBPB	X		X
OPLOD	X		X
OPSAV	X		X
OPRLE	X		X
OPWLE	X		
OPRAT	X		X
OPWAT	X		X
OPRSP	X		X
OPWSP	X		X
OPRPL	X		X
OPRLL	X		X
OPWLL	X		
OPRCH	X		
OPRFN	X	X	X
OPDEL	X	X	X
OPREN	X	X	X

# OPOPN

open file

**Action:** This call opens a file for reading, writing, or update.

**On entry:** BHA points to the filename, terminated by CR (&0D).  
 Y = &40 for input  
 Y = &80 for output  
 Y = &C0 for update

**On exit:** C = 0 means that the file is open and Y = handle.  
 If C = 1 then the file failed to open, the error code is in X, and BHA points to a zero-terminated error message.  
 D preserved

**To call:** COP  
 EQUB @OPOPN% from BASIC  
 OPSYS £OPOPN from MASM

**Example:**

```

170 RSP# &10          \ Put processor in 16-bit XY mode.
175 WRD &10          \ Tell assembler that processor is in 16-bit XY mode.
180 COP               \ Execute COP call.
190 EQUB @OPBHA%      \ Make BHA point to following string.
200 EQUS "filename"   \ Filename here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &40          \ Open type in Y (OPENIN).
230 COP               \ Execute COP call.
240 EQUB @OPOPN%      \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                 \ File open, handle in Y.
900 .end%             \ End here.
910 RTL               \ Return from subroutine.

```

This example will attempt to open the file called "filename" for input.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPEND test for end of file

**Action:** This call will test an open file to see if the end of file has been reached.

**On entry:** Y contains the file handle returned by OPOPN.

**On exit:** If C = 0 then

if lsb A = 0 the end of file has not been reached.

if lsb A = 1 then the end of file has been reached.

Note: only the least significant bit of A (or HA) need be tested.

If C = 1 then the call failed.

X = error code (X = 0 means "Not implemented on this driver").

BHA points to a zero-terminated error message.

DAXY preserved

**To call:**

COP

EQUB @OPEND% from BASIC

OPSYS EOPEND from MASM

**Example:**

170 RSP# &10	\ Put processor in 16-bit XY mode.
175 WRD &10	\ Tell assembler that processor is in 16-bit XY mode.
180 COP	\ Execute COP call.
190 EQUB @OPBHA%	\ Make BHA point to following string.
200 EQU\$ "filename"	\ Filename here.
210 EQUB &00	\ Terminated by zero.
220 LDY# &40	\ Open type in Y (OPENIN).
230 COP	\ Execute COP call.
240 EQUB @OPOPN%	\ Open file.
250 BCS end%	\ Go to end if failed.
260 :	\ File open, handle in Y.
270 COP	\ Execute COP call.
280 EQUB @OPEND%	\ Test for end of file.
290 BCS closefile%	\ If error then close file.
300 LSRA	\ Shift low bit of A into carry.
310 BCS closefile%	\ If carry set then the end of file
	\ has been reached and file should be closed.
320 RTL	\ Return.
330 .closefile%	\ Routine to close file, see OPCLS.
900 end%	\ End here.
910 RTL	\ Return from subroutine.

This example will check if the end of the file called "filename", has been reached. If it has, then the files will be closed by the example given for OPCLS.

Note that you can only close a file which has been opened using OPOPN (or OPENIN, OPENOUT, OPENUP from BASIC).

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPCLS close file

**Action:** This call closes a file which has been opened using OPOPN.

**On entry:** Y = handle allocated by OPOPN.

**On exit:** If C = 0 then file is closed.

If C = 1 then the file close failed, X = error code and BHA points to a zero-terminated error message.

D preserved

**To call:** COP

EQUB @OPCLS% from BASIC

OPSYS &OPCLS from MASM

**Example:**

170 RSP# &10	\ Put processor in 16-bit XY mode.
175 WRD &10	\ Tell assembler that processor is in 16-bit XY mode.
180 COP	\ Execute COP call.
190 EQUB @OPBHA%	\ Make BHA point to following string.
200 EQU "filename"	\ Filename here.
210 EQUB &00	\ Terminated by zero.
220 LDY# &40	\ Open type in Y (OPENIN).
230 COP	\ Execute COP call.
240 EQUB @OPOPN%	\ Open file.
250 BCS end%	\ Go to end if failed.
260 :	\ File open, handle in Y.
270 COP	\ Execute COP call.
280 EQUB @OPEND%	\ Test for end of file.
290 BCS closefile%	\ If error then close file.
300 LSRA	\ Shift low bit of A into carry.
310 BCS closefile%	\ If carry set then the end of file
	\ has been reached and file should be closed.
320 RTL	\ Return.
330 .closefile%	\ Routine to close file.
340 COP	\ Execute COP call.
350 EQUB @OPCLS%	\ Close file.
360 BCS end%	\ Go to end if failed.
900 .end%	\ End here.
910 RTL	\ Return from subroutine.

This COP must be called in 16-bit XY mode (handles are 16 bits).

This example will attempt to close the file called "filename", when the end of file has been reached. A BASIC program to do this would be as follows:

```

10 handle% = OPENIN "filename"
20 IF EOF(handle%) THEN PROCclosefile(handle%)
30 :REM continue
100 DEF PROCclosefile(h%)
110 CLOSE#(h%)
120 ENDPROC

```

# OPBGT get byte

**Action:** Read one byte from the currently open file whose handle is in Y. The byte is returned in A.

**On entry:** Y contains the handle from OPOPN.

**On exit:** A contains the byte read from the file.

If C = 1 then an error has occurred, and A is invalid.  
No registers preserved

**To call:** COP  
EQUB @OPBGT% from BASIC  
OPSYS £OPBGT from MASM

**Example:**

170 RSP# 830	\ Put processor in 16-bit mode.
172 WRD &30	\ Tell assembler that processor is in word mode.
175 PHK	\ Push the current bank on to the stack.
180 PLB	\ Pull this value into the bank register.
185 BNK P%/&10000	\ Tell assembler that B = K.
190 LDA# filename%	\ Load HA with the middle and low bytes of the label <i>filename%</i> .
210 LDY# &40	\ BHA now points to the label <i>filename%</i> .
220 COP	\ Load Y with the open type (OPENIN).
230 EQUB @OPOPN%	\ Execute COP call.
240 BCS end%	\ Call OPEN routine.
260 COP	\ If file not opened then end.
270 EQUB @OPBGT%	\ Execute COP call.
280 .end%	\ Read byte from file into A.
290 RTL	\ Set label for end.
	\ Return from subroutine.

This example will open a file called "filename" for input, and then read the first byte from that file into A.  
**This COP must be called in 16-bit XY mode (handles are 16 bits).**

# OPBPT put byte

**Action:** Write one byte to an open file. The file must previously have been opened using OPOPN.

**On entry:** Y contains the file handle from OPOPN.  
A contains the byte to be written.

**On exit:** No registers preserved

**To call:** COP  
EQUB @OPBPT% from BASIC  
OPSYS £OPBPT from MASM

**Example:**

170 RSP# &30	\ Put processor in 16-bit mode.
172 WRD &30	\ Tell assembler that processor is in word mode.
175 PHK	\ Push the current bank on to the stack.
180 PLB	\ Pull this value into the bank register.
185 BNK P%&10000	\ Tell assembler that B = K.
190 LDA# filename% AND &FFFF	\ Load HA with the middle and low byte of the label <i>filename</i> %. \ BHA now points to the label <i>filename</i> %.
210 LDY# &80	\ Load Y with the open type (OPENOUT).
220 COP	\ Execute COP call.
230 EQUB @OPOPN%	\ Call OPEN routine.
240 BCS end%	\ If file not opened then end.
260 LDA# byte%	\ Load A with byte to be sent to file.
270 COP	\ Execute COP call.
280 EQUB @OPBPT%	\ Write byte from A into file.
290 .end%	\ Set label for end.
300 RTL	\ Return from subroutine.

This example will open a file called "filename" for output, and then write the byte contained in *byte*% to the file

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPBGB get block

**Action:** Reads the number of bytes specified in the control block from the currently open file whose handle is in Y, to the address in memory specified in the control block. The bytes are read from the file starting at the position given by the sequential pointer. The sequential pointer can be altered using OPWSP.

**On entry:** Y = handle returned by OPOPN.  
BHA points to a control block of the following form.

Offset

- |   |   |        |
|---|---|--------|
| 0 | Start address<br>of memory<br>(low byte first)            | <- BHA |
| 4 | Length of memory<br>to be transferred<br>(low byte first) |        |
| 8 |   |        |

**On exit:** If C = 0 then X = 0 means transfer successful, X = 2 means that fewer bytes were transferred than specified in the control block, and the control block is updated to show the number of bytes that were transferred.  
If C = 1 then the transfer was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
COP  
EQUB @OPBGB% from BASIC  
OPSYS \$OPBGB from MASM

**Example:**

```

170 RSP# &10          \ Put processor in 16-bit XY mode.
175 WRD &10          \ Tell assembler that processor is in 16-bit XY mode.
180 COP               \ Execute COP call.
190 EQUB @OPBHA%      \ Make BHA point to following string.
200 EQUS "filename"   \ Filename here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &40          \ Open type in Y (OPENIN).
230 COP               \ Execute COP call.
240 EQUB @OPOPN%      \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                 \ File open, handle in Y.
270 PHY &30          \ Push Y to save handle.
280 RSP# &30          \ Put processor in word mode.
285 WRD &30          \ Tell assembler that processor is in 16-bit mode.
290 LDY# @HDMMT%      \ First we'll allocate some memory.
                      \ Load Y with the pool handle,
                      \ in this case, the task memory pool.
300 LDX# @MMASA%      \ Load X with the reason code, in this
                      \ case MMASA to allocate a small ascending pool.
310 LDA# &0400          \ Load HA with the number of bytes
                      \ to be allocated.
320 COP               \ We'll choose 1 Kbyte.
330 EQUB @OPMM%        \ Execute COP call.
340 BCS closefile%     \ Call memory management routine.
                      \ Memory not allocated so close file.

