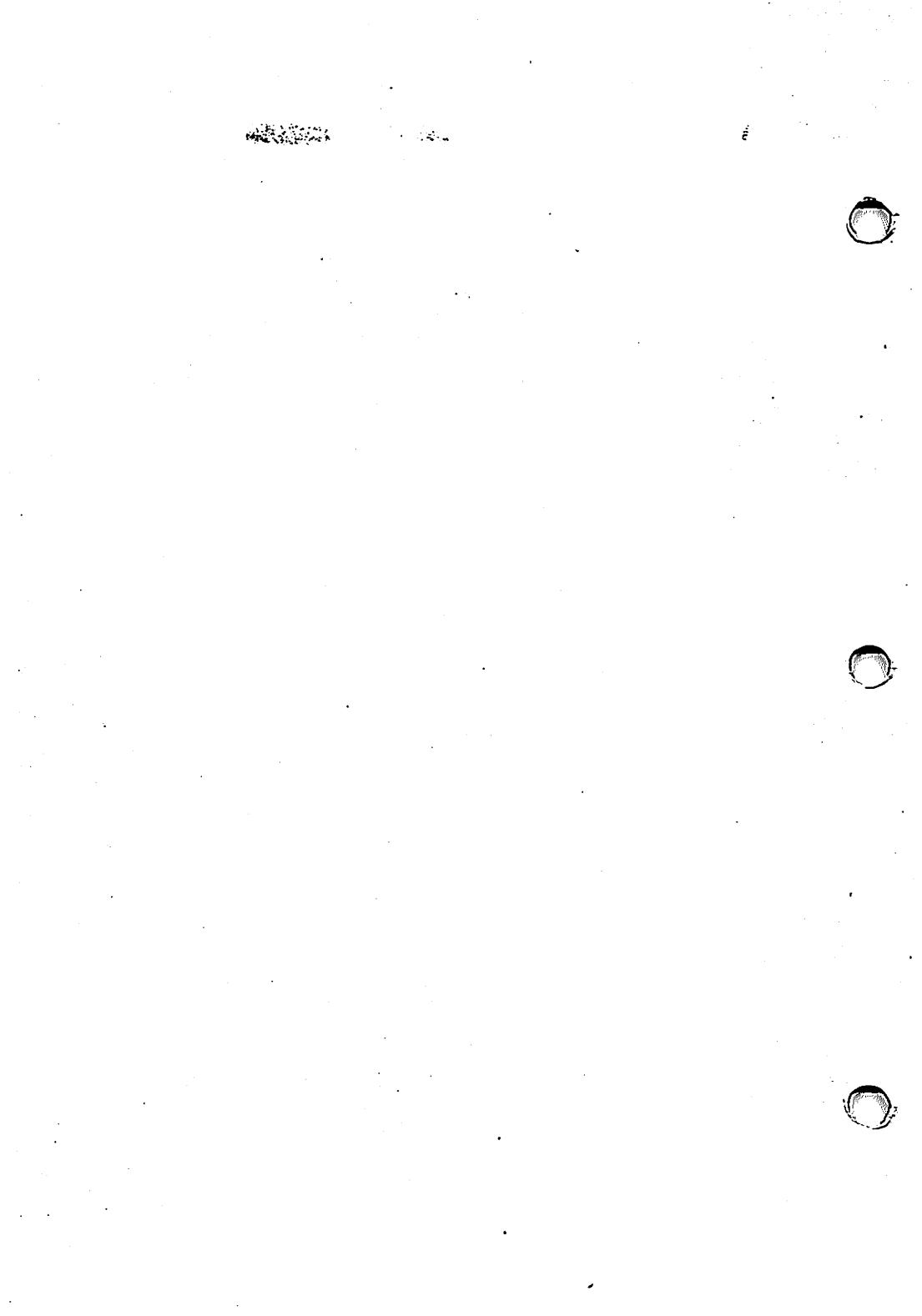


---

## **Communicator Software Tools**

---

Part No 0452,013  
Issue 1  
25 August 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:

Customer Services Department,  
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.

The approval of Communicator for connection to telephone networks is INVALIDATED if it is subject to any modification in any material way not authorised by BABT or is used with, or connected to:

- (i) internal software that had not been formally accepted by BABT
- (ii) external control software or external control apparatus which causes the operation of the modem or associated call set-up equipment to contravene the requirements of the standards set out in BABT/SITS/82/005S/D.

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



# Contents

1. Introduction	1.1
1.1 Hardware	1.1
1.2 Software	1.2
1.3 Macro and Definition Files	1.3
1.4 BASIC Files	1.3
1.5 Documentation	1.4
1.6 Safety and approvals	1.6
2. Introduction to Communicator Programming	2.1
2.1 An Overview of Communicator Systems Software	2.1
2.1.1 Software Levels	2.1
2.1.2 Software Categories	2.2
2.1.3 The Menu Module	2.2
2.1.4 The Terminal Session Software	2.2
2.2 The Hardware	2.3
2.2.1 Processor	2.3
2.2.2 Memory	2.3
2.3 Using BBC BASIC	2.5
2.3.1 Extensions to BBC BASIC	2.5
2.3.2 Enhancement to the ON ... statement	2.5
2.3.3 The LIST IF construction.	2.5
2.3.4 The TIMES pseudo-variable	2.5
2.3.5 The EXT = statement	2.5
2.3.6 The COLOR statement	2.5
2.3.7 Extensions to the assembler	2.6
2.3.8 Improved LISTO formatting	2.6
2.3.9 Other minor changes	2.6
2.3.10 Running existing BASIC programs	2.7
2.3.11 The @HELP procedure	2.7
2.3.12 BASIC programs that call machine code	2.7
2.3.13 Calling operating system functions	2.7
2.3.14 Extensions to CALL and USR	2.8
2.3.15 Calling user machine code programs	2.9
2.4 Creating BASIC Programs	2.11
2.4.1 TWIN	2.11
2.4.2 Code compression	2.14
2.4.3 Module Support	2.16
2.5 Using assembly language	2.18
2.5.1 Programmer's model of the 65SC816	2.19
2.5.2 Register descriptions	2.20
2.5.3 65SC816 addressing modes	2.21
2.5.4 The 65SC816 instruction set	2.28
2.5.5 General advice for programmers	2.49
2.5.6 Bank-independent code	2.50
2.5.7 Position independent code	2.51
2.5.8 Jumping to OS entry points	2.54
2.6 Creating Assembler Programs	2.58
2.6.1 Creating Assembler Source Files	2.58
2.6.2 Creating Assembler Object files	2.59
2.6.3 Creating an Executable Module Image.	2.61

2.6.4 Summary	2.63
2.7 Coding Conventions and Library Support	2.64
2.7.1 BASIC	2.64
2.7.2 Assembler	2.66
2.7.3 KILL conventions	2.67
3. TWIN	3.1
3.1 Introduction	3.1
3.1.1 Facilities	3.1
3.1.2 How this chapter is arranged	3.2
3.1.3 Conventions used in this chapter	3.2
3.1.4 Starting TWIN	3.4
3.1.5 The screen layout	3.4
3.1.6 The keyboard	3.9
3.1.7 Leaving TWIN	3.10
3.2 Working with TWIN	3.10
3.2.1 Types of file	3.10
3.2.2 Date stamping TWIN documents	3.11
3.2.3 TWIN and the operating system	3.11
3.2.4 TWIN concurrency	3.13
3.3 Editing	3.14
3.3.1 Insert and overtype modes	3.14
3.3.2 Moving around the text	3.14
3.3.3 Changing the text	3.16
3.3.4 Clearing and restoring text	3.17
3.4 Finding and replacing text	3.17
3.4.1 Simple finding and replacing	3.18
3.4.2 Counting occurrences	3.19
3.4.3 Specifying a target string	3.19
3.4.4 Specifying a replacement string	3.22
3.5 Files	3.23
3.5.1 Saving and loading files	3.23
3.5.2 Printing files	3.26
3.6 TWIN function keys	3.26
3.7 Concurrency: a worked example	3.31
3.7.1 TWIN and AAam, the ARM assembler	3.31
3.7.2 Using a task	3.33
3.8 Error messages	3.33
3.9 Tasks and events	3.34
3.10 Moving TWIN	3.34
3.11 TWIN called from ARM BASIC	3.35
4. The External ROM Expansion Card	4.1
4.1 Introduction	4.1
4.2 The External ROM Expansion Card	4.2
4.2.1 Introduction	4.2
4.2.2 Connecting the External ROM Expansion Board	4.2
4.2.3 The Reset button	4.3
4.2.4 The External EPROM ZIF Sockets	4.3
4.2.5 The Internal RAM	4.4
4.3 The System Utilities	4.4
4.3.1 Introduction	4.4
4.3.2 Invoking the System Utilities	4.4
4.3.3 The CBFREE Utility	4.5
4.3.4 The HANDLES Utility	4.5

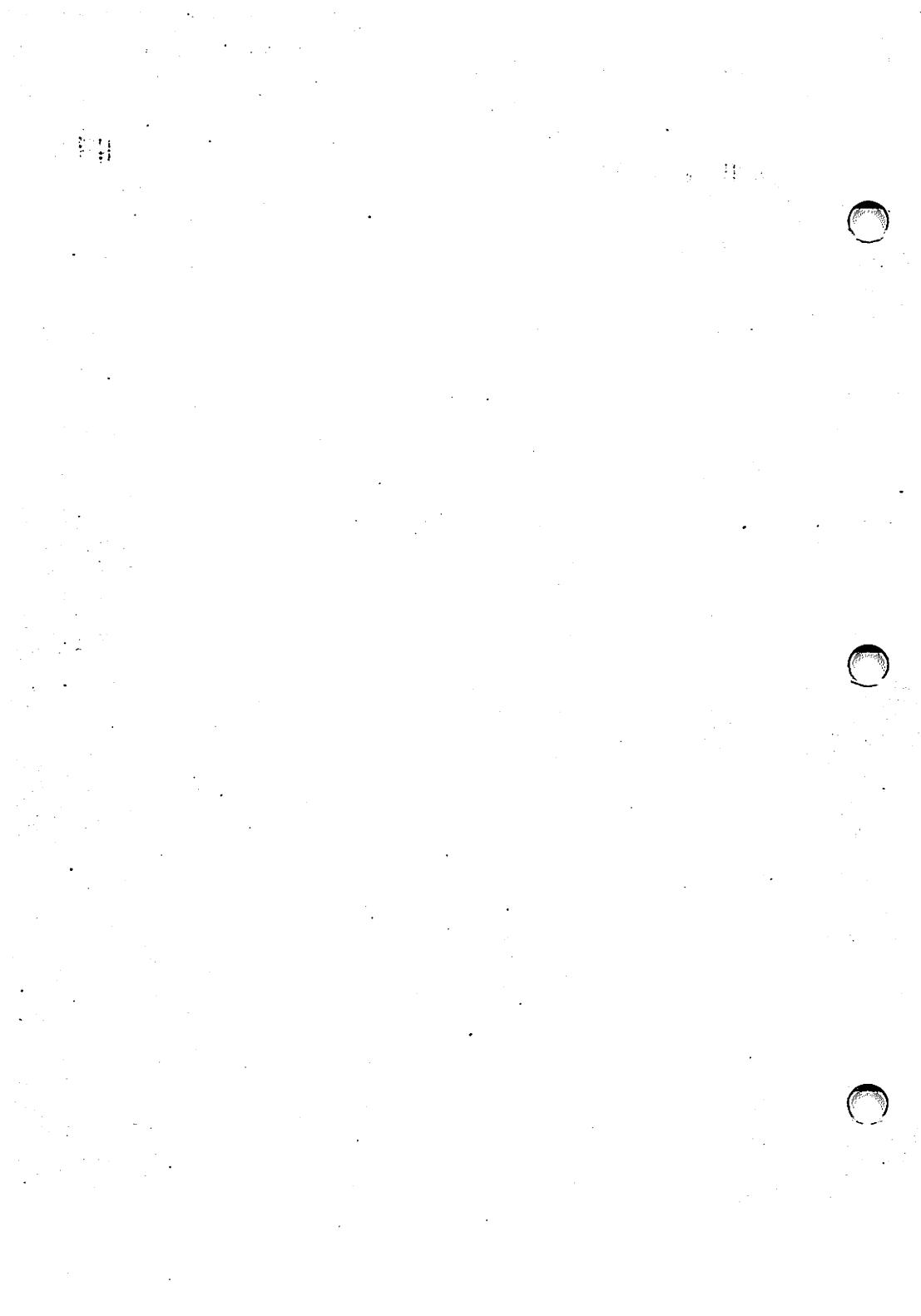
4.3.5 The MAP Utility	4.6
4.3.6 The MODULES Utility	4.7
4.3.7 The IRQLIST Utility	4.9
4.4 The Trace Utility	4.10
4.4.1 Introduction	4.10
4.4.2 Use of breakpoints and reporting	4.10
4.4.3 Looking at memory	4.14
4.4.4 Patching memory and registers	4.15
4.4.5 Memory protection	4.17
4.4.6 Real-time tracing	4.18
4.4.7 Miscellaneous	4.19
5. Support Utilities	5.1
5.1 Introduction	5.1
5.2 UNCOMMENT	5.1
5.3 EDCONVERT	5.1
5.4 TCRUNCH	5.2
5.5 MAKEMOD	5.2
5.6 ALOADER	5.3
6. BlueScreen Software	6.1
6.1 Overview	6.1
6.2 Specification	6.2
6.2.1 Screen format	6.2
6.2.2 Help support	6.2
6.3 User interface	6.3
6.3.1 BlueScreen	6.3
6.3.2 Required interfaces	6.3
6.3.3 Invocation interfaces	6.5
6.4 Environment	6.9
6.5 Structure	6.9
6.6 Data structures	6.10
6.6.1 Textual data structures	6.10
6.7 Procedures	6.10
6.7.1 Util1 routines	6.11
6.7.2 Usage	6.22
6.7.3 Util2 routines - must be provided by user	6.22
6.7.4 Machine code entry points	6.23
6.8 Source Code Management	6.23
6.9 Error Messages	6.24
6.10 OS calls required	6.24
7. Device Drivers	7.1
7.1 Introduction	7.1
7.2 Device Driver Module Reason Codes	7.2
7.3 Device Driver Control Streams	7.3
7.4 Device Driver Filenames and Handles	7.3
7.5 Writing Device Drivers	7.4
7.5.1 Device Driver Reason Codes	7.4
7.5.2 Returning Status	7.4
7.5.3 Device Driver Macros	7.5
7.5.4 Implementing a Device Driver	7.11
8. Communicator emulation modules	8.1
8.1 Overview	8.1
8.2 Software Interface	8.2
8.3 Environment	8.2

8.4 Structure	8.3
8.5 Data structures	8.4
8.6 Procedures	8.5
8.6.1 RCODES	8.5
8.6.2 EMNAME	8.5
8.6.3 EMINIT	8.5
8.6.4 EMKILL	8.5
8.6.5 EMHELP	8.5
8.6.6 EMCGET	8.5
8.6.7 EMCPUT	8.5
8.6.8 checkhandle	8.5
8.6.9 EMcrflags	8.6
8.6.10 EMOPEN	8.6
8.6.11 EMLALA	8.6
8.6.12 EMCLOSE	8.6
8.6.13 gdzero	8.6
8.6.14 nullroutine	8.6
8.7 OS calls required	8.7
8.8 Communicator BBC and Teletype emulation modules	8.8
8.8.1 Overview	8.8
8.8.2 Teletype emulation Specification	8.8
8.8.3 BBC emulation Specification	8.8
8.8.4 Data structures	8.9
8.8.5 Procedures	8.9
8.9 Communicator Videotex emulation module	8.11
8.9.1 Overview	8.11
8.9.2 Videotex emulation Specification	8.11
8.9.3 Data structures	8.12
8.9.4 Procedures	8.12
8.9.5 OS calls required	8.16
8.10 Communicator VT100 emulation module	8.16
8.10.1 Overview	8.16
8.10.2 Specification	8.17
8.10.3 Data structures	8.20
8.10.4 Procedures	8.23
8.11 OS calls required	8.34
9. Application Escape	9.1
9.1 Introduction	9.1
9.2 Overview of facilities	9.1
9.3 Interface description	9.2
9.3.1 Entry conditions	9.2
9.3.2 Exit conditions	9.2
9.4 Responsibilities of the PHONEESC module.	9.3
9.5 Validation of PHONEESC return state by TSHELL	9.3
9.5.1 Killing PHONEESC	9.3
9.6 Closedown	9.4
9.6.1 Normal closedown of TSHELL with PHONEESC coroutine active	9.4
9.6.2 Reporting what has happened	9.4
10. Macros and definition files	10.1
10.1 Introduction	10.1
10.2 The SYSTEM Macros	10.2
10.2.1 Introduction	10.2
10.2.2 The SYSTEM Operational Macros	10.3

10.2.3 The SYSTEM Manifest Macros	10.6
10.3 The DRIVER macros	10.16
10.3.1 Introduction	10.16
10.3.2 The DRIVER Operational Macros	10.16
10.3.3 The Driver Manifest Macros	10.23
10.4 Miscellaneous GET Files	10.24
10.4.1 Introduction	10.24
10.5 An Example Module	10.25
10.5.1 DummyMacro	10.25
10.5.2 Dummy00	10.26
10.5.3 Dummy05	10.28
10.5.4 The MAKE File	10.33
11. Modem driver and BABT requirements	11.1
11.1 Introduction	11.1
11.2 Modem Driver Commands	11.2
11.2.1 Introduction	11.2
11.2.2 Baud rate and modem mode	11.3
11.2.3 Dialling	11.4
11.2.4 Ring Detection and Auto-Answer Facilities	11.5
11.2.5 Handshake Protocol	11.6
11.2.6 The Idle Line Watchdog	11.6
11.2.7 Buffer Functions	11.8
11.2.8 Carrier Detection	11.8
11.2.9 Data Format Commands	11.9
11.2.10 Miscellaneous Commands	11.10
11.3 Accessing the Driver, Typical Call Sequences	11.12
11.3.1 Opening the Modem	11.12
11.3.2 Starting Data Calls	11.12
11.3.3 Starting Voice Calls	11.12
11.3.4 Answering calls	11.12
11.3.5 Transmitting and Receiving data	11.13
11.4 Finite State model of modem behaviour	11.13
11.4.1 Transition 0	11.13
11.4.2 Transition 2	11.14
11.4.3 Transition 4	11.14
11.4.4 Transition 6	11.14
11.4.5 Transition 8	11.14
11.4.6 Transition 10	11.14
11.4.7 Transition 12	11.14
11.4.8 Transition 14	11.14
11.5 Guidelines to BABT Requirements	11.15
11.5.1 introduction	11.15
11.5.2 Dialling	11.15
11.5.3 Connection	11.15
11.5.4 Answering	11.15
11.5.5 Disconnection	11.15
12. MASM	12.1
12.1 General source file format	12.2
12.2 The 65SC816 mnemonics	12.3
12.3 The addressing mode suffixes	12.5
12.4 Banks and bank-independence	12.6
12.5 MASM expressions	12.6
12.5.1 Types of operand	12.6