```

```

350 :
360 STA controlblock%
370 PHB
380 PHB
390 PLA
400 STA controlblock%+2
410 PHK
420 PLB
425 BNK P%&10000
430 LDA# controlblock% AND &FFFF
440 PLY
450 COP
460 EQUB @OPBGB%
470 BRA closefile%
480 .controlblock%
490 EQUD &00000000
500 EQUD &00000400
510 .closefile%
520 COP
530 EQUB @OPCLS%
540 BCS end%
900 .end%
910 RTL

    \ Y contains handle of newly allocated pool.
    \ BHA contains base address of newly allocated pool.
    \ Store HA at offsets 0 and 1
    \ of the control block.
    \ Push B.
    \ Push B.
    \ Pull the high byte of the address into A.
    \ This instruction pulls 2 bytes, hence lines 360 and 370.
    \ Store the high byte at
    \ offset 2 of the control block.
    \ Push high byte of program counter.
    \ Pull this byte into B.
    \ Tell assembler that B = K.
    \ Load HA with address of control block.
    \ BHA now contains address of control block.
    \ Get handle in Y from stack.
    \ Execute COP call.
    \ Get block.
    \ Close file.
    \ Start of control block.
    \ 4 byte space for address.
    \ 4 byte length of block, in bytes.
    \ In this case we'll read 1 Kbyte into memory.
    \ Routine to close file.
    \ Execute COP call.
    \ Close file.
    \ Go to end if failed.
    \ End here.
    \ Return from subroutine.

```

This example will allocate a memory pool of 1 Kbyte, and will read 1 Kbyte from the file opened by OPOPN into the memory pool. This same example can be accomplished from BASIC as follows.

```

10 DIM P% 10, controlblock% 10, buffer% &400 :REM allocate space for code, control block, and buffer
20 [ .getBytes% COP : EQUB @OPBGB% : RTL :] :REM COP code
30 getbytes% = getbytes% + @WRD% :REM routine to be called in word mode
40 Y% = OPENIN "filename" :REM open file for input
50 A% = controlblock% :REM load HA with address of control block
60 Icontrolblock% = buffer% :REM initialise control block
70 controlblock%14 = &400 :REM initialise control block
80 CALL getbytes% :REM call routine
90 CLOSE# Y% :REM close file

```

# OPBPB put block

**Action:** Writes the number of bytes specified in the control block, from the address in memory specified in the control block, to the file whose handle is in Y. The bytes are written to the file starting at the position given by the sequential pointer. The sequential pointer can be altered using OPWSP.

**On entry:** Y = handle returned by OPOPN.  
BHA points to a control block of the following form.

Offset

- |   |   |        |
|---|---|--------|
| 0 | Start address<br>of memory<br>(low byte first)            | -> BHA |
| 4 | Length of memory<br>to be transferred<br>(low byte first) |        |
| 8 |   |        |

**On exit:** If C = 0 then transfer was successful.  
If C = 1 then the transfer was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
COP  
EQUB @OPBPB% from BASIC  
OPSYS £OPBPB from MASM

**Example:**

```

20 DIM buffer% 100 :REM allocate 100 byte buffer
170 RSP# &10          \ Put processor in 16-bit XY mode.
175 WRD &10          \ Tell assembler that processor is in 16-bit XY mode.
180 COP
190 EQUB @OPBHA%
200 EQUS "filename"
210 EQUB &00          \ Execute COP call.
220 LDY# &80          \ Make BHA point to following string.
230 COP
240 EQUB @OPOP%       \ Filename here.
250 BCS end%          \ Terminated by zero.
260 :                 \ Open type in Y (OPENOUT).
270 RSP# &30          \ Execute COP call.
275 WRD &30          \ Open file.
280 PHK
290 PLB
295 BNK P%&/10000    \ Go to end if failed.
300 LDA# controlblock% AND &FFFF  \ File open, handle in Y.
310 COP
320 EQUB @OPBPB%
330 BRA closefile%
340 .controlblock%
350 EQUD buffer%      \ Put processor in word mode.
360 EQUD 100          \ Tell assembler that processor is in 16-bit mode.
370 .closefile%        \ Push high byte of program counter.
380 COP
390 EQUB @OPCLS%
400 BCS end%          \ Pull this byte into B.
410 .end%              \ Tell assembler that B = K.
420 LDA# HA            \ Load HA with address of control block.
430 STX# B              \ BHA now contains address of control block.
440 COP
450 EQUD start%        \ Execute COP call.
460 EQUD len%           \ Get block.
470 EQUD 0              \ Close file.
480 EQUD start%         \ Start of control block.
490 EQUD len%           \ 4 byte address of start of
500 EQUD buffer%        \ buffer to be written.
510 EQUD len%           \ 4 byte length of block, in bytes.
520 EQUD 100             \ In this case 100 (decimal) bytes.
530 EQUD 0              \ Routine to close file.
540 COP
550 EQUD 0              \ Execute COP call.
560 EQUD 0              \ Close file.
570 EQUD 0              \ Go to end if failed.
580 EQUD 0              \ End here.
590 RTL                \ Return from subroutine.

```

This example will write 100 bytes from memory starting at the address specified by the BASIC variable *buffer%* to the file opened by OPOP%. This same example can be accomplished from BASIC as follows:

```

10 DIM P% 10, controlblock% 10, buffer% 100 :REM allocate space for code, control block, and buffer
20 [.putbytes% COP : EQUB @OPBPB% : RTL : ] :REM COP code
30 putbytes% = putbytes% + @WRD% :REM routine to be called in word mode
40 Y% = OPENOUT "filename" :REM open file for output
50 A% = controlblock% :REM load HA with address of control block
60 !controlblock% = buffer% :REM Initialise control block
70 controlblock%4 = 100 :REM Initialise control block
80 CALL putbytes% :REM call routine
90 CLOSE# Y% :REM close file

```

# OPLOD

load file

**Action:** Loads a file into memory from the specified filing system.

**On entry:** BHA points to a control block of the following form.

**Offset**

0	Pointer to name of file (low byte first)	<- BHA
4	File LOAD address (low byte first)	
8	File EXEC address (low byte first)	
12	Initial memory location (low byte first)	
16	Length of memory area (low byte first)	

20

A zero length parameter &00000000 will cause any length of file to be loaded. The length is updated upon exit to show the number of bytes actually loaded.

The load address and execution address are not parameters - they are set on exit to the file's values.

**On exit:** If C = 0 and X = 0 then the file was loaded successfully and the load and execution addresses are valid.

If C = 0 and X = 2 then the file was loaded successfully but the load and execution addresses are invalid.

If C = 1 then the load was unsuccessful, X = error code, BHA points to a zero-terminated error message.

Y preserved

**To call:**

COP
EQUB @OPLOD%
from BASIC
OPSYS £OPLOD
from MASM

**Example:**

```
20 DIM space% 10000 :REM make room for file
170 RSP# &30          \ Put processor in word mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 PHK               \ Push high byte of program counter.
190 PLB               \ Pull high byte into B.
195 BNK P%/&10000    \ Tell assembler that B = K.
200 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
210 COP               \ Execute COP call.
220 EQUB @OPLOD%     \ Load file.
230 RTL               \ End.
240 .filename%        \ Address of file name.
250 EQUUS "filename" \ File name.
260 EQUB &00          \ Terminated by zero.
270 .controlblock%    \ Address of control block.
280 EQUD filename%   \ Pointer to filename.
290 EQUD &00000000    \ Four bytes for load address.
300 EQUD &00000000    \ Four bytes for execution address.
310 EQUD space%      \ Address of start of memory.
320 EQUD 10000        \ Maximum length of file (or zero for any length).
```

This example will load a file called *filename* into memory, provided the file is not more than 10000 bytes in length.

# OPSAV save file

**Action:** Saves an area of memory to the specified filing system.

**On entry:** BHA points to a control block of the following form.

**Offset**

0	Pointer to name of file (low byte first)	<- BHA
4	File LOAD address (low byte first)	
8	File EXEC address (low byte first)	
12	Initial memory location (low byte first)	
16	Length of memory area (low byte first)	

20

**On exit:**

If C = 0 and X = 0 then the file was saved successfully and the load and execution addresses were successfully updated.

If C = 0 and X = 2 then the file was saved successfully but the load and execution addresses were not successfully updated.

If C = 1 then the save was unsuccessful, X = error code, BHA points to zero-terminated error message.

Y preserved

**To call:**

COP  
EQUB @OPSAV% from BASIC  
OPSYS £OPSAV from MASM

**Example:**

20 DIM buffer% 1700 :REM area to be saved

170 RSP# &30	\ Put processor in word mode.
175 WRD &30	\ Tell assembler that processor is in 16-bit mode.
180 PHK	\ Push high byte of program counter.
190 PLB	\ Pull high byte into B.
195 BNK P%&10000	\ Tell assembler that B = K.
200 LDAJ controlblock% AND &FFFF	\ Load HA with pointer to control block.
210 COP	\ Execute COP call.
220 EQUB @OPSAV%	\ Save file.
230 RTL	\ End.
240 .filename%	\ Address of file name.
250 EQUIS "filename"	\ File name.
260 EQUB &00	\ Terminated by zero.
270 .controlblock%	\ Address of control block.
280 EQUD filename%	\ Pointer to filename.
290 EQUD buffer%	\ Load address.
300 EQUD buffer%	\ Execution address.
310 EQUD buffer%	\ Address of start of memory.
320 EQUD 1700	\ Length of memory to be saved.

This example will save 1700 bytes from *buffer%* to a file called *filename*. The load and execution addresses will be saved as the address from where the file came.

# OPRLE

read load address, execute address, and length of file

**Action:** Reads the load address, execute address, and length of a file.

**On entry:** BHA points to a control block of the following form.

Offset

- |   |  |        |
|---|--|--------|
| 0 | Pointer to name<br>of file<br>(low byte first) | -> BHA |
| 4 | File LOAD address<br>(low byte first)          |        |
| 8 | File EXEC address<br>(low byte first)          |        |

12

The load address and execution address are not parameters - they are set on exit to the file's values.

**On exit:** If C = 0 then the control block has been updated to give the load and execute addresses, and the length is in XHA (2 bytes in X, 4 bytes in total).  
If C = 1 then the save was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
**COP**  
**EQUB @OPRLE%** from BASIC  
**OPSYS EOPRLE** from MASM

**Example:**

170 RSP# &30	\ Put processor in word mode.
175 WRD &30	\ Tell assembler that processor is in 16-bit mode.
180 PHK	\ Push high byte of program counter.
190 PLB	\ Pull high byte into B.
195 BNK P%&10000	\ Tell assembler that B = K.
200 LDA# controlblock%	\ Load HA with pointer to control block.
210 COP	\ Execute COP call.
220 EQUB @OPRLE%	\ Read load and execute addresses.
230 RTL	\ End.
240 .filename%	\ Address of file name.
250 EQU\$ "filename"	\ File name.
260 EQUB &00	\ Terminated by zero.
270 .controlblock%	\ Address of control block.
280 EQUD filename%	\ Pointer to filename.
290 EQUD &00000000	\ Four bytes for load address.
300 EQUD &00000000	\ Four bytes for execution address.

This example will read the load and execution address of the file called *filename* into the control block, and will read the length of the file into XHA (4 bytes).