12.5.2 Symbols	12.6
12.5.3 Numeric constants	12.7
12.5.4 String constants	12.7
12.5.5 Variables	12.7
12.5.6 Location counters	12.7
12.5.7 Numeric operators	12.8
12.5.8 String operators	12.9
12.5.9 Logical and comparison operators	12.9
12.5.10 Operator precedence	12.10
12.6 Common MASM directives	12.11
12.6.1 ASSERT - Make an assertion	12.11
12.6.2 BANK - Set assumed data bank value	12.11
12.6.3 EMODE - Assemble for emulate mode	12.12
12.6.4 END - end of assembly	12.12
12.6.5 LNK - link next source file	12.12
12.6.6 LONG - Allow long instructions	12.12
12.6.7 NOBANK - Prevent assumed bank register	12.12
12.6.8 NOLONG - Disallow long instructions	12.13
12.6.9 OPT - set listing options	12.13
12.6.10 ORG - set code origin	12.13
12.6.11 RMODE - Reset one-byte register bits	12.13
12.6.12 ROUT - Declare start of routine	12.14
12.6.13 SMODE - Set one-byte register bits	12.14
12.6.14 TTL - set listing title	12.15
12.6.15 [ - start conditional assembly section	12.15
12.6.16 ! - Print message	12.15
12.6.17 * - Define a symbol	12.15
12.6.18 ^ - Set variable code origin	12.15
12.6.19 = - Reserve (variable) memory bytes	12.15
12.6.20 = - Insert bytes into code space	12.16
12.6.21 & - Insert double bytes into code space	12.16
12.6.22 % - Fill code space with zeroes	12.16
12.7 Variables	12.17
12.7.1 Types of variable	12.17
12.7.2 Assigning variables	12.17
12.7.3 String variable substitution	12.17
12.7.4 Local variables	12.18
12.8 Macros	12.19
12.8.1 Defining macros	12.19
12.8.2 Macro parameters	12.19
12.8.3 Default parameters and missing parameters	12.20
12.8.4 Nesting macro calls	12.21
12.8.5 Local variables	12.21
12.8.6 Macro substitution method	12.22
12.8.7 The MEXIT directive	12.22
12.9 The WHILE loop	12.23
12.10 Using MASM	12.24
12.10.1 Starting MASM	12.24
12.10.2 Preparing to assemble	12.24
12.10.3 LENGTH	12.24
12.10.4 WIDTH	12.25
12.10.5 PRINT	12.25
12.10.6 STOP	12.25

<b>12.10.7 TERSE</b>	<b>12.26</b>
<b>12.10.8 MLEVEL</b>	<b>12.26</b>
<b>12.10.9 XREF</b>	<b>12.26</b>
<b>12.10.10 Assembling a file</b>	<b>12.26</b>
<b>12.10.11 SYMBOL</b>	<b>12.27</b>
<b>12.10.12 GET</b>	<b>12.27</b>
<b>12.10.13 SAVE</b>	<b>12.28</b>
<b>12.11 Using XREF</b>	<b>12.28</b>
<b>12.11.1 ADD</b>	<b>12.29</b>
<b>12.11.2 CLEAR</b>	<b>12.29</b>
<b>12.11.3 INIT</b>	<b>12.29</b>
<b>12.11.4 LIST</b>	<b>12.29</b>
<b>12.11.5 XREF</b>	<b>12.29</b>
<b>12.11.6 RESULT</b>	<b>12.30</b>
<b>12.11.7 SUMMARY</b>	<b>12.30</b>
<b>12.12 MASM errors</b>	<b>12.31</b>
<b>12.12.1 Non-fatal errors</b>	<b>12.31</b>
<b>12.12.2 Fatal errors</b>	<b>12.34</b>
<b>12.13 MASM - Problems</b>	<b>12.37</b>
<b>12.13.1 Assignment</b>	<b>12.37</b>
<b>12.13.2 Local labels within macros</b>	<b>12.37</b>
<b>12.13.3 Duplicated local labels</b>	<b>12.37</b>
<b>12.13.4 Null strings variables in labels</b>	<b>12.38</b>
<b>12.13.5 Macro calls within macros</b>	<b>12.38</b>
<b>12.13.6 Commented out macro definitions</b>	<b>12.39</b>
<b>12.13.7 LDYZ</b>	<b>12.39</b>
<b>12.13.8 Direct page offset addressing</b>	<b>12.39</b>
<b>12.13.9 Bad direct page with STZZ</b>	<b>12.40</b>
<b>12.13.10 The GET directive</b>	<b>12.40</b>
<b>12.13.11 The BANG (!) directive</b>	<b>12.40</b>
<b>12.13.12 GETting non-existent files</b>	<b>12.40</b>
<b>12.13.13 Code size</b>	<b>12.40</b>



# **1. Introduction**

This Chapter serves as an introduction for users of the Communicator OEM development package. It describes in brief the hardware, the software and the documentation which make up the development system.

## **1.1 Hardware**

The Communicator OEM development package is supplied along with the following hardware:

- BBC Master 128 Microcomputer
- Floppy Disc Drive
- Hard Disc Drive
- ARM Second Processor
- Monochrome (composite video) monitor
- External ROM Expansion Card
- Appropriate cables

The required software, listed below, is supplied on an ADFS format floppy disc. In addition, to test the software being developed, a Communicator will also be required.

## 1.2 Software

The following items of software are provided:

<b>TWIN</b>	This is a powerful screen-based editor. It is used to create programs written in 65SC816 assembler. It may also be used as an editor for text purposes, if a suitable text formatter is available.
<b>MASM</b>	A macro assembler for 65SC816 assembly language. It is very fast, as all processing takes place with both the source and object file in memory. Multiple source files may be linked to allow the assembly of large programs. The assembler supports macros, assembly-time variables, conditional and repetitive assembly. Symbol tables in alpha or numerical order are supported.
<b>XREF</b>	This utility is used in conjunction with MASM to provide a fully cross-referenced listing of identifiers used in MASM source files.
<b>TRACE and SYSUTILS</b>	These are contained in a ROM on the external ROM expansion card and may be called at any time by issuing the appropriate star (*) commands. Trace is a machine code monitor with break pointing and powerful emulation features. Sysutils are a set of useful utilities for displaying system resources.
<b>Basic support utilities</b>	These are supplied on disc and are used to compress and generally increase the efficiency of BASIC programs prior to including them as firmware.
<b>Loaders</b>	These are used to create loadable assembler object and BASIC modules.
<b>65TURBO</b>	This is a 6502 emulator package for the ARM second processor. The assembler and support utilities expect to be run on a 6502 based machine.

### **1.3 Macro and Definition Files**

A set of macro and definition files are supplied as part of the OEM development package to provide support for OEM applications written in assembler. These files contain the necessary macros, system constants and data structures to allow OEM developers to interface their code efficiently and cleanly to existing Communicator software such as the MOS.

The following files are supplied and should be located in the Winchester disc directory \$S.get:

```
SYSTEM  
SYS2  
SYS2A  
SYS3  
SYS4  
decode01  
dtd01  
FILESYSTEM  
hardware  
OS  
FX
```

### **1.4 BASIC Files**

Three BASIC source files are supplied as part of the OEM development package. These files provide support for the Communicator BlueScreen Software.

The following files are supplied and are located in the Winchester disc directory \$S.BAS\_UTILS:

```
GN4_UTIL1  
I_util1  
GN3_UTIL2
```

See the BlueScreen Chapter for more information.

## **1.5 Documentation**

In addition to this introduction, the following Chapters document the various software components in the system, and for the loader hardware:

### **Chapter 2 : Introduction to Communicator Programming**

This contains information for programmers writing new software, or converting software from other machines. Software written in BBC BASIC or 6500 family assembler may be converted to run on the current machine with a small amount of effort.

It details the instruction set and architecture of the 65SC816 processor. It gives the assembly format of instructions for both the MASM and BASIC assemblers. There is also a section on converting BASIC programs to run under the new version.

The introduction covers the software style that should be adopted for developing Communicator code and the restrictions imposed by some of the utilities provided.

### **Chapter 3 : TWIN**

TWIN is a program which runs on the development package BBC Microcomputer for producing source files.

### **Chapter 4 : The External ROM Expansion Card**

The External ROM Expansion Card package allows users to test and develop their own EPROM based software. In addition the board is supplied with a set of ROM based utilities, including TRACE, a debugging utility for machine code programs running on Communicator. It provides single step, register/memory examination and alteration and disassembly of store.

### **Chapter 5 : The BASIC Support Utilities**

This Chapter describes the utilities provided to assist in the creation of efficient BASIC modules. The following utilities are provided:

<b>TCRUNCH</b>	-	BASIC program compressor
<b>UNCOMMENT</b>	-	Strips comments from a BASIC program
<b>EDCONVERT</b>	-	Converts a BASIC constant definitions file into a DEMANIFEST TWIN command file.
<b>MAKEMOD</b>	-	Turns a BASIC program into a module
<b>ALOADER</b>	-	Generic loader. Creates a single loadable object ROM image from a set of a set of assembler object files.

### **Chapter 6 : The BlueScreen Software**

This covers the BlueScreen menu software and how to integrate user developed applications with it.

### **Chapter 7 : The Device Driver**

This guide covers device drivers and how to implement them.