# OPWLE

write load address and execute address of file

**Action:** Writes new load and execute addresses to a file.

**On entry:** BHA points to a control block of the following form.

Offset

- |   |  |        |
|---|--|--------|
| 0 | Pointer to name<br>of file<br>(low byte first) | -> BHA |
| 4 | File LOAD address<br>(low byte first)          |        |
| 8 | File EXEC address<br>(low byte first)          |        |

12

**On exit:** If C = 0 then the file has been updated.

If C = 1 then the update was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
 COP  
 EQUB @OPWLE% from BASIC  
 CPSYS £OPWLE from MASM

**Example:**

```

20 DIM loadexec% 0 :REM address at which to load and run
170 RSP# &30          \ Put processor in word mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 PHK               \ Push high byte of program counter.
190 PLB               \ Pull high byte into B.
195 BNK P%&10000      \ Tell assembler that B = K.
200 LDAB controlblock% AND &FFFF \ Load HA with pointer to control block.
210 COP               \ Execute COP call.
220 EQUB @OPWLE%       \ Write load and execute addresses.
230 RTL               \ End.
240 .filename%         \ Address of file name.
250 EQUUS "filename"  \ File name.
260 EQUB &00           \ Terminated by zero.
270 .controlblock%     \ Address of control block.
280 EQUUD filename%   \ Pointer to filename.
290 EQUUD loadexec%   \ New load adres.
300 EQUUD loadexec%   \ New execution address.

```

This example will write a new load address and a new execute address to the file called *filename%*.

# OPRAT read attributes

**Action:** Reads the 4-byte file attributes.

**On entry:** BHA points to a control block of the following form.

Offset

0	Pointer to name of file (low byte first)	<- BHA
4	File attributes (low byte first)	
8		

The attributes have the following meanings:

low byte	bit 0 0 not readable by owner 1 readable by owner
bit 1	0 not writable by owner 1 writable by owner
bit 2	undefined
bit 3	0 not locked 1 locked
bit 4	0 not readable by public 1 readable by public
bit 5	0 not writable by public 1 writable by public
bit 6	undefined
bit 7	undefined

low middle byte days

high middle byte bits 0 to 3 months  
bits 4 to 7 years since 1981

high byte undefined

**On exit:** If C = 0 and X = 1 then a file was found and the attributes were read successfully.  
 If C = 0 and X = 2 then a directory was found and the attributes were read successfully.  
 If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated  
 error message.  
 Y preserved

**To call:**  
**COP**  
**EQUB @OPRAT%** from BASIC  
**OPSYS COPRAT** from MASM

**Example:**

170 RSP# &30	\ Put processor in word mode.
175 WRD &30	\ Tell assembler that processor is in 16-bit mode.
180 PHK	\ Push high byte of program counter.
190 PLB	\ Pull high byte into B.
195 BNK P%&10000	\ Tell assembler that B = K.
200 LDA@ controlblock% AND &FFFF	\ Load HA with pointer to control block.
210 COP	\ Execute COP call.
220 EQUB @OPRAT%	\ Read attributes.
230 RTL	\ End.
240 .filename%	\ Address of file name.
250 EQU\$ "filename"	\ File name.
260 EQUB &0	\ Terminated by zero.
270 .controlblock%	\ Address of control block.
280 EQUD filename%	\ Pointer to filename.
290 EQUD &00000000	\ Four bytes for attributes.

This example will read the 4-byte attributes of the file called *filename* into the control block.

# OPWAT write attributes

**Action:** Writes the 4-byte file attributes.

**On entry:** BHA points to a control block of the following form.

Offset

0	Pointer to name of file (low byte first)	<- BHA
4	File attributes (low byte first)	
8		

For the meaning of attributes, see OPRAT.

**On exit:** If C = 0 then the attributes were written successfully.

If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.

Y preserved

**To call:**

COP  
EQUB @OPWAT% from BASIC  
OPSYS £OPWAT from MASM

**Example:**

```

30 DIM attributes% 3 :REM reserve 4 bytes for attributes
170 RSP# &30          \ Put processor in word mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 PHK               \ Push high byte of program counter.
190 PLB               \ Pull high byte into B.
195 BNK P%&10000     \ Tell assembler that B = K.
200 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
210 COP               \ Execute COP call.
220 EQUB @OPWAT%      \ Write attributes.
230 RTL               \ End.
240 .filename%        \ Address of file name.
250 EQUUS "filename" \ File name.
260 EQUB &00          \ Terminated by zero.
270 .controlblock%    \ Address of control block.
280 EQUD filename%   \ Pointer to filename.
290 EQUD attributes% \ Four byte attributes.

```

This example will write the attributes contained in the BASIC variable *attributesef105*, to the file called *filename*.

# OPRSP

read sequential pointer

**Action:** Reads the sequential pointer - the position within a file at which the next read/write operation will occur.

**On entry:** Y = handle  
BHA points to a control block of the following form.

Offset

0 Sequence number or logical length (low byte first)	-> BHA
--	--------

4

**On exit:** If C = 0 then the sequential pointer was read successfully into the control block.  
If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
COP  
EQUB @OPRSP% from BASIC  
OPSYS £OPRSP from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP               \ Execute COP call.
190 EQUB @OPBHA%      \ Make BHA point to following string.
200 EQUS "filename"   \ Filname here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &40          \ Open type in Y (OPENIN).
230 COP               \ Execute COP call.
240 EQUB @OPOP%       \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                 \ File open, handle in Y.
270 PHK               \ Push high byte of program counter.
280 PLB               \ Pull high byte into B.
285 BNK P#/&10000     \ Tell assembler that B = K.
290 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
300 COP               \ Execute COP call.
310 EQUB @OPRSP%      \ Read sequential pointer.
320 BRA end%          \ End.
330 .controlblock%    \ Address of control block.
340 EQUD &00000000    \ Space for sequential pointer.
900 .end%              \ End here.
910 RTL               \ Return from subroutine.

```

This example will open the file called *filename* and read the sequential pointer. (As the file has only just been opened the value will be 0.) Note that in this example the file is still open. It must be closed when it is finished with.

# OPWSP write sequential pointer

**Action:** Writes a new value to the sequential pointer - the position within a file at which the next read/write operation will occur.

**On entry:** Y = handle  
BHA points to a control block of the following form.

Offset

0	Sequence number or logical length (low byte first)	-> BHA
4		

**On exit:** If C = 0 then the sequential pointer was written successfully.  
If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
COP  
EQUB @OPWSP% from BASIC  
OPSYS £OPWSP from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP
190 EQUB @OPBHA%      \ Make BHA point to following string.
200 EQUS "filename"    \ Filename here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &80          \ Open type in Y (OPENOUT).
230 COP
240 EQUB @OPOPN%      \ Execute COP call.
250 BCS end%          \ Open file.
260 :                 \ Go to end if failed.
270 PHK               \ File open, handle in Y.
280 PLB               \ Push high byte of program counter.
285 BNK P%/&10000      \ Pull high byte into B.
290 LDA# controlblock% AND &FFFFF \ Tell assembler that B = K.
300 COP               \ Load HA with pointer to control block.
310 EQUB @OPWSP%      \ Execute COP call.
320 BRA end%          \ Write sequential pointer.
330 .controlblock%    \ End.
340 EQUD 172          \ Address of control block.
900 .end%             \ New value of sequential pointer.
910 RTL               \ End here.
                           \ Return from subroutine.

```

This example will open the file called *filename* and write the value 174 (decimal) to the sequential pointer. Note that in this example the file is still open. It must be closed when it is finished with.

# OPRSP

read sequential pointer

**Action:** Reads the sequential pointer - the position within a file at which the next read/write operation will occur.

**On entry:** Y = handle  
BHA points to a control block of the following form.

Offset

0 Sequence number or logical length (low byte first)	<- BHA
--	--------

4

**On exit:** If C = 0 then the sequential pointer was read successfully into the control block.  
If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**  
COP  
EQUB @OPRSP% from BASIC  
OPSYS EOPRSP from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP              \ Execute COP call.
190 EQUB @OPBHA%      \ Make BHA point to following string.
200 EQUS "filename"   \ Filename here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &40          \ Open type in Y (OPENIN).
230 COP              \ Execute COP call.
240 EQUB @OPOPN%      \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                \ File open, handle in Y.
270 PHK              \ Push high byte of program counter.
280 PLB              \ Pull high byte into B.
285 BNK P%/&10000     \ Tell assembler that B = K.
290 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
300 COP              \ Execute COP call.
310 EQUB @OPRSP%      \ Read sequential pointer.
320 BRA end%          \ End.
330 .controlblock%    \ Address of control block.
340 EQUD &00000000    \ Space for sequential pointer.
900 .end%             \ End here.
910 RTL              \ Return from subroutine.

```

This example will open the file called *filename* and read the sequential pointer. (As the file has only just been opened the value will be 0.) Note that in this example the file is still open. It must be closed when it is finished with.

# OPWSP write sequential pointer

**Action:** Writes a new value to the sequential pointer - the position within a file at which the next read/write operation will occur.

**On entry:** Y = handle  
BHA points to a control block of the following form.

Offset

0 Sequence number or logical length (low byte first)	<- BHA
--	--------

4

**On exit:** If C = 0 then the sequential pointer was written successfully.  
If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:** COP  
EQUB @OPWSP% from BASIC  
OPSYS @OPWSP from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP
190 EQUB @OPBHA%
200 EQUUS "filename" \ Execute COP call.
210 EQUB &00          \ Make BHA point to following string.
220 LDY# &80          \ Filname here.
230 COP              \ Terminated by zero.
240 EQUB @OPOPEN%     \ Open type in Y (OPENOUT).
250 BCS end%          \ Execute COP call.
260 :                 \ Open file.
270 PHK              \ Go to end if failed.
280 PLB              \ File open, handle in Y.
285 BNK P%&10000     \ Push high byte of program counter.
290 LDA# controlblock% AND &FFFF \ Pull high byte into B.
300 COP              \ Tell assembler that B = K.
310 EQUB @OPWSP%     \ Load HA with pointer to control block.
320 BRA end%          \ Execute COP call.
330 .controlblock%   \ Write sequential pointer.
340 EQUD 172          \ End.
350 .end%             \ Address of control block.
360 EQUD 172          \ New value of sequential pointer.
370 .end%             \ End here.
380 RTL              \ Return from subroutine.

```

This example will open the file called *filename* and write the value 174 (decimal) to the sequential pointer. Note that in this example the file is still open. It must be closed when it is finished with.