### **Chapter 8 : Emulation Modules**

The Emulation Module Guide covers the current terminal emulation modules, how they operate, how they relate to the rest of the system and how to implement new ones.

### **Chapter 9 : Application Escape**

This covers the Application Escape facility which enables OEMs to develop their own terminal session applications and install them into the standard Communicator phone handling software.

### **Chapter 10 : Macros and Definition Files**

This covers the assembler Macros and Definition files supplied with the OEM development package, what they are and how to access them.

### **Chapter 11 : The Modem Driver and BABT Requirements**

This Chapter covers the Modem Driver and guidelines to writing applications that conform to BABT requirements.

### **Chapter 12 : MASM**

This Chapter describes the powerful 65SC816 cross-assembler which runs in the development machine. Code produced by this assembler may be transferred into Communicator using the loader software described in the next Chapter. It may then be executed under TRACE.

### ***Further Reference***

Other manuals useful for software development with Communicator are:

- The Communicator Systems Manual** - an in-depth discussion of the internals of Communicator's operating system.
- The Communicator Service Manual** - describes the hardware of the machine in detail. A circuit description, assembly and disassembly, and fault finding are included.

## 1.6 Safety and approvals

The mains power supply supplied with Communicator is specified to operate as follows:

Input:	240 Volts a.c. 50Hz at 45W
Output:	21 Volts a.c. 50Hz at 1.6A

The power supply should be disconnected from the mains supply when the computer is not used for long periods.

**IMPORTANT:** The wires in the mains lead for the power supply unit are coloured in accordance with the following code:

BLUE - NEUTRAL

BROWN - LIVE

As the colours of the wires may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows when replacing the plug:

The wire which is coloured blue must be connected to the terminal which is marked with the letter N, or coloured black or blue.

The wire which is coloured brown must be connected to the terminal which is marked with the letter L, or coloured red or brown.

If the socket outlet available is not suitable for the plug supplied, the plug should be cut off and the appropriate plug fitted and wired as previously noted. The moulded plug which has been cut off must be disposed of as it would be a potential shock hazard if it were to be plugged in with the cut end of the mains cord exposed.

The moulded plug must be used with the fuse and fuse carrier firmly in place. The fuse carrier is of the same basic colour (see note) as the coloured insert in the base of the plug. Different manufacturers' plugs and fuse carriers are not interchangeable. In the event of loss of the fuse carrier the moulded plug MUST NOT be used. Either replace the moulded plug with another conventional plug wired as previously described, or obtain a replacement fuse carrier from an authorised supplier. In the event of the fuse blowing, it should be replaced, after clearing any faults, with a 3A fuse that is ASTA approved to BS 1362.

This power supply is designed and manufactured to comply with BS415, BS6301 and BS6484. In order to ensure the continued safety of this item it should be returned to an authorised supplier for repair.

Do not use the supply in conditions of extreme heat, cold, humidity, dust or vibration.

**Note:** Not necessarily the same shade of that colour.

The Acorn Communicator is made in the United Kingdom by :

Acorn Computers Ltd  
Fulbourn Road  
Cherry Hinton  
Cambridge CB1 4JN

The approval of Communicator for connection to the British Telecom Public Switched Telephone Network is INVALIDATED if the apparatus is subject to any modification in any material way not authorised by BABT or is used with, or connected to:

- (i) internal software that had not been formally accepted by BABT.
- (ii) external control software or external control apparatus which causes the operation of the modem or associated call set-up equipment to contravene the requirements of the standard set out in BABT/SITS/82/005S/D.

All apparatus connected to this modem and thereby connected directly or indirectly to the British Telecom Public Switched Telephone Network must be approved as defined in Section 16 of the British Telecommunications Act 1981.

Approval of this apparatus will be INVALIDATED by the use of any power supply that does not comply to BS6301.



## 2. Introduction to Communicator Programming

This Chapter is intended as an introduction for software developers. Most attention is paid to BASIC and assembly language, as it is envisaged that these will be the most frequently used languages.

Section 2.1 is a short overview of Communicator System Software components, what they are and how they interrelate. More information can be garnered from the Communicator Systems Manual.

Section 2.2 is a short introduction to the hardware of the machine. It only covers enough detail to tell the programmer which facilities are available. Readers interested in the lower-level details of the machine's circuitry should consult the Service Manual.

Section 2.3 discusses BASIC programs. Because the version of BASIC used is a dialect of BBC BASIC, many existing programs for the BBC Microcomputer should be easy to transport. Minor amendments may have to be made, for example to deal with the HELP key. Programs which call machine code routines or the operating system will need minor changes.

Section 2.4 introduces the user to the facilities available for the creation of BASIC programs within the OEM Development System.

Section 2.5 is for assembly language programmers. It describes the machine's 65SC816 processor. The addressing modes and instruction types are discussed, and the mnemonic formats for both the MASM and BBC BASIC assemblers described. For further details on the assembler, consult the MASM Guide, or the BASIC User Guide. This section also provides a section of 'hints and tips' for assembly language programmers.

Section 2.6 covers the facilities provided by the OEM Development System for the creation of assembler modules.

Section 2.7 gives some brief descriptions of procedures to be followed when designing BASIC or assembler programs for Communicator.

It is strongly recommended that developers thoroughly explore the Communicator Systems Manual before attempting to read this Chapter.

### 2.1 An Overview of Communicator Systems Software

This section gives a brief description of some of the elements comprising Communicator Systems Software.

#### 2.1.1 Software Levels

At the lowest level, all software resident on the Communicator exists as Modules. A module is a sequence of bytes, stored in ROM or RAM, starting with a Module Header. The module header contains information describing the contents of the module. Modules provide the Communicator MOS with the basic information it requires to locate and if necessary invoke software entities within the system. Modules are independently assembled; the MOS provides the linkage mechanism for passing control between modules.

At a higher level than modules are coroutines. These are loosely coupled processes which form part of a hierarchy of similar coroutines. A coroutine has its own stack and direct page. A coroutine may or may not be a task.

A task is a program that can be run from Communicator Top-Level Menu. A task may be comprised of one or more coroutines.

### **2.1.2 Software Categories**

Communicator software falls into five categories:

- (1) The MOS (Communicator Operating System)
- (2) Terminal emulation software
- (3) Device driver software
- (4) Filing system software
- (5) Applications software

Most OEM developed programs will fall into the fourth category.

More detail on the above can be found in the Chapters Terminal Emulation, Blue-Screen Software and Device Drivers, elsewhere in this manual. See also the Communicator Systems Manual.

### **2.1.3 The Menu Module**

One of the more important modules to the applications programmer is MENU. Menu is a high-level piece of software that allows a user to start and stop tasks. Menu conforms to the blue-screen user interface.

Menu provides the user interface to the MOS's memory management and coroutine facilities allowing multiple task creation, invocation and deletion. The menu module maintains a flow of control between tasks allowing task switching via preempt keys.

OEMs will often want to be able to invoke their applications from menu and should design their code so that it conforms to menu's modus operandi. Software should be implemented as one or more routines which should be preemptible by the user so that control can be passed back to menu.

The configure menu should be used to include the application as an option within the top-level menu.

See the Communicator Systems Manual for more information on Menu and task switching.

### **2.1.4 The Terminal Session Software**

A major proportion of Communicator software is directed toward providing a phone based terminal session service. The components of this software include device drivers (e.g. MODEM:), a built-in phone directory (PHONE:) with auto-dial, a terminal handling user interface (TSHELL) with associated terminal kernel (KEYPAGE) and finally terminal emulation modules to customise the actions of keypage so as to make it conform to specific terminal types. See Chapters 8, Emulation Modes, and 11, Modem Driver and BABT Requirements.

TSHELL provides a facility, Application Escape, which allows OEMs to implement their own terminal session software whilst retaining the directory and auto-dial services provided by the PHONE module. See Chapter 9, Application Escape.

See also the Communicator Systems Manual.

## **2.2 The Hardware**

This section describes the aspects of the machine's hardware which are relevant to the software writer. More detailed discussions (eg circuit descriptions) are given in the Service Manual.

### **2.2.1 Processor**

As usual, the heart of any system is the processor. In the present machine, this is the 65SC816, a development of the popular 6502. The processor has sixteen-bit internal registers and a 24-bit address bus. Externally, the data bus is eight bits (one byte) wide. The 24-bit address range is organised as 256 banks of 65536 ( $2^{16}$ ) bytes each.

The processor may operate in 'emulation' or 'native' mode. In the former, it emulates the timing and addressing of a 6502, but retains the extra instructions in the 65SC816 instruction set. In the latter mode, the full power of the enhanced addressing range and sixteen-bit registers is available. In the current machine, the processor always runs in native mode.

### **2.2.2 Memory**

#### **RAM**

There are three types of random-access memory (RAM), in addition to the read-only memory (ROM) banks. The three types of RAM are:

- (1) 32K bytes CMOS RAM, used for storage of configuration information, (for example telephone numbers), even when the machine is switched off.
- (2) 32K bytes of screen RAM, used to hold the text and graphics being displayed.
- (3) The main banks of dynamic RAM - either 512K bytes or 1M byte.

The first type of RAM is used by the operating system to store important information which must be preserved when the machine is switched off, for example the type of telephone dialling to be used the MODEM. In addition, application software may use the CMOS RAM.

Access to the video RAM is entirely through the VDU drivers, a part of the operating system. As with all RAM, the video RAM may be accessed directly, but in the interests of compatibility with other software which is also using the screen, the VDU drivers should be used for all character and graphics plotting.

The last type of RAM is used by operating system software and applications software. Allocation of main memory is handled entirely by the operating system. This is necessary as more than one program might be accessing the RAM, due to the multi-tasking ability of the machine. It is especially important that bank zero (addresses &000000-&00FFFF) is handled in a well-defined manner, as many uses are made of this space (eg the processor stack and direct-page addressing modes use it).

Because of the bank-based organisation of the memory-map, it is important that programs written for the machine are bank-independent, that is they may run in any of the 256 possible banks without alteration. There are a few simple rules to follow when writing such programs in assembly language (eg avoiding the 'long' addressing modes). BASIC programs are bank-independent by default. The next section describes ways of ensuring that assembly language programs are written in a bank-independent way.

#### **ROM**

The ROM in the machine contains the system software (the machine operating, device drivers etc), languages such as BASIC, and applications programs.

Applications software (either in BASIC or machine code) may also be stored in ROM. This has the advantage of not using any main RAM for program storage. Again, it is important to aim for bank-independence when writing assembly language programs which reside in ROM.

## **Chapter 2**

### ***Memory map***

The memory map of the machine is detailed in the Service Manual. Below is a table of addresses occupied by the various memory elements in the machine:

Main RAM	&000000-&07FFFF (512K DRAM)
Main RAM	&000000-&0FFFFF (1M DRAM)
Video RAM	&450000-&457FFF
CMOS RAM	&460000-&467FFF (32K CMOS)
ROM slot 0	&FE0000-&FFFFFFFFFF
ROM slot 1	&FC0000-&FDFFFFFF
ROM slot 2	&FA0000-&FBFFFFFF
ROM slot 3	&F80000-&F9FFFFFF

### ***Input/output***

The machine is well equipped with IO (input/output) capability. Most of the IO devices are memory mapped. Those which are not are controlled through the IIC bus (these are the DTMF dialler, the real-time clock/calendar and the teletext display controller). Below is a summary of the IO devices.

### ***MODEM***

This provides communication to remote computers through the telephone line. The data formats, transmission speeds and other attributes of the MODEM are configurable in software. Dialling may be performed using either loop-disconnect or DTMF dialling. Some of the MODEM's current operating modes (eg type of dialling) are held in the CMOS RAM, so only have to be set-up once for a given site. Software to perform auto-dialling and auto-answering of data transmission calls is provided.

A telephone handset may be fitted to the side of the machine. This works in parallel with the MODEM.

### ***Printer***

A standard parallel printer port is available. Software is included to drive the port from BASIC or machine code programs.

### ***RS423***

An RS423 (RS232-compatible) serial port is included. This may be used for driving serial printers or other serial devices (eg external MODEMs). Software is present to configure the send/receive speed of the port, its data format and parity bits.

### ***Econet***

A standard interface for the Econet local area network is present. This enables the machine to access a remote FileStore for mass storage of programs and data. Communication is also possible between various machines on the same network. Speeds of up to 200K bits per second are possible. A network has a size limit of 500 metres, though several networks may be connected together using 'bridges'.

Software support for Econet is through the Network Filing System (NFS). This provides a command line interface (eg \*LOAD) and a machine code interface to a versatile, hierarchical filing system.

## **2.3 Using BBC BASIC**

There are several ways of writing applications software to run on the machine. It is envisaged that BASIC, assembly language and Pascal will be the most frequently used languages.

Because the BASIC is a dialect of the popular BBC BASIC (used on the Acorn BBC Microcomputer, the Acorn Electron and certain CP/M machines), the easiest route for 'porting' existing software will be through this language. Alternatively, machine programs written using 6502 assembler may be modified to run on the 65SC816, possibly taking advantage of its extra features. This chapter concentrates on BASIC applications.

### **2.3.1 Extensions to BBC BASIC**

As mentioned above, the BASIC used is a dialect (ie extension of) BBC BASIC. All the well-known features of this language, eg the assembler, procedures and functions, IF...THEN...ELSE are included. In addition, various extensions are provided, eg ON...PROC are available. Below is a description of the differences between this BASIC (also known as Communicator BASIC) and existing versions of BBC BASIC (BASIC I, II and III).

### **2.3.2 Enhancement to the ON ... statement**

It is now possible to call procedures in an ON statement, as well as subroutines. For example:

```
120 ON int$-129 PROCleft, PROCright, PROCdown, PROCup ELSE PROCerr
```

The PROC calls have exactly the same syntax as normal procedure calls, ie may have parameters separated by commas between brackets:

```
1240 ON menu$ PROCadd(name$,age$), PROCdel(name$), PROCsshow(name$)
```

### **2.3.3 The LIST IF construction.**

The LIST command has been extended to provide a 'cross-reference' facility. The standard LIST command (eg LIST or LIST 100,2000) may be followed by the keyword IF, which is in turn followed by a string of characters. Only program lines containing these characters will be listed. The characters following the IF are tokenised as usual, so keywords may be found. Examples are:

LIST IF IF	list IF statements
LIST IF TIME	list lines with TIME as a function
LIST IF name\$(	list lines using the array name\$()
LIST 100, 200 IF var-	list assignments to var in lines 100-200

### **2.3.4 The TIME\$ pseudo-variable**

This is related to TIME, and provides access to the system's real-time battery-backed clock/calendar. It may be used as a function, eg PRINT TIME\$, or as a statement, eg TIME\$="....". For the format of the string used by TIME\$, see the entry in the BASIC keyword section.

### **2.3.5 The EXT#= statement**

In BASIC II, the function EXT# is used to return the length of an open sequential file. In Communicator BASIC, you can also assign to EXT#, to change the length of an open file. This facility can only be used with suitable filing systems (eg ADFS, Network, but not card).

### **2.3.6 The COLOR statement**

In Communicator BASIC, the COLOUR statement may be spelt alternatively as COLOR and still be accepted. It will, however, be listed as COLOUR.

### 2.3.7 Extensions to the assembler

The processor used in the machine is the 65SC816, a CMOS version of the 6502 processor with an enhanced instruction set. To cater for this, the assembler built into BASIC has been extended to accept the new instructions. The new mnemonics, addressing and directives modes are described later in this Chapter.

Assembly listings are now formatted in a more readable manner than in earlier BASICs, so that labels of up to nine characters (including the initial .) may be used without disturbing the formatting.

You can use lower case letters throughout the source program if you choose, such as lda &70,x and equs "fred".

The assembler no longer gets confused by the accumulator addressing mode. For example, in previous versions ASL ALFRED would be treated as ASL A followed by the 'comment' LFRED. This no longer happens (the same applies to LSR, ROL, ROR, DEC and INC).

In order to facilitate the writing of programs which use the operating system routines, several special pseudo-variables have been defined. These may only be used as functions (ie their values may not be changed). They hold the addresses of the various OS entry points and reason codes. All the variables start with @ and end with %. The rest of the name is the same as documented in the OS Guide. For example the address MM (for the memory-management routine) is held in the variable @MM%.

### 2.3.8 Improved LISTO formatting

After a LISTO 7 command, LIST indents REPEATs and NEXTs more neatly, so that UNTILs lie under the corresponding REPEATs and NEXTs line up with their FORs.

BASIC also now strips all trailing spaces from input program lines, and if the current LISTO option is non-zero will also strip leading spaces (between the line number and the first non-space character on the line). A side-effect of this is that blank lines such as

```
1000 [RETURN]  
(typing {SPACE} into the program line) cannot be entered if LISTO is non-zero.  
(Use  
1000: [RETURN]  
instead.)
```

### 2.3.9 Other minor changes

SAVE can take any string expression as its argument such as SAVE a\$+b\$, and the indirection operators ! and ? may be used in formal parameters eg DEF FN!red(!&70).

You may now use ASCII code 141 in comments and strings, for example to produce double-height characters in teletext display modes:

```
100 REM<&68D>Big comment  
110 REM<&68D>Big comment
```

(Previously, the RENUMBER and LIST commands could be confused by this code.)

In addition, LIST will now list comments that include teletext colour codes as coloured lines in Teletext modes; you no longer have to include a " just after the REM.

The AUTO command no longer prints a space after the line number.

The VDU command recognises a new punctuation character !. This may be used where , or ; is allowed, and has the effect of sending nine zero characters to the VDU drivers. For example, these two lines are equivalent:

VDU 23!  
VDU 23,0,0,0,0,0,0,0,0,0

### **2.3.10 Running existing BASIC programs**

Most programs which do not call machine code will work without alteration under Communicator BASIC. Those which call machine code programs using CALL or USR will probably have to be changed. More details of these changes are given below.

### **2.3.11 The @HELP procedure**

One of the keys on the keyboard of the machine has the legend HELP on it. The user will press this key when he or she encounters problems with the current program. Different languages interpret such a keypress in different ways. BASIC reacts by automatically calling the procedure @HELP. If PROC@HELP cannot be found in the current program the error

No help available

is generated. It is strongly recommended that all application programs are written (or modified) to incorporate a DEF PROC@HELP.

```
10000 DEF PROC@HELP  
10010 ON help% PROChelp1, PROChelp2, PROChelp3  
10020 ENDPROC
```

It is assumed that help% is a variable which is set to 1, 2 or 3 at various places in the program to tell PROC@HELP roughly where the HELP key was pressed. The three procedures PROChelp1, PROChelp2 and PROChelp3 print some useful information on the screen.

When the ENDPROC of the PROC@HELP procedure is executed, control returns to the statement immediately after where PROC@HELP was called. The HELP key is checked at the end of each statement (ie the end of a line or at a :) and whenever input is taken from the user.