# OPRPL read physical length

**Action:** Reads the physical length of a file, ie the actual length stored on the media.

**On entry:** BHA points to a control block of the following form.

**Offset**

0	Pointer to name of file (low byte first)	<- BHA
4	Length (low byte first)	
8		

**On exit:** If C = 0 then the physical length was read successfully into the control block.

If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:**

COP  
EQUB @OPRPL% from BASIC  
OPSYS OPRPL from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP               \ Execute COP call.
190 EQUB @OPBHA%      \ Make BHA point to following string.
200 EQUUS "filename" \ Filename here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &40          \ Open type in Y (OPENIN).
230 COP               \ Execute COP call.
240 EQUB @OPOPN%      \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                 \ File open, handle in Y.
270 PHK               \ Push high byte of program counter.
280 PLB               \ Pull high byte into B.
285 BNK P%&1000        \ Tell assembler that B = K.
290 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
300 COP               \ Execute COP call.
310 EQUB @OPRPL%      \ Read physical length.
320 BRA end%          \ End.
330 .controlblock%    \ Address of control block.
340 EQUD &00000000    \ Space for physical length.
900 .end%             \ End here.
910 RTL               \ Return from subroutine.

```

This example will open the file called *filename* and read the physical length. Note that in this example the file is still open. It must be closed when it is finished with.

# OPRLL read logical length

**Action:** Reads the logical length (extent) of a file.

**On entry:** Y = handle  
BHA points to a control block of the following form.

Offset

0 Sequence number or length (low byte first)	-> BHA
--	--------

4

**On exit:** If C = 0 then the logical length was read successfully into the control block.  
If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.  
Y preserved

**To call:** COP  
EQUB @OPRLL% from BASIC  
OPSYS \$OPRLL from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP               \ Execute COP call.
190 EQUB @OPBHA%       \ Make BHA point to following string.
200 EQUUS "filename"  \ Filenname here.
210 EQUB &00           \ Terminated by zero.
220 LDY# &40           \ Open type in Y (OPENIN).
230 COP               \ Execute COP call.
240 EQUB @OPOPN%       \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                 \ File open, handle in Y.
270 PHK               \ Push high byte of program counter.
280 PLB               \ Pull high byte into B.
285 BNK P%/&10000      \ Tell assembler that B = K.
290 LDA# controlblock% AND &FFFF  \ Load HA with pointer to control block.
300 COP               \ Execute COP call.
310 EQUB @OPRLL%       \ Read logical length.
320 BRA end%          \ End.
330 .controlblock%    \ Address of control block.
340 EQUD &00000000     \ Space for logical length.
900 .end%              \ End here.
910 RTL               \ Return from subroutine.

```

This example will open the file called *filename* and read the logical length. Note that in this example the file is still open. It must be closed when it is finished with.

# OPWLL write logical length

**Action:** Writes the logical length (extent) of a file.

**On entry:** Y = handle  
BHA points to a control block of the following form.

Offset

0	Sequence number or length (low byte first)	<- BHA
---	--	--------

4

**On exit:** If C = 0 then the logical length was written successfully.

If C = 1 then the call was unsuccessful, X = error code, BHA points to a zero-terminated error message.

Y preserved

**To call:** COP  
EQUB @OPWLL% from BASIC  
OPSYS COPWLL from MASM

**Example:**

```

170 RSP# &30          \ Put processor in 16-bit mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 COP               \ Execute COP call.
190 EQUB @OPBHA%       \ Make BHA point to following string.
200 EQUUS "filename"  \ Filename here.
210 EQUB &00          \ Terminated by zero.
220 LDY# &80          \ Open type in Y (OPENOUT).
230 COP               \ Execute COP call.
240 EQUB @OPOPN%       \ Open file.
250 BCS end%          \ Go to end if failed.
260 :                 \ File open, handle in Y.
270 PHK               \ Push high byte of program counter.
280 PLB               \ Pull high byte into B.
285 BNK P%/&10000      \ Tell assembler that B = K.
290 LDAJ controlblock% AND &FFFF \ Load HA with pointer to control block.
300 COP               \ Execute COP call.
310 EQUB @OPWSP%       \ Write logical length.
320 BRA end%          \ End.
330 .controlblock%     \ Address of control block.
340 EQUD $1A00          \ Logical length.
900 .end%              \ End here.
910 RTL               \ Return from subroutine.

```

This example will open the file called *filename* and write to it a logical length of &1A00. Note that in this example the file is still open. It must be closed when it is finished with.

# OPRCH read catalogue header and information

**Action:** Given the pathname, which may contain wildcards, this call will give the full directory name and information about that directory.

**On entry:** BHA points to a control block of the following form.

Offset

- |   |  |        |
|---|--|--------|
| 0 | Pointer to name<br>of directory<br>(low byte first)        | -> BHA |
| 4 | Pointer to 49 byte<br>long buffer area<br>(low byte first) |        |
| 8 |  |        |

**On exit:** If C = 0, the 49 byte buffer area specified in the control block returns with the following information.

Buffer contents after call:

BHA undefined

Offset

- |          |  |
|----------|--|
| 0        | 0=owner, &FF=public  |
| 1        | Cycle number   |
| 2        | Boot option  |
| 3 to 12  | Directory name (without wildcards),<br>left-justified & padded with spaces |
| 13 to 28 | CSD disc name, same format (used to be last)                               |
| 29 to 38 | CSD name, same format  |
| 39 to 48 | LIB name, same format  |

If C = 1 then directory not found, X = error code, BHA points to zero-terminated error message.  
Y preserved

**To call:**

COP  
EQUB @OPRCH% from BASIC  
OPSYS EOPRCH from MASM

**Example:**

```
20 DIM buffer% 48 :REM allocate 49 byte buffer space
170 RSP# &30          \ Put processor in word mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 PHK               \ Push high byte of program counter.
190 PLB               \ Pull high byte into B.
195 BNK P%/&10000     \ Tell assembler that B = K.
200 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
210 COP               \ Execute COP call.
220 EQUB @OPRCH%      \ Read directory info.
230 RTL               \ End.
240 .pathname%        \ Address of directory pathname.
250 EQUUS "$.fred.x*" \ Pathname.
260 EQUB &00          \ Terminated by zero.
270 .controlblock%    \ Address of control block.
280 EQUD pathname%    \ Pointer to pathname.
290 EQUD buffer%      \ Pointer to 49 byte buffer.
```

This example will read the name of the directory which matches the pathname \$.fred.x\* and place information about this directory in the 49 byte buffer.

# OPRFN Read file (object) names from directory

**Action:** Given the pathname, which may contain wildcards, this call will read the number and names of entries in the specified directory. The information is put in an area of memory whose start address is supplied. Each name is terminated by zero (&00).

**On entry:** BHA points to a control block of the following form.

**Offset**

- |    |  |        |
|----|--|--------|
| 0  | Pointer to name<br>of directory<br>(low byte first)                                    | -> BHA |
| 4  | Pointer to start<br>of memory area<br>(low byte first)                                 |        |
| 8  | Type of info wanted  |        |
| 10 | Number of names  |        |
| 12 | Filing system<br>work space (16 bytes)<br>(Initially the whole<br>area should be zero) |        |

28

The type of info values are:

- 0 for name only (maximum 16 bytes per name)
- 1 for short information (maximum 20 bytes per entry)
- 2 for full information (maximum 80 bytes per entry)

The actual content of these results is filing system dependent.

The number of names is a parameter which can be set so the call will read up to that number of entries. If there are any remaining then the call can be used again, without disturbing the control block, to carry on from where it left off. For the first call however, the workspace should be zeroed.

**On exit:** If C = 0 then X = the number of names read, and the names and info are in memory separated by zeros.

If C = 1 then directory not found, X = error code, BHA points to zero-terminated error message.  
Y preserved

**To call:**

COP  
EQUB @OPRFN% from BASIC  
OPSYS EOPRFN from MASM

**Example:**

```

20 DIM memory% 1000 :REM allocate 1000 bytes for names and info
170 RSP# &30          \ Put processor in word mode.
175 WRD &30          \ Tell assembler that processor is in 16-bit mode.
180 PHK               \ Push high byte of program counter.
190 PLB               \ Pull high byte into B.
195 BNK P%/&10000    \ Tell assembler that B = K.
200 LDA# controlblock% AND &FFFF \ Load HA with pointer to control block.
210 COP               \ Execute COP call.
220 EQUB @OPRFN%     \ Read entry names.
230 RTL               \ End.
240 .pathname%        \ Address of directory pathname.
250 EQUIS "$.fred.meals" \ Pathname.
260 EQUB &00          \ Terminated by zero.
270 .controlblock%    \ Address of control block.
280 EQUD pathname%   \ Pointer to pathname.
290 EQUD memory%    \ Pointer to memory area.
300 EQUW &0000        \ Type of info.
310 EQUW 10           \ Number of names to be read.
320 EQUD &00000000    \ Initialise workspace.
330 EQUD &00000000    \ Initialise workspace.
340 EQUD &00000000    \ Initialise workspace.
350 EQUD &00000000    \ Initialise workspace.

```

Note: if the number of names to be read (line 310) is greater than &16 then the maximum number actually read will be &16.

This example will read the names of up to 36 entries from the directory whose pathname is \$.fred.meals. X will contain the actual number of entries whose names were read.

# OPDEL delete a named object

**Action:** This call will attempt to delete a named object (ie a file or a directory) from a directory.

**On entry:** BHA points to the object name.

**On exit:** If C = 0 then the object was deleted.

If C = 1 then object not found or not deleted, X = error code, BHA points to zero-terminated error message.

Y preserved

**To call:**

COP

EQUB @OPDEL% from BASIC

OPSYS £OPDEL from MASM

**Example:**

170 RSP# &30	\ Put processor in 16-bit mode.
175 WRD &30	\ Tell assembler that processor is in 16-bit mode.
180 COP	\ Execute COP call.
190 EQUB @OPBHA%	\ Make BHA point to following string.
200 EQU\$ "objectname"	\ Object name here.
210 EQUB &00	\ Terminated by zero.
230 COP	\ Execute COP call.
240 EQUB @OPDEL%	\ Delete object.

This example will attempt to delete the object called *objectname*.

# OPREN rename object

**Action:** Will attempt to rename an object (file or directory).

**On entry:** BHA points to a control block of the following form.

Offset

0	Pointer to old filename (low byte first)	← BHA
4	Pointer to new filename (low byte first)	
8		

**On exit:** If C = 0 then the object has been renamed.

If C = 1 then object not found or not renamed, X = error code, BHA points to zero-terminated error message.