### **2.3.12 BASIC programs that call machine code**

There are two reasons for calling machine code from BASIC. The first is to access a function on the operating system which is not directly supported in BASIC. For example, a program might call the memory management routines in the operating system using the CALL statement. The other reason for calling machine code is speed: the BASIC built-in assembler can be used to write small routines called from BASIC which perform simple tasks much quicker than BASIC.

### **2.3.13 Calling operating system functions**

Where a program calls the OS, changes will have to be made. The addresses and functions of BBC Microcomputer OS routines are very different from those in the present machine's OS. To ease conversion, some 'compatible' routines are provided with similar entry and exit conditions. As long as the hardware supports a function, the compatible entry should work.

Examples of compatible routines are:

@OSB% Compatible OSBYTE routine  
@OSW% Compatible OSWORD routine

As an example, if a BBC BASIC program contained the lines:

```
100 osbyte=4FFF4  
...  
...  
1000 A%-138:X%-0:Y%-ch:CALL osbyte
```

## *Chapter 2*

the converted version might be:

```
100 osbyte=@OSBT  
...  
...  
1000 A%=$138:X%=$0:Y%=$ch:CALL osbyte
```

Note that there is no change at all in the line which actually calls the compatible OSBYTE routine. Similarly, OSWORD calls may be converted using @OSW%. The reason code should be loaded in X%, and two low bytes of the parameter block address in X% and Y%.

An example would be to read the 100Hz timer:

```
100 DIM pbt 4          : REM Five-byte parameter block  
110 A%=$1             : REM Use OSWORD 1  
120 X%=$pbt           : REM LSB of parameter block  
130 Y%=$pbt DIV $100  : REM MSB of parameter block  
140 CALL OSW%          : REM Do the call
```

The USR function also works with the compatible calls. The result is a four-byte number in the form &PYXA, ie is made from the least significant bytes of the the contents of the A, X, Y and P registers on exit from the called routine.

### **2.3.14 Extensions to CALL and USR**

The actions of CALL and USR described above are exactly as found in other (6502) versions of BBC BASIC. In addition, the present BASIC provides more information about the state of the processor when the called routine returned, and allows the new D (direct page) and B (data bank) registers to be initialised on entry.

There are four pseudo-variables which, when added to the address to be called, alter the action of CALL and USR. These are:

```
@USEB% Load B register with the third byte of B% variable  
@USED% Load D register with the two LSBs of D% variable  
@WRDAM% Enter the machine code in 16-bit memory mode  
@WRDXY% Enter the machine code in 16-bit index mode
```

The meanings of the B and D registers are explained in the next Chapter, as are the 16-bit/8-bit modes. By default, the value loaded into B (the data bank register) is the same as the most significant byte of the address of the called routine. The D (direct page register) is the same as that used by BASIC. Since this is equivalent to 'zero page' on the 6502, direct page addresses &70-&8F may be used by direct page instructions.

The values used above may be over-ridden. By adding, for example, @USEB% to the call address, the third byte of the B% variable will be loaded into the B register before the machine code is executed. Examples are:

```
B%=&120000  
CALL routine+@USEB%  
result=USR(routine+@USEB%)
```

Note that the brackets are required in the case of USR, otherwise the address would be taken to be simply 'routine', with the result of the USR being added to @USEB%.

Whenever a machine code routine returns, the values of various registers are stored in some more pseudo-variables. The names of these variables correspond to the registers' names, as below:

@A%	The Accumulator	(first two bytes)
@X%	The X index register	(first two bytes)
@Y%	The Y index register	(first two bytes)
@D%	The Direct page register	(first byte)
@B%	The data Bank register	(third byte)
@P%	The Processor status register	(first byte)
@C%	The C flag	(first bit)

These may be used as an alternative to using USR to obtain results from machine code calls.

### 2.3.15 Calling user machine code programs

Programs which call machine code assembled using the BASIC assembler should work with minor alteration. In particular, any RTS instructions which are executed to return to BASIC should be changed to RTL. This is due to the way in which BASIC executes the machine code.

Parameters may be passed, as usual, in a list after the CALL statement's execution address. A parameter block is set up to reflect these parameters. However, its format and location in memory are different from previous versions of BBC BASIC.

The format of the parameter block is as follows:

```
&00 Count of parameters
&01 Three byte pointer to first parameter
&02
&03
&04 Type of first parameter
&05 Pointer to second parameter
&06 ...
```

The count is in the range zero (no parameters) to 63. The address part of each entry is in the order least significant byte to most significant byte. The MSB will generally be the same as BASIC's data bank register. The types of variables are as follows:

```
&00 One-byte integer, eg ?X
&04 Four-byte integer, eg A%, !&70
&05 Real number, eg A
&80 Indirection string, eg $$&500
&81 String variable, eg A$
```

The pointer in the case of a string variable is the address of the string information block. This is a five-byte block containing the following information:

```
&00 Three-byte pointer to the start of the string
&01
&02
&03 Number of bytes allocated to the variable
&04 Current length of the string
```

Again, the pointer is in the order LSB to MSB. The address of the parameter block is passed on the stack, underneath the return address. The stack-relative addressing modes may be used to access the parameters. The table below shows the displacements of various items with respect to the stack pointer:

```
&06 MSB of parameter block address
&05 Middle byte of parameter block address
&04 LSB of parameter block address
&03 MSB of return address to BASIC
```

## Chapter 2

&02 Middle byte of return address to BASIC  
&01 LSB of return address to BASIC  
&00 Top of stack

A program segment to load the first nine bytes (count plus two parameter entries) of the parameter into a work area is given below:

```
.tabset 5 12 30
.entry
    LDA 6,S
    PHA
    PLB
.loop LDA (4,S),Y
    STA parBlk,Y
    DEY
    BPL loop
.rest
...
...
.parBlk      EQUUS"123456789" \Nine-byte area
```

Note that the data bank register does not have to be loaded from the stack as on entry to CALL routines B is initialised to the program bank of the code. This will be the same as the program bank containing the parameter block. Once the parameter block has been moved to 'parBlk', it may be used exactly as if it was set-up at address &600 under 6502 BASIC.

## 2.4 Creating BASIC Programs

This section describes the various steps to be undertaken in the creation of a BASIC module to run on Communicator.

### 2.4.1 TWIN

The most flexible way to write BASIC programs for the C-512 (or indeed, for the BBC) is to use the TWIN editor. Source code is typed in without any line numbers, but preceded by an AUTO statement for the starting line number. The resulting text file can then be EXECed into BASIC as if typed at the keyboard. Line numbers are never explicitly referenced (except in the AUTO statement). If explicit line references are essential (eg for RESTORE), then labels can be used. GOTO is also occasionally required to handle serious error conditions, but otherwise, GOTO and GOSUB are discouraged, and statements such as:

**IF ERL<500 RUN**

are definitely forbidden!

It is usually convenient to split up large BASIC programs into two or more files, each file containing related procedures and functions which might be worked on at the same time. This ensures that even large programs have room for copious comments and long, meaningful variable names. These features aid readability and can be used freely at this stage, since they will all be squeezed out before the program gets to the BASIC interpreter. This method of development in separate files and linking the program together later, is also useful in that it enables the programmer to use the standard Blue Screen utilities procedures (see Chapter 6). These utilities support the typical application Blue Screen layout, text and graphical screen dumps to a Brother (or compatible) printer, and HELP information. The use of these utilities imposes a certain structure on the user's program.

The use of TWIN to write program source also has the advantage that a certain amount of preprocessing can take place before the program gets into BASIC proper.

#### *Uncomment*

The first stage in this preprocessing is the removal of comments by a TWIN command file invoked by <F1>uncomment. This removes all comments from the text, both BASIC comments starting with REM and OS comments starting with \*|. Lines consisting entirely of asterisks are also removed (although in some early versions of uncomment, multiple lines of asterisks are not fully purged). This allows great flexibility in the copious commenting of programs without increasing the executable size of the final code. It should be noted that Uncomment does not remove OS comments with a space between the asterisk and the vertical bar, and this feature can be used to ensure that the default TWIN filename is changed such that pressing <F4><return> does not accidentally overwrite your original file. See the example source file given below under Summary.

**Demanifest**

The second stage in the preprocessing is the substitution of manifest constants. If a particular number (for example the number of entries in a table, or the number of display lines on a screen) is constant and used often in a program, it can make the program much easier to understand if the constant is given a meaningful name. This can be done by assigning the relevant value to a variable early in the program, but this takes up both codespace and workspace. It is often more useful to be able to define manifest constants which are replaced by the actual numerical value before the program reaches BASIC proper. A simplified method for performing this substitution has been provided for software developers working on BASIC programs using TWIN.

Manifest constant names look like ordinary BASIC integer or string variables, except that the first character is a commercial at (@). The values to be taken by these constants when the program is run are defined in a file in the program development directory, usually called CONSTS. This is written in a user-readable format such as :

Manifest file for program example. Defines > Consts

```
@undefined$      = -1      ; useful value to return  
@screen_width$  = 80      ; change if we use another mode  
@screen_mode$   = 0       ; ditto  
@title$         = "Example V 1.00"
```

It is important that the first line of the file is blank and the title line with the filename is the second line of the file. This is to ensure that the command <F1>Edconvert will remove the filename when converting this constant definition file. The process creates a file in which each line contains an <F5> global exchange to substitute the appropriate constant in place of the manifest. The resulting file should be saved as Demanifest. The demanifest file can now, itself, be used with <F1> in TWIN, to perform the substitution of all the manifests.

### **Summary**

In summary then, a source file should consist of the following elements :

- 1) filename for source file:                \*| removable
- 2) filename for compressed file            \* | non-removable
- 3) comments giving title, author, history etc.
- 4) an AUTO statement to make the program EXECable
- 5) Source code containing comments, manifests, etc.