Y preserved

**To call:**

COP

EQUB @OPREN% from BASIC

OPSYS £OPREN from MASM

**Example:**

170 RSPW &30	＼ Put processor in word mode.
175 WRD &30	＼ Tell assembler that processor is in 16-bit mode.
180 PHK	＼ Push high byte of program counter.
190 PLB	＼ Pull high byte into B.
195 BNK P%/&10000	＼ Tell assembler that B = K.
200 LDAW controlblock% AND &FFFF	＼ Load HA with pointer to control block.
210 COP	＼ Execute COP call.
220 EQUB @OPREN%	＼ Rename object.
230 RTL	＼ End.
240 .oldname%	＼ Address of old name.
250 EQUS "fred"	＼ Old name.
260 EQUB &00	＼ Terminated by zero.
270 .newname%	＼ Address of new name.
280 EQUS "bob"	＼ New name.
290 EQUB &00	＼ Terminated by zero.
300 .controlblock%	＼ Address of control block.
310 EQUD oldname%	＼ Pointer to old name.
320 EQUD newname%	＼ Pointer to new name.

This example will attempt to rename the file or directory called *fred* to the new name *bob*.



## 8. Interrupt handling

There are three COP calls which allow devices to be added to or removed from the list, maintained by the operating system, of possible sources of interrupts.

### OPIIQ Intercept interrupt

**Action:** This call is used to add a device's interrupt service to the list of such services maintained by the operating system.

**On entry:** Inline 3 byte hardware address of the device which requires servicing's status register.  
Inline 1 byte EOR mask, allowing inversion of bits to the correct logic level if necessary.  
BHA points to the start address of the interrupt routine.  
X contains an AND mask to discriminate between different devices causing interrupts. (X must be set to zero if a call to OPMIQ is required.)  
Y contains the priority (range 1 to 255). This will be the position within the list of devices which the new entry will occupy. The lower the value the higher the priority.  
D is set to the direct page required whilst in the interrupt routine.  
P flags are set to give the mode required in the interrupt routine. (The operating system sets the I flag.)

**On exit:** If C = 0 then the call succeeded, Y = handle and HA = handle.  
If C = 1 then the call failed, Y = 0 and HA = 0.  
No registers preserved

**To call:**

```
COP
EQU8 @OPIIQ%
EQU1 hwaddress%
EQU8 eormask%    from BASIC
OPSY8 OPIIQ
= $HWBANK
& $HWADDRESS
= $EORMASK      from MASM
```

**Example:**

170 LDA# <i>introutine%</i> DIV &10000	\ Load the accumulator with \ the high byte of the start of the \ interrupt routine.
180 PHA	\ Push the high byte on to the stack.
190 PLB	\ Pull the high byte into the bank register.
195 BNK <i>introutine%&amp;10000</i>	\ Tell assembler that B = high \ byte of interrupt routine address.
200 RSP# &20	\ Put processor into 16 bit \ accumulator mode.
205 WRD &20	\ Tell assembler that processor \ is in 16-bit accumulator mode.
210 LDA# <i>introutine%</i>	\ Load the accumulator with \ the remaining 2 bytes of the start \ of the interrupt routine.
220 LDA# <i>andmask%</i>	\ Load X with the AND mask.
230 LDY# <i>priority%</i>	\ Load Y with the priority.
240 LDA# <i>dp%</i>	\ Load the accumulator with \ the required 2 byte direct page.
250 TAD	\ Transfer this value to the \ direct page register (D).
260 SEP# &20	\ Put processor in 8-bit accumulator mode.
265 BYT &20	\ Tell assembler that processor is in 8-bit \ accumulator mode.
270 COP	\ Execute COP call.
280 EQUB @OPIIQ%	\ Add interrupt service to list.
290 EQUAL <i>hwaddress%</i>	\ 3 byte hardware address.
300 EQUB <i>eormask%</i>	\ EOR mask.

Note: the BASIC variables *introutine%*, *mask%*, *priority%*, *dp%*, *hwaddress%* and *eormask%* must all be allocated sensible values before this example program will run.

This example will add an interrupt service to the list maintained by the operating system.

When IRQ is pulled low, the operating system works its way through the list from the top until it discovers the device which caused the interrupt. Therefore, the lower the value given to OPIIQ in Y, the nearer the top of the list and the quicker the interrupt will be serviced.

The hardware address is the 3-byte address of the device's status register. It is this register's contents which must be examined to determine if this was the device causing the IRQ. The status byte at the hardware address is first ANDed with a location set up by the operating system to contain &FF. (The address of this location can be forced to a different value by the OPMIQ call.)

Next the byte is EORed with the EOR mask. This will either leave the byte alone (*eormask% = &00*) or will invert some or all of the bits to get the correct logic levels.

The byte is then ANDed with the AND mask in X.

If the resulting value is zero then the device did not cause the interrupt and the operating system continues down the list. If the result is not zero then control is switched to the interrupt routine at the address in BHA.

# OPMIQ Modify interrupt intercept

**Action:** This call allows the modification of the address of the AND mask (by default set to a location containing &FF), and the value of the AND mask contained in X. (X should be set to zero by OPIIQ before using OPMIQ. OPMIQ is then used to specify the required mask.)

**On entry:** Y = handle returned by OPIIQ.  
 BHA = 0 means do not modify the address of the AND mask.  
 BHA <> 0 means BHA is the new address of the AND mask.  
 X = 0 means do not modify the AND mask.  
 X <> 0 means X is the new AND mask.

**On exit:** C = 0 means that the interrupt intercept was modified.  
 C = 1 means that the interrupt intercept was not modified.  
 No registers preserved

**To call:** COP  
 EQUB @OPMIQ% from BASIC  
 OPSYS EOPMIQ from MASM

**Example:**

Add this example to the end of the example given for OPIIQ.

```

310 LDA# andmask1%           \ Load the accumulator with the AND
                             \ mask to be stored at the modified address.
320 STA maskaddress%         \ Store the AND mask at the
                             \ modified address.
330 LDA# maskaddress% DIV &10000 \ Load the accumulator with
                             \ the high byte of the modified
                             \ address of the AND mask.
340 PHA                      \ Push this byte on to the stack.
350 PLB                      \ Pull this byte into the data
                             \ bank register (B).
355 BNK maskaddress%/&10000 \ Tell assembler that B = high byte
                             \ of mask address.
360 RSP# &30                  \ Put the processor into 16 bit
                             \ accumulator and XY mode.
365 WRD &30                  \ Tell assembler that the processor is in 16 bit mode.
370 LDA# maskaddress%         \ Load the accumulator with
                             \ the remaining 2 bytes of the
                             \ modified address.
380 LDX# andmask2%           \ Load X with the other AND mask.
390 COP                      \ Execute COP call.
400 EQUB @OPMIQ%             \ Modify interrupt service list.

```

Note: the BASIC variables *inroutine%*, *mask%*, *priority%*, *dp%*, *hwaddress%*, *eormask%*, *andmask1%*, *andmask2%* and *maskaddress%* must all be allocated sensible values before this example program will run.

This example will modify the entry in the list of interrupts made by OPIIQ. The address of the AND mask and the X AND mask are both modified. If this routine is used then X should be set to zero when calling OPIIQ.

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPRIQ Release interrupt intercept

**Action:** This call removes the specified interrupt service from the list.

**On entry:** Y = handle returned by OPIIQ.

**On exit:** C = 0 means that the call released the interrupt intercept.

C = 1 means that the call failed to release the interrupt intercept.

No registers preserved

**To call:** COP  
EQUB @OPRIQ% from BASIC

OPSY \$OPRIQ from MASM

**Example:**

Add this example to the end of the example given for OPIIQ.

```
310 COP      \ Execute COP call.  
320 EQUB @OPRIQ% \ Release interrupt intercept  
                  \ whose handle is in Y.
```

Note: the BASIC variables *introutine%*, *mask%*, *priority%*, *dp%*, *hwaddress%*, *eormask%*, *andmask1%*, *andmask2%* and *maskaddress%* must all be allocated sensible values before this example program will run.

This example will remove the interrupt service just added to the list.

This COP must be called in 16-bit XY mode (handles are 16 bits).

## 9. AR arithmetic package module

The arithmetic package is a module. Eventually, the entry address of modules will be available through the operating system. However, for now it is called using the COP OPARM call. The reason code is loaded into X as usual. See section 7.1.4 for details on COP calls.

Many of the arithmetic functions available in languages such as BASIC are difficult to code in 65SC816 assembler, and occupy a lot of space. To simplify programs which involve complicated arithmetic, the operating system has an entry point which performs most of the required tasks, eg addition, multiplication, sine etc. There are also some integer arithmetic routines and routines to convert between integer and floating point formats.

Arguments to and results from the arithmetic package, are passed on a software stack. BHA is used as a stack pointer. As with the hardware stack, the stack pointer points to the first unused location, and the stack grows downwards.

Floating point numbers are stored in a five-byte packed format. The format is one-byte exponent, followed by a four-byte mantissa in the order most significant to least significant byte. The exponent is held in excess-128 form, ie the power of two to which the mantissa must be raised, plus 128. The most significant (first) byte of the mantissa holds the sign of the whole number. Numbers are normalised such that there is an implied binary. 1 just to the left of bit six of the MSB of the mantissa.

As an example of a number in floating point representation, the diagram below shows the five bytes used to represent -1.5

Byte	Binary	Decimal
Exponent	10000001	129
Mantissa 1	11000000	192
Mantissa 2	00000000	0
Mantissa 3	00000000	0
Mantissa 4	00000000	0

The derivation of this is as follows. 1.5 in binary is 1.1. In normalised form this is .11 times two ( $2^1$ ). Thus the exponent is  $128+1=129$ . Bit seven of the mantissa is the sign bit of the whole number. It is negative in this case, so the sign bit is a one. Next comes the implied .1 of the normalised number. Bit six of mantissa 1 forms the .01 part of the number. The rest of the mantissa is zero.

The value of zero has the special representation of all zeroes in the exponent and mantissa bytes.

Integers are stored as four-byte two's complement values. The least significant byte is stored at the lowest memory address and the most significant byte is stored last.

In common with most of the functions supported by the operating system, the floating point package used the X register's contents as a reason code. The rest of this chapter describes the entry and exit conditions of the routines. It is assumed that in all cases X contains the reason code and BHA points to a stack containing the arguments. On exit BHA is updated and the stack contains the result if applicable.

The notation STACK(1) is used to denote the item at the top of the stack, STACK(2) for the next item, and so on. Some of the routines generate errors. An error is indicated by the carry flag being set on exit, with an error code in the X register.

# ARIAD • Integer plus

This takes two four-byte integer operands and returns their sum.

<b>Entry:</b>	STACK(1)	LHS of a+b
	STACK(2)	RHS of a+b
	X	ARIAD
<b>Exit:</b>	STACK(1)	Result of a+b

# ARISB - Integer subtract

This takes two four-byte integers and returns their difference.