For example:

```
*| > example_1      main program file for EXAMPLE module
* | > xmpl_1      compressed main program for EXAMPLE

*| Created      1986.12.10 by Andy - Pennine Software
*| Last modified 1986.12.11 by Andy

*| Modification history

*| 1986.12.11 Changes for new OS interfaces - release 1.00

AUTO 1000

PROCdeclarations

REM main program is a tight loop structure :

REPEAT
...
...

UNTIL example_endedit > @example_termination_value

END
```

The source should be loaded into TWIN and the commands

```
<F1>uncomment<return>
<F1>demanifest<return>
<F4><return>
```

entered. This will result in a compressed source file with no comments and no manifests unsubstituted.

## 2.4.2 Code compression

The uncommented and demanifested source code can now be \*EXEC'ed into BASIC, on either a BBC or a Communicator (the latter is essential for very large programs, when a \*BASIC 10000 may be found useful). It is at this stage that the Blue Screen Utilities should be incorporated if required (see Chapter 6). The resulting complete BASIC program can be tested immediately, and it is probably a good idea to do so, since errors are much easier to identify at this stage when the program still has long variable names and is generally easier to read. If preliminary testing indicates that the program is OK, then save it with a name ending in "/B" (B for BASIC).

The code can now be compressed into the minimum possible space using the TCRUNCH utility on a Communicator. Remember that TCRUNCH should be executed under 65TURBO. It should be noted that this imposes a limitation on the size of application that can be developed, since the whole program must be TCRUNCHED in one go (otherwise variable names would no longer match up between one part of a program and another), and TCRUNCH will not accept input greater than 64K in size. There are also one or two restrictions on the source code which are detailed below. The following procedure should be followed:

```
*exec xmpl_1
...
<escape>
*exec xmpl_2
...
<escape>
*exec $.S.BAS_utils.i_util1
...
<escape>
SAVE "EXAMPLE/B"
*65TURBO
*TCRUNCH EXAMPLE/B EXAMPLE/A
```

This gives a compressed file identified by an ending of "/A" (A for After crunching or Abbreviated). This is as far as the process need be taken if the program is to be LOADED and RUN or CHAINED from the BASIC environment on Communicator.

### *Restrictions imposed by TCRUNCH*

#### *EVAL*

Because TCRUNCH changes all variable names used in the program to provide the shortest set of names possible (usually one, or at most two, character identifiers), no part of the code may rely on variable names being unchanged. This means that the EVAL statement, which takes a string argument which TCRUNCH will not change, cannot evaluate expressions involving the programmer's variable or function names. For instance the following cannot be \*TCRUNCHED:

```
plotted_function$ = "FNcomplex_curve("
y_point#=EVAL(plotted_function$+STR$(x_point#)+")")
```

Since the function complex\_curve would have a shortened name wherever it was \*TCRUNCHable, but the name in the string would be unchanged. This would result in a "No such FN/PROC" error. Similar situations with variable names tend to give rise to "No such variable" errors. If the programmer happened to be using very short variable names in such an application, it is possible that the name preserved in the string would match a name generated by TCRUNCH, which could give very obscure results indeed. The moral: avoid both EVAL with variable names and the use of short variable names.

### *Multiple entry points to procedures*

Because BASIC ignores a DEFPROC or DEFFN statement, and all of the line following, if it encounters it during sequential execution of a program, it is possible to provide multiple entry points to a procedure. For example:

```
DEFPROCexplicit (At,Bt,Ct)
...
...
DEFPROCimplicit (At)
something% = At * Bt + Ct
...
...
ENDPROC
```

This would work in an unCRUNCHed BASIC program, with the effect that the explicit call allows the parameters B% and C% to be supplied with the call, while the implicit call would enter the same code a little later, and use the global values of these variables. This can be quite a useful (though definitely 'dirty') technique for saving code space, but will not work without modification after being TCRUNCHed. The reason for this is that TCRUNCH tries to compress as much as possible onto a single line of code, subject to not concatenating IF's and \*commands so as to change the logic. In this case, the line "something% = ..." would be appended to the DEFPROC, so that if executed by entry from PROCexplicit, the whole DEFPROCimplicit line would be ignored including the concatenated code.

If this technique is essential to your code, then the solution is to modify the DEFPROCimplicit line to prevent the following line being concatenated. This can be achieved in two ways:

DEFPROCimplicit (At) :IF	:REM hanging IF
DEFPROCimplicit (At) :*	:REM null OS command

A similar approach is needed if the bad programming practice of using GOTO to jump to a DEFPROC line is used. Though such practices are to be discouraged, the problem may arise in the conversion of existing software.

### 2.4.3 Module Support

A final stage is required if the program is to be made into a module, so that it may be incorporated into a system ROM or loaded and installed as a module in RAM (necessary if it is to appear on the main menu). The MAKEMOD program is used to add a module header to the front of the program, and a checksum to the end.

MAKEMOD is a BASIC program which converts \*crunched BASIC programs into modules for installation on an Acorn Communicator. The program will run on either a Communicator or the ARM second processor under 65TURBO and in both cases is started with the command \*MAKEMOD. This assumes the correct library has been selected with \*LIB, ie. \$LIBRARYC for the C512, and \$LIBRARY for the ARM.

This final stage requires a further file in the source code directory, called build\_data, and a sub-directory called Modules.

*build\_data format :*

```
> build_data for <module name>
<source directory>
<destination directory>
<flags>
<module name>
<file name>
```

any other text - comments etc.

The maximum file size is 512 bytes, but if this causes anyone any trouble it can be increased quite simply (by changing a value in the program in \$S\_TOOLS.MAKEMOD). Note that the module name does not have to be the same as the file name (with or without its /A). Either of the source directory or the destination directory may be null, but not both. It is recommended that the source directory is null (ie. use the current directory) and the destination directory is Modules.

The flags are in the form : (\$MHCPOS% + (\$MHCBNK% + ...  
which is EVALed

The program makes decisions about the contents of the header on the basis of the flags specified and the module name (in the case of Menu and certain other programs designed for use in "Space/Break" systems). There are four cases handled by \*MAKEMOD:

(1) For most programs, designed to run directly below the menu, the following flags will give a result like the old MAKEMODF :

(\$MHCPOS% + (\$MHCBNK% + (\$MHCBA\$% + (\$MHCMEN%

(2) For a Menu module (not necessarily the main Menu program), the module name must be "Menu" (case as shown) and the flags :

(\$MHCPOS% + (\$MHCBNK%

This ensures a result compatible with the old MAKEMODM

(3) For BASIC programs to run deeper in the structure than Menu-visible ones :

(\$MHCPOS% + (\$MHCBNK%

This gives a result compatible with the old MAKEMOD

(4) For service programs that are not to appear on the menu, but which can be invoked by \*<modulename> in a space/break system, whereupon they will take over the machine :

(\$MHCPOS% + (\$MHCBNK%

modulename must be preceded by an asterisk ( eg. \*SERVICE )

*Example*

> build\_data for EXAMPLE

Modules

(\$MHCBNK% + (\$MHCPOS% + (\$MHCLEN%

Example

Example/A

Installable module in \$S.EXAMPLE MODULES.Example/A

The facility to add comments at the end should only be needed if you are doing something particularly obscure, which is not recommended anyway, since adherence to the naming conventions for files and directories makes life much easier for anyone who has to rebuild the system ROMs without benefit of consultation with the original authors.

## **2.5 Using assembly language**

It is less easy to convert existing machine code programs written for 6502-based computers (in particular the BBC Microcomputer) than BASIC ones. This is because many of the specific properties of a particular machine which are hidden from the BASIC are very visible to the assembly language programmer.

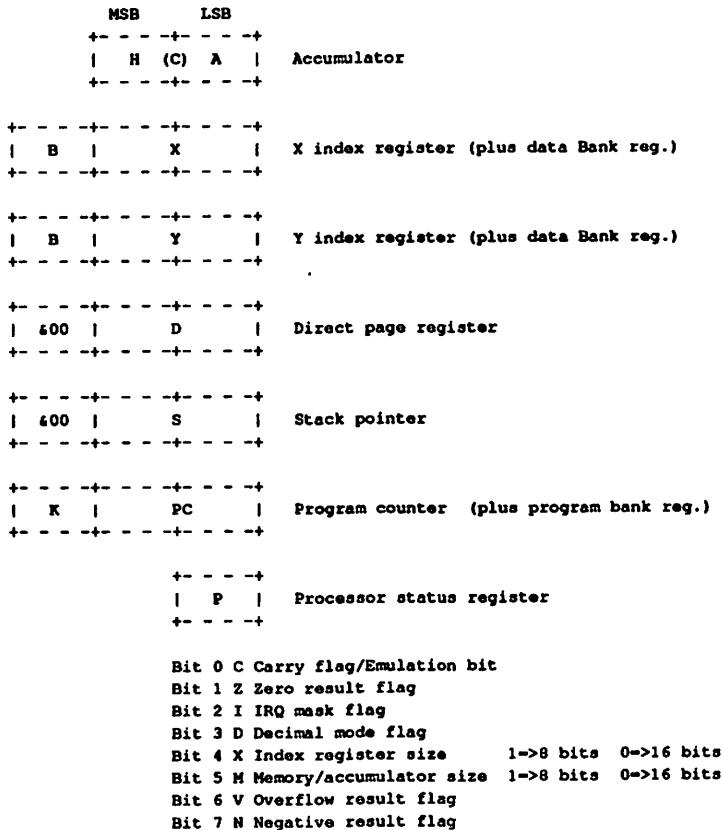
The good news is that programmers writing assembly language applications from scratch are able to use the powerful architecture of the 65SC816 processor. This includes such features as sixteen-bit accumulator and index registers, 24-bit address range (16M bytes) and powerful new addressing modes.