Entry:	STACK(1)	LHS of a-b
	STACK(2)	RHS of a-b
	X	ARISB
Exit:	STACK(1)	Result of a-b

# ARING - Integer negate

This negates the integer operand on the stack. Note the largest negative integer, -2147483648 will return an incorrect result as there is no representable positive integer of the same magnitude.

Entry:      STACK(1)              Operand  
              X                  ARING  
Exit:        STACK(1)              O-operand

## **ARIML - Integer multiply**

This multiplies the two four-byte integer numbers on the stack and returns their product. Although the operands occupy four bytes each, they must lie in the two-byte range of -32768 to +32767. The result may occupy the whole of the four-byte integer range.

**Entry:**      STACK(1)            LHS of  $a \cdot b$   
                  STACK(2)            RHS of  $a \cdot b$   
                  X                    ARIML

**Exit:**        STACK(1)            Result  $a \cdot b$

# ARFML - Floating point multiply

This multiplies the two five-byte floating point numbers on the stack and returns their product. An error will be given if the product's magnitude is greater than the maximum representable by the floating point format (approximately 1.7E38).

Entry:	STACK(1)	LHS of a*b
	STACK(2)	RHS of a*b
	X	ARMFL
Exit:	C=0	No error occurred
	STACK(1)	Result a*b
or	C=1	Overflow
	X	Error code

# ARFDV

- Floating point divide

This divides the two five-byte floating point numbers on the stack and returns their product. An error is given if the righthand operand is zero.

<b>Entry:</b>	STACK(1)	LHS of a/b
	STACK(2)	RHS of a/b
	X	ARFDV
<b>Exit:</b>	C=0	No error
	STACK(1)	Result a/b
or	C=1	Divide by zero error
	X	Error code

# ARFPW - Floating point power

This raises one floating point operand to the power given by the other one, leaving the result on the stack. If the lefthand operand is negative and the righthand operand is not an integer, an error is given.

Entry:	STACK(1)	LHS of $a^b$
	STACK(2)	RHS of $a^b$
	X	ARFPW
Exit:	C=0	No error occurred
	STACK(1)	Result of $a^b$
or	C=1	An error occurred
	X	Error code

# ARSIN - Sine function

This unary operation returns the sine of its argument. The argument is in radians. The error 'Accuracy lost' is given if the operand's magnitude is greater than 8838608.

Entry:	STACK(I)	a
	X	ARSIN
Exit:	C=0	No error occurred
	STACK(I)	SIN(a)
or	C=1	An error occurred
	X	Error code

# ARCOS

- Cosine function

This unary operation returns the cosine of its argument. The argument is in radians. The operand should be in the range +/-8838608 to avoid 'Accuracy lost' errors.

Entry:	STACK(1)	a
	X	ARCOS
Exit:	C=0	No error occurred
	STACK(1)	COS(a)
or	C=1	An error occurred
	X=Error code	

# ARTAN - Tangent function

This unary operation returns the tangent of its argument. The argument is in radians. The maximum magnitude of the argument is 8838608.

Entry:	STACK(1)	a
	X	ARTAN
Exit:	C=0	No error occurred
	STACK(1)	TAN(I)
or	C=1	An error occurred
	X	Error code

# ARASN - Inverse sine function

This unary operation returns the arcsine of its argument. The result is in radians. The argument must be in the range +/-1 to avoid the 'neg root' error.

Entry:	STACK(1)	a
	X	ARASN
Exit:	C=0	No error occurred
	STACK(1)	ASN(a)
or	C=1	An error occurred
	X	Error code

# ARCOS - Inverse cosine function

This unary operation returns the arccosine of its argument. The result is in radians. The argument should be in the range +/-1.

Entry:	STACK(1)	a
	X	ARCOS
Exit:	C=0	No error occurred
	STACK(1)	ACS(a)
or	C=1	An error occurred
	X	Error code

# ARATN • Inverse tangent function

This unary operation returns the arctangent of its argument. The result is in radians.

Entry:      STACK(I)            a  
              X                    ARATN

Exit:        STACK(I)            ATN(I)

# ARRAD

- Degree to radian conversion

This unary operation converts its argument into radians by multiplying the argument by the conversion factor PI/180.

Entry:      STACK(1)      a  
              X              ARRAD

Exit:      STACK(1)      RAD(a)

# ARDEG • Radian to degree conversion

This unary operation converts its argument into degrees by multiplying the argument by the conversion factor 180/PI.

Entry:	STACK(1)	a
	X	ARDEG
Exit:	STACK(1)	DEG(a)

# ARLOG - Logarithm to the base ten

The unary operation takes the common (base 10) logarithm of its argument, leaving the result on the stack. The argument must be greater than zero to avoid the 'Log range' error.

Entry:	STACK(1)	a
	X	ARLOG
Exit:	C=0	No error occurred
	STACK(1)	LOG(a)
or	C=1	An error occurred
	X	Error code

## **ARLNA - Logarithm to the base E**

This unary operation takes the natural (base E) logarithm of its argument, leaving the result on the stack. If the argument is zero, a 'Log range' error is given.

<b>Entry:</b>	<b>STACK(!)</b>	a
	X	ARLNA
<b>Exit:</b>	<b>C=0</b>	No error occurred
	<b>STACK(!)</b>	LN(a)
<b>or</b>	<b>C=1</b>	An error occurred
	X	Error code

## AREXP - Raising E to a power

This unary operation raises E (2.71828182...) to a given power. If the result is too large to be represented, a 'Too big' error is given.

Entry:	STACK(I)	a
	X	AREXP
Exit:	C=0	No error occurred
	STACK(I)	EXP(a)
or	C=1	An error occurred
	X	Error code

# ARSQR - Square root

This unary function takes the square-root of its argument. The argument must be greater than or equal to zero, otherwise a 'neg root' error will be given.

Entry:	STACK(1)	a
	X	ARSQR
Exit:	C=0	No error occurred
	STACK(1)	SQR(a)
or	C=1	An error occurred
	X	Error code

# ARADD - Floating point addition

This binary operation leaves the sum of its two floating arguments on the stack.

<b>Entry:</b>	STACK(1)	LHS of a+b
	STACK(2)	RHS of a+b
	X	ARADD
<b>Exit:</b>	STACK(1)	Result of a+b

# ARCMP • Floating point compare

This compares two real numbers on the stack.

Entry:	STACK(1)	LHS of a-b
	STACK(2)	RHS of a-b
	X	ARCMP
Exit:	STACK(1)	Status of a-b

Note: The status on the stack is a one-byte value which, when loaded into the processor status register, will reflect the state of the result of the subtraction, ie C=1 implies  $a \geq b$ , Z=1 implies  $a=b$ .

# ARNRM - Floating point normalise

This routine normalises the real number on the stack. When using the other calls in this module, the programmer will also be supplied with the normalised result. This routine is therefore provided for users who perform their own manipulations on floating point numbers and require a normalisation routine.

Entry:	STACK(I)	a
	X	ARNRM
Exit:	STACK(I)	Normalised(a)

## **ARFIX - Floating point to integer conversion**

This routine performs conversion between five-byte floating point representation and four-byte integer representation. If the floating point number is outside of the range -2147483648 to +2147483647, a 'Too big' error is generated.

<b>Entry:</b>	<b>STACK(1)</b>	<b>a</b>
	<b>X</b>	<b>ARFIX</b>
<b>Exit:</b>	<b>C=0</b>	No error occurred
	<b>STACK(1)</b>	INT(a)
or	<b>C=1</b>	An error occurred
	<b>X</b>	Error code

## **ARFLT** - Integer to floating point conversion

This routine performs the opposite conversion to the previous routine. It takes a four-byte integer value and replaces it with a five-byte floating point value of the same sign and magnitude.

Entry:	STACK(1)	a
	X	ARFLT
Exit:	STACK(1)	FLOAT(a)

U

## **10. General calls**

General calls are described on the following pages.

# OPCOM command line interpret string at given address

Calls to the command line interpreter should be made either in the normal way using star commands or using the OPCOM COP call.

Note that, on Communicator, abbreviations of star commands are not allowed, so \*h. <RETURN> will not call \*help.

**Action:** This call sends the address of a command line string to the operating system's command line interpreter. The string must be terminated by CR (ASCII &0D).

**On entry:** BHA points to the start of the command line.

**On exit:** No registers preserved

**To call:**

COP	
EQUB @OPCOM%	from BASIC
OPSYS \$OPCOM	from MASM

**Example:**

170 PHK	\ Push the high byte of the program
180 PLB	\ counter on the stack.
185 BNK P%&10000	\ Pull this byte into the bank register.
190 PER stringaddress%	\ B is now set to the current bank.
195	\ Tell assembler that B = K.
200 PLA	\ Push the 2 bytes of the address of
210 SWA	\ the label stringaddress% on the stack.
220 PLA	\ Pull the low byte into A.
230 SWA	\ Swap A with H.
240 COP	\ Swap A with H to get the correct 16 bit value in AH.
250 EQUB @OPCOM%	\ BHA now points to the start address of the string.
260 RTL	\ Execute COP call.
270 .stringaddress%	\ Command line interpret the string
280 EQUUS "exec file"	\ at the address given in BHA.
290 EQUB &00	\ Return from subroutine to avoid executing
	\ the first character in the string as an op-code.
	\ Set the label stringaddress%.
	\ Place the command line string
	\ starting at this label.
	\ Terminate the string with &00.

This example will perform the function 'exec file'.

# OPWRS write inline string

**Action:** Send the following string (terminated by &00) to the VDU drivers.

**On entry:** ASCII string following COP call terminated by &00

**On exit:** DBAXY preserved

**To call:**

```
COP
EQUB @OPWRS%
EQU string$           from BASIC
EQUB &00
OPSYS COPWRS
= "string",0          from MASM
```

**Example:**

```
170 COP                  \ Execute COP call.
180 EQUB @OPWRS%          \ Send the following string to the VDU drivers.
190 EQU "error"           \ EQU to place the ASCII values of a string in the code.
200 EQUB &00                \ EQUB to terminate string with &00.
```

This example will print *error*.

# OPWRA write string at given address

- Action:** Send the string in memory, pointed to by BHA, to the VDU drivers.
- On entry:** BHA = pointer to memory location containing first character of string. String terminated by &00.
- On exit:** DBAXY preserved
- To call:**
- COP
  - EQUB @OPWRA% from BASIC
  - OPSYS £OPWRA from MASM

**Example:**

```

170 PHK           \ Push the high byte of the program
                  \ counter on the stack.
180 PLB           \ Pull this byte into the bank register.
                  \ B is now set to the current bank.
185 BNK P%/&10000 \ Tell assembler that B = K.
190 PER stringaddress% \ Push the 2 bytes of the address of
                        \ the label stringaddress% on the stack.
200 PLA           \ Pull the low byte into A.
210 SWA           \ Swap A with H.
220 PLA           \ Pull the high byte into A.
230 SWA           \ Swap A with H to get the correct 16 bit value in AH.
                  \ BHA now points to the start address of the string.
                  \ Execute COP call.
240 COP           \ Send the string pointed to by BHA to the VDU drivers.
250 EQUB @OPWRA% \ Return from subroutine to avoid executing
                  \ the first character in the string as an op-code.
260 RTL           \ Set the label stringaddress%.
270 stringaddress% \ Place a string starting at this label.
280 EQUUS "message"
290 EQUB &00       \ Terminate the string with &00.

```

This example will print *message*.

Note that this example is written for 8-bit mode operation and will require modification to run in 16-bit mode.

# OPERR system error

**Action:** This is used as a debugging aid. When the call is executed, a fatal error is generated. A zero-terminated string immediately following the COP signature is printed, followed by a listing of the contents of the 65SC816's registers. The register string is printed as:

address of COP OPERR signature byte (3 bytes)  
 status (1 byte)  
 D (2 bytes)  
 BHA (3 bytes)  
 X (2 bytes)  
 Y (2 bytes)

The bytes are printed as pairs of hexadecimal digits separated by spaces.

**On entry:** Inline error string.

**On exit:** No exit. The machine "hangs".

**To call:**

```
COP
EQUB @OPERR%
EQUUS "error message"
EQUB &00      from BASIC
OPSYS EOPERR
= "error message",0 from MASM
```

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPERR%	\ System error.
190 EQUUS "Can't find any workspace"	\ Inline error message.
200 EQUB &00	\ Terminated by zero (&00).

This example will print the message "*Can't find any workspace*", followed by a listing of the CPU registers.

# OPRLH read hex number

- Action:** BHA points to a line of hex in memory. This call translates a certain number of hex digits into nibbles stored starting at the location pointed to by DX. The number of nibbles required is in Y.
- On entry:** BHA points to line of hex characters.  
DX points to location at which number is to be stored in direct page.  
Y contains the number of nibbles required.
- On exit:** BHA points to the rest of the line.  
If C = 0 then DX points to the Y nibbles in direct page.  
If C = 1 then the call failed.  
DX, Y preserved
- To call:** COP  
EQUB @OPRLH% from BASIC  
OPSYS £OPRLH from MASM

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPADP%	\ Allocate direct page with the \ following number of free bytes. \ We'll choose 4 bytes.
190 EQUW &0004	\ HA will now contain the address of the direct page.
200 TAD	\ Transfer the direct page address held in HA \ into the direct page register.
210 LDX# &00	\ Load X with zero.
220 PHK	\ DX now points to the 4 bytes allocated.
230 PLB	\ Push the high byte of the program counter.
235 BNK P%&10000	\ Pull this byte into B.
240 LDA# ( <i>line%</i> AND &FF00) DIV &100\	\ Tell assembler that B = K. \ Put the label high byte in A.
250 SWA	\ Put this value in H.
260 LDA# <i>line%</i> AND &FF	\ Put the label low byte in A. \ BHA now points to the label <i>line%</i> .
270 LDY# &08	\ Load Y with the number of nibbles. \ In this case 8 which corresponds \ to a 32 bit integer value.
280 COP	\ Execute COP call.
290 EQUB @OPRLH%	\ Read 8 digits of the hex \ number at <i>line%</i> .
300 RTL	\ Return from subroutine.
310 .line%	\ Set the label <i>line%</i> .
320 EQUUS "1A97CF0F"	\ Set hex number as an example.
330 EQUUS "004DCB23"	\ Set hex number as an example.

This example will allocate 4 bytes in direct page for the result. It then reads 8 digits of hex from memory pointed to by BHA and puts the result in the 4 bytes in direct page. In this example, the number read will be 1A97CF0F. After the call, BHA will point to the rest of the line, in this case the number 004DCB23.

# OPBHA make BHA point to inline string

**Action:** This call returns in BHA the address of the start of the zero-terminated string which follows immediately.

**On entry:** Inline string terminated by zero (&00).

**On exit:** BHA points to the start of the inline string.  
DXY preserved

**To call:**

COP	
EQUB	@OPBHA%
EQU\$ string\$	
EQUB &00	from BASIC
OPSYS £OPBHA	
= "string",0	from MASM

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPBHA%	\ Make BHA point to string which
	\ follows immediately.
190 EQU\$ "cell9"	\ String goes here.
200 EQUB &00	\ Terminated by zero (&00).

This example will put the address of the start of the string "cell9" in BHA, ie the address of the ASCII value of the character "c".

# OPSUM

compute end-around-carry sum

**Action:** Gives a sum of all the bits in a block whose start is pointed to by BHA and whose length in bytes is in Y.

**On entry:** BHA points to the start of the block to be summed.  
Y = length of block in bytes.

**On exit:** If C = 0 then the sum has been computed and the result is in HA.  
If C = 1 then either the length was zero (Y = 0) or all the bytes in the block were zero (HA = 0).  
DX,Y preserved

**To call:**  
COP  
EQUB @OPSUM% from BASIC  
OPSYS EOPSUM from MASM

**Example:**

170 RSP# &30	\ Put processor in word mode.
180 WRD &30	\ Tell assembler that processor is in word mode.
190 PHK	\ Push high byte of program counter on to stack.
200 PLB	\ Pull high byte into bank register.
210 BNK P%&10000	\ Tell assembler that B has changed.
220 LDA# blockstart% AND &FFFF	\ Load the accumulator with \ the block start address.
230 LDY# (blockstart%-blockend%)	\ Load Y with length of block.
240 COP	\ Execute COP call.
250 EQUB OPSUM	\ Sum the block. \ The sum is in HA.
260 RTL	\ End.
270 .blockstart%	\ Start of block.
280 EQU "data"	\ Body of block.
290 .blockend%	\ End of block.

This example will sum the block whose start address is blockstart% and whose end address is blockend%. The sum is returned in HA.

# OPADY add Y to BHA

**Action:** BHA becomes BHA + Y, Y is zeroed.

**On entry:** No requirements

**On exit:** BHA := BHA + Y

Y = 0

DBX preserved

**To call:** COP  
EQUB @OPADY% from BASIC

OPSYS COPADY from MASM

**Example:**

```

170 RSP# &30          \ Put processor in word mode.
175 WRD &30          \ Tell assembler that processor is in word mode.
180 Label%             \ Set arbitrary label.
190 PHK               \ Push high byte of program counter.
200 PLB               \ Pull into B.
205 BNK P%/&10000     \ Tell assembler that B = K.
210 LDA# label% AND &FFFF \ Load HA with address.
220 LDY# &06             \ Load Y with 6.
230 COP               \ Execute COP call.
240 EQUB @OPADY%       \ Add Y to BHA.
250 :                  \ BHA now points to start% + 6.

```

This example will add 6 to the address held in BHA.

# OPVER read OS version number

Action: Puts the operating system version number in A.

On entry: No requirements

On exit: A = version number  
DBXY preserved

To call:  
COP  
EQUB @OPVER% from BASIC  
OPSYS \$OPVER from MASM

Example:

```
170 COP          \ Execute COP call.  
180 EQUB @OPVER% \ Read OS version number into A.
```

This example will read the operating system version number into A.

# OPXKC examine keyboard character

**Action:** If keyboard buffer is not empty then A becomes the first character in the buffer. Note that the character is not removed from the keyboard buffer so repetition of this call will yield the same result. Consequently, a call to OPXKC cannot cause a pre-empt to occur.

**On entry:** No requirements

**On exit:** If C = 0 then A = first character in keyboard buffer.  
If C = 1 then keyboard buffer is empty.  
DX preserved

**To call:** COP  
EQUB @OPXKC% from BASIC  
OPSYS EOPXKC from MASM

**Example:**

170 COP	\ Execute COP call.
180 EQUB @OPXKC%	\ Examine keyboard character.

This example will examine the keyboard buffer. If it is full then the first character will be put in A. If it is empty then C is set.

## **11. BBC similar calls**

Many Communicator calls are similar to BBC OS1.2 calls.

Some, such as OPWRC and OPNLI are very similar (OSWRCH and OSNEWL on the BBC), while others, such as OPOSB (BBC OSBYTE), support only a tiny subset of the BBC functions.

The following COP calls are all similar in varying degrees to calls on the BBC Microcomputer.

# OPWRC write character

**Action:** Send the byte in A to the VDU drivers.

**On entry:** A=character code

**On exit:** DBAXY preserved

**To call:**  
COP  
EQUB @OPWRC% from BASIC  
OPSYS £OPWRC from MASM

**Example:**

```
170 LDA# ASC("J")      \ Load A with ASCII code for J.  
180 COP                \ Execute COP call.  
190 EQUB @OPWRC%       \ Send character in A to VDU drivers.
```

This example will print the letter *J*.

# OPWRS write inline string

**Action:** Send the following string (terminated by &00) to the VDU drivers.

**On entry:** ASCII string following COP call terminated by &00

**On exit:** DBAXY preserved

**To call:** COP

EQUB @OPWRS%

EQUUS string\$

EQUB &00 from BASIC

OPSYS £OPWRS

= "string",0 from MASM

**Example:**

```
170 COP          \ Execute COP call.  
180 EQUB @OPWRS% \ Send the following string to the VDU drivers.  
190 EQUUS "error" \ EQUUS to place the ASCII values of a string in the code.  
200 EQUB &00      \ EQUB to terminate string with &00.
```

This example will print *error*.

## OPNLI newline

**Action:** Send LF CR to the VDU drivers. Line feed is ASCII 10 (decimal), carriage return is ASCII 13 (decimal).

**On entry:** No requirement

**On exit:** DBAXY preserved

**To call:**  
COP  
EQUB @OPNL% from BASIC  
OPSYS £OPNLJ from MASM

**Example:**

```
170 COP          \ Execute COP call.  
180 EQUB @OPNL%  \ Send LF CR to the VDU drivers.
```

This example will give a newline.

# OPCLI

command line interpret string at given address

**Action:** This call sends the address of a command line string to the operating system's command line interpreter. The string must be terminated by CR (ASCII &0D).

**On entry:** EITHER: BYX contains the absolute address of the start of the command line  
OR: Y = 0 and X contains an offset from the direct page register D. The start of the command line is in the direct page at address D+X.

**On exit:** No registers preserved

**To call:**  
COP  
EQUB @OPCLI% from BASIC  
OPSYS £OPCLI from MASM

**Example:**

170 PHK	\ Push the high byte of the program \ counter on the stack.
180 PLB	\ Pull this byte into the bank register. \ B is now set to the current bank.
185 BNK P%&10000	\ Tell assembler that B = K.
190 PER stringaddress%	\ Push the 2 bytes of the address of \ the label stringaddress% on the stack.
200 PLX	\ Pull the low byte into X.
210 PLY	\ Pull the high byte into A. \ BYX now points to the start address of the string.
220 COP	\ Execute COP call.
230 EQUB @OPCLI%	\ Command line interpret the string at \ the address given in BYX.
240 RTL	\ Return from subroutine to avoid executing \ the first character in the string as an op-code.
250 .stringaddress%	\ Set the label stringaddress%.
260 EQUS "help"	\ Place the command line string \ starting at this label.
270 EQUB &0D	\ Terminate the string with CR.

This example will perform the function "help".

If the string is held in direct page then if Y is zero, X is taken as an offset from the direct page register D.

# OPOSB BBC OSBYTE call

**Action:** This call carries out various operations, the specific operation depending on the contents of A on entry. Other data can be passed in X and Y. If results are generated, these are returned in X and Y. Only a very small subset of BBC OSBYTE calls is supported.

**On entry:** A contains the reason code. The reason code determines the function of the call.

**On exit:** X and Y will contain results if the call produces them.  
D preserved

**To call:**  
COP  
EQUB @OPOSB% from BASIC  
OPSYS £OPOSB from MASM

The following BBC compatible OSBYTE calls are supported:

HEX	DEC	FUNCTION COMMENT
00 0	Read OS version	Returns X = 0 for 816MOS
02 2	Select input stream	only keyboard allowed
03 3	Select output stream	only vdu allowed
04 4	Enable/disable editing	BBC compatible
0F 15	Flush buffer	0 kbd and sound ONLY 1 current input buffer
12 18	Reset soft keys	BBC compatible
13 19	Wait for VSYNC	BBC compatible
15 21	Flush specific buffers	0 keyboard buffer 4-7 equivalent - sound buffer
1A 26	handset detection	( Communicator ) supported
75 117	Read VDU status	BBC compatible
76 118	Update keyboard leds	BBC compatible
78 120	Write keys pressed info	BBC compatible
7C 124	Clear ESCAPE condition	BBC compatible
7D 125	Set ESCAPE condition	BBC compatible
7E 126	Acknowledge ESCAPE	BBC compatible
80 128	read buffer info	info on keyboard buffer only
81 129	Read key with time limit	BBC compatible
82 130	Read high order address	BBC compatible
86 134	Read POS and VPOS	BBC compatible
87 135	Read character at cursor	BBC compatible
8A 138	Insert value into buffer	only keyboard buffer
91 145	Get character from buffer	only keyboard buffer
98 152	Examine buffer status	only keyboard buffer
99 153	Insert character into buffer	only keyboard buffer
9A 154	Write to video ULA control	BBC compatible
A0 160	Read VDU variable	similar to BBC - but new addresses
AC 172	( C-series specific )	select .- or * on numeric pad
B1 177	Read/write input stream	only keyboard allowed
B2 178	Read/write key semaphore	BBC compatible
C1 193	Read/write flash counter	BBC compatible - probably lies
C8 200	Read/write ESCAPE,BREAK effect	partly supported ie. escape
C9 201	Read/write keyboard disable	BBC compatible
CA 202	Read/write keyboard status	BBC compatible
D2 210	Read/write sound suppression	BBC compatible 1986.09.09
D4 212	Read/write bell information	electron compatible
D5 213	Read/write bell frequency	BBC compatible 1986.09.09
D6 214	Read/write bell duration	BBC compatible 1986.09.09
D8 216	Read/write soft key length	BBC compatible
DC 220	Read/write ESC character	BBC compatible
E1 225	Read/write function key status	BBC compatible
E2 226	Read/write shifted status	BBC compatible
E3 227	Read/write control status	BBC compatible
E4 228	Read/write shift/cntrl status	BBC compatible
E5 229	Read/write ESC key status	BBC compatible
EC 236	Read/write char dest status	only vdu allowed
ED 237	Read/write cursor edit status	BBC compatible

**Example:**

```
170 LDA# &E3      \ Load A with the reason code.  
180 LDX# &80      \ Load X with data.  
190 COP          \ Execute COP call.  
200 EQUB @OPOS8%  \ Call OSBYTE routine.
```

This example will set the base code of CTRL f keys to &80. That is, f0 will give code &80, f1 will give &81, f2 will give &82, etc.

# OPOSW BBC OSWORD call

**Action:** This call carries out various operations, the specific operation depending on the contents of A on entry. BYX points to a control block in memory, and this block contains data for the call, and will contain results from the call.

**On entry:** EITHER: BYX points to a control block in memory  
OR: Y = 0 and X contains an offset from the direct page register D. The start of the control block is in the direct page at address D+X.  
A contains the reason code. The reason code determines the function of the call.

**On exit:** D preserved

For OPOSW with A = 0 (read line from input)  
Y = line length (including CR if applicable).  
If C = 0 then CR terminated input.  
If C = 1 then ESCAPE terminated input.

**To call:**

COP  
EQUB @OPOSW% from BASIC

OPSYS £OPOSW from MASM

The following BBC compatible OSWORD calls are supported:

hex	dec	function	comments
00	0	Read line from i/p stream	BBC compatible
01	1	Read system clock	BBC compatible
02	2	Write system clock	BBC compatible
03	3	Read interval timer	BBC compatible ) no event
04	4	Write interval timer	BBC compatible ) facility
09	9	Read pixel value	BBC compatible
0A	10	Read character definition	BBC compatible
0B	11	Read palette for given colour	BBC compatible
0C	12	Write palette for given colour	BBC compatible
0D	13	Read graphics cursor posns	BBC compatible

Example:

```
170 PHK
180 PLB
185 BNK P%&10000
190 LDY# (time% AND &FF00) DIV &100
200 LDX# time% AND &FF
210 LDAA &01
220 COP
230 EQUB @OPOSW%
240 RTL
250 .time%
260 EQUUS "00000"
```

\ Push the current bank on to the stack.  
\ Pull this value into the bank register.  
\ Tell assembler that B = K.  
\ Load Y with the high byte of the label *time%*.  
\ Load X with the low byte of the label *time%*.  
\ BYX now points to the label *time%*.  
\ Load A with the reason code.  
\ Execute COP call.  
\ Call OSWORD routine.  
\ Return from subroutine.  
\ Set label *time%* where the value of the system clock is to be placed.  
\ 5 bytes to store time.

This example will read the value of the system clock. The 5 byte result will be placed in memory at *time%+0* (low byte) to *time%+4* (high byte).

# OPBGT BBC OSBGET

**Action:** Read one byte from an open file. The file must previously have been opened using OPOPN.

**On entry:** Y contains the file handle from OPOPN.  
The byte is obtained from the part of the file pointed to by the file pointer. The value of the pointer can be set to any value using OPWSP.

**On exit:** A contains the byte read from the file.  
If C = 1 then the end of the file has been reached and the byte in A is invalid.  
No registers preserved

**To call:** COP  
EQUB @OPBGT% from BASIC  
OPSYS £OPBGT from MASM

**Example:**

170 RSP# &30	\ Put processor in 16-bit mode.
172 WRD &30	\ Tell assembler that processor is in word mode.
175 PHK	\ Push the current bank on to the stack.
180 PLB	\ Pull this value into the bank register.
185 BNK P%&10000	\ Tell assembler that B = K.
190 LDA# filename% AND &FFFF	\ Load HA with the middle and low bytes of the label <i>filename%</i> . \ BHA now points to the label <i>filename%</i> . \ Load Y with the open type (OPENIN).
210 LDY# &40	\ Load Y with the open type (OPENIN).
220 COP	\ Execute COP call.
230 EQUB @OPOPN%	\ Call OPEN routine.
240 BCS end%	\ If file not opened then end.
260 COP	\ Execute COP call.
270 EQUB @OPBGT%	\ Read byte from file into A.
280 .end%	\ Set label for end.
290 RTL	\ Return from subroutine.
300 .filename%	\ Set label <i>filename%</i> .
310 EQUS "filename"	\ Insert filename string.
320 EQUB &0D	\ Filename must be terminated by CR (&0D).

This example will open the file whose name is "filename" for input, and then read the first byte from that file into A.

**This COP must be called in 16-bit XY mode (handles are 16 bits).**

# OPBPT BBC OSBPUT

**Action:** Write one byte to an open file. The file must previously have been opened using OPOPN.

**On entry:** Y contains the file handle from OPOPN.  
A contains the byte to be written.

The byte will be written to the part of the file pointed to by the file pointer, which can be changed using OPWSP.

**On exit:** No registers preserved

**To call:** COP  
EQUB @OPBPT% from BASIC  
OPSYS £OPBPT from MASM

## Example:

170 RSP# &30	\ Put processor in 16-bit mode.
172 WRD &30	\ Tell assembler that processor is in word mode.
175 PHK	\ Push the current bank on to the stack.
180 PLB	\ Pull this value into the bank register.
185 BNK P%&10000	\ Tell assembler that B = K.
190 LDAB filename% AND &FFFF	\ Load HA with the middle and low byte of the label <i>filename%</i> . \ BHA now points to the label <i>filename%</i> .
210 LDY# &C0	\ Load Y with the open type (OPENUP).
220 COP	\ Execute COP call.
230 EQUB @OPOPN%	\ Call OPEN routine.
240 BCS end%	\ If file not opened then end.
260 LDAA byte%	\ Load A with byte to be sent to file.
270 COP	\ Execute COP call.
280 EQUB @OPBPT%	\ Write byte from A into file.
290 .end%	\ Set label for end.
300 RTL	\ Return from subroutine.
310 .filename%	\ Set label <i>filename%</i> .
320 EQUUS "filename"	\ Insert filename string.
330 EQUB &0D	\ Filename must be terminated by CR (&0D).

This example will write the byte contained in *byte%* to the file whose name is "filename".

This COP must be called in 16-bit XY mode (handles are 16 bits).

# OPASC BBC OSASCII

**Action:** Send the byte in A to the VDU drivers. If the byte is &0D (carriage return) then send &0A &0D to the VDU drivers (line feed + carriage return).

**On entry:** A = character code.

**On exit:** DBAXY preserved

**To call:** COP  
EQUB @OPASC% from BASIC  
OPSYs EOPASC from MASM

**Example:**

170 LDA# &0D	\ Load A with the ASCII code for \ a carriage return.
180 COP	\ Execute COP call.
190 EQUB @OPASC%	\ Send contents of A to VDU drivers. \ If A contains &0D then send &0A first.

This example will give a newline, same as OPNLI.