The architecture and instruction set of the 65SC816 is described in the first sections of this Chapter. Notation for both the MASM and BBC BASIC assemblers is given, so this Chapter should be read by programmers intending to use either of these systems.

After the processor description, general hints and tips for programmers used to programming in 6502 assembler are given. Because of the extra features of the 65SC816, certain instructions have slightly different effects from 'identical' ones on the 6502. Additionally, advice on writing bank-independent code is given.

### 2.5.1 Programmer's model of the 65SC816

The architecture of the processor used in the machine is based on the 65C12 processor, which in turn was a (small) development of the 6502. Indeed, the 65SC816 may run in an 'emulation' mode, where it acts exactly as a 65C12, though this facility is not used on the present machine. The diagram below shows the register set of the chip:



### 2.5.2 Register descriptions

**A** - Accumulator. This is the register used for 'data processing' type operations. Its width is eight bits when the M bit of the status register is set, otherwise it is 16 bits wide. The top byte is referred to individually as H (for high). When all 16 bits of the register are used in instructions, it is referred to as C rather than A.

**X,Y** - Index registers. These registers are used for indexing operations. See the description of addressing modes below for more details. They are eight bits wide when the X bit of the status register is set, otherwise they are 16 bits wide.

**B** - Data bank register. The 65SC816 achieves 24-bit addressing by dividing up the address space into 256 banks each of 64K bytes. When absolute-type address mode instructions are executed, the bank number of the operand address is obtained from the B register. For example, the instruction

LDA &1000,X

or

LDAAX &1000

will load the accumulator with the byte at &1000+X in the bank whose number is in B. Other ('long') addressing modes specify the bank to use in the instruction.

**D** - Direct page register. This is a register used in the direct page addressing mode instructions. Direct addressing is analogous to the 6502's zero page addressing, except that the 'zero page' may be anywhere in the first 64K of the memory map. A direct page instruction such as

LDA (&12),Y

or

LDAIY &12

has the value of D added to the operand (&12 in this case) before memory (in bank zero) is accessed. Instructions execute fastest when D is a multiple of 256, ie is on a page boundary.

**S** - Stack pointer. Unlike the 6502, whose stack is confined to address &01XX, the 65SC816 has a sixteen bit stack pointer. Thus the stack may occupy any part of bank zero.

**K** - Program bank register. This holds the bank number of the code which is currently executing. That is, it forms the high order eight bits of the program counter, as implied by the diagram. Jump and branch-type instructions have destinations in the current program bank, unless a 'long' address mode is used, in which case the bank to use is given in the instruction.

**PC** - Program counter. This forms the least significant sixteen bits of the address of the next instruction. When the end of a bank is reached (PC is &FFFF) the program counter 'wraps around' to zero. The program bank register, K, is not incremented, however.

**P** - Processor status register. This contains nine flags. Only eight are visible at once, the carry and emulation flag occupying the same bit. Instructions exist to set and reset explicit bits in the status register (eg SEC, CLV) and also to act on several bits at once (RSP, SEP).

The flags N(egative), Z(ero) are set by most data operations to reflect the result of the destination register. The (o)V(erflow) flag is set by the ADC and SBC instructions, and C(arry) is set by arithmetic, shift and rotate instructions. The BIT instruction also affects N, Z and C.

The X flag determines the size of the index registers X and Y. When X is set, the registers are eight bits wide, and all operations on them affect only the lowest eight bits. The act of setting the X bit clears the top byte of X and Y. When the X bit is clear, the registers are sixteen bits wide.

The M flag determines the width of accumulator and read-modify-write (eg ROR &1234) operations. When M is set, operations are single-byte ones. However, the high byte of A is preserved in eight-bit mode, and operations which transfer A to sixteen-bit registers (eg TCS) use both bytes. When M is clear, accumulator and R-M-W operations are two-byte ones.

### 2.5.3 65SC816 addressing modes

There are 24 addressing modes available on the 65SC816. Some are the same as 6502 modes, some enhanced versions of these, and others are totally new. Note that with some addressing modes, when an addition takes place in address calculation (eg when indexing) crossing page and bank boundaries adds a cycle to the instruction, ie slows it down a little. Similarly, in direct page modes, an extra cycle is added if the low byte of D is not zero.

In the headings below are the names of the addressing modes and how they are expressed in BBC BASIC assembler and MASM respectively. For example, the line

Immediate	#op	IM op
-----------	-----	-------

means that an instruction using immediate addressing uses #op in BBC BASIC assembler and IM op in MASM. Note that in MASM the addressing mode suffix must follow the mnemonic without intervening spaces. 'op' is used to stand for the operand, and 'zop' is used where the operand is a direct page one.

Examples of actual instructions are:

LDAIM ">"	LDA# ASC">"	#op
-----------	-------------	-----

Items in curly brackets (eg {L}) are optional, and may be used to 'force' a given addressing mode. Items separated by | are choices, so {A|B} means either A, B or nothing may appear.

In examples, the BBC BASIC version of the instruction is always given first, followed by the MASM version:

Immediate	#op	IM op
-----------	-----	-------

The operand follows the instruction. It is either one or two-bytes long, depending on the register being affected and the M and X flags. For example:

LDA#10	LDAIM 10	M=1 1-byte operand
LDA#10	LDAIM 10	M=0 2-byte operand
CPX#1234	CPXIM 1234	X=1 assembly error
CPY#1234	CPYIM 1234	X=0 2-byte operand

Note that the assembler has to be kept informed of the state of M and X while generating code so that the correct size operand may be generated. Directives are provided to achieve this. See the MASM or BBC BASIC guide for details.

Absolute	op	(B) op
----------	----	--------

The operand is the two-byte address in the current data bank, or program bank for the jmp instruction. For example, if B=&23 then

LDA &2219	LDA &2219	#op
-----------	-----------	-----

will load A with a one (M=1) or two-byte (M=0) value at address &232219.

**Absolute long**

{L} op

{L} op

The operand is the three-byte address following the instruction. For all but the jmp instruction, the three bytes are low address, high address and data bank of the operand, as in

STA &120034      STA &120034

For JMPL, the three bytes are the new values of PC low, PC high and program bank, eg

JMPL &FF0010      JMPL &FF0010

Note that if the bank address is zero and you want to assemble an absolute long, rather than absolute instruction, the suffix 'L' must be added to the mnemonic:

ANDL &4523      ANDL &4523

**Direct**

zop

zop

As noted above, this corresponds to zero page 6502 addressing. The operand is a single byte value which is a displacement from the direct page pointer D. It is added to D to find the address required. So, if D=&1210, the instruction

CMP &AA      CMP &AA

will compare A with the byte (or two) bytes at &1210+&AA=&12BA.

**Accumulator**

A

A

This is really a type of 'implied' addressing, where the operand is implicitly the accumulator (though the assembler still demands an operand so 'A' is used). An example is:

LSR A      LSRA

Either eight or sixteen bits will be affected depending on the state of the M flag.

**Implied**

The operand is implicit in the instruction. For example:

TAX      TAX      ;Transfer A to X  
SWA      SWA      ;Swap high and low bytes of A

## Direct page indirect indexed (zop), Y

IY zop

This is similar to zero page indirect indexed on the 6502. It is often pronounced 'indirect,Y'. Instead of being in zero page, the operand byte is in the direct page indicated by D. First the operand byte is added to D. Then a two-byte address is obtained from this location in bank zero. The contents of Y are added to this address (Y being one or two bytes depending on the X flag) which, when added to the data bank register, gives the location of the final operand. The final operand may be one or two-bytes, depending on the state of M.

As an example, suppose flags X=0 and M=0 (so indexing and data are in sixteen-bit mode). Also assume D=&8900 and Y=&30A0. The instruction

LDA (&23).Y      LDA[Y] &23

will get a two-byte address from &923 and &924. Say this is &2100. Y will be added to this to give the address of the final operand, &XXS1A0, where XX is the current data bank. The sixteen bit number at this address will be loaded into A.

Direct page indirect indexed long L (zop),Y

LIY zop

This is similar to the previous mode. However, the address held in bank zero is a three byte ('long') address which includes the bank number of the operand. Thus B is not used by the instruction (but obviously remains intact for the next instruction). Long mode is indicated by adding 'L', as in

CMPL. (& 20) Y CMPL.Y & 20

## **Direct page indexed indirect (zop,X)**

IX zop

This is similar to the 6502 zero page indexed indirect mode, but uses the direct page instead of zero page. It is often called 'indirect,X'. The one-byte operand is added to D. Then X is added to this sum (as either a one-byte or two-byte value, depending on the X flag). This gives an address in bank zero of a two-byte address. This is the address in the current data bank of the one or two-byte final operand.

For example, suppose D=3200, X=456 and the M and X flags are zero. The instruction

**CMP (&23,X)**      **CMPIX &23**

will add &3200+&23+&456 to give &3679. At this address in bank zero is a two-byte address. This is an address in the current data bank of the final two-byte operand.

**Direct page,X** **zop,X**

**ZX zop**

In this mode, which is analogous to zero page, X on the 6502, the one-byte operand is added to D. Then X is added to this sum to give the address in bank zero of the final one or two-byte operand. An example is

**STZ &3F.X**      **STZZX &3F**

## *Chapter 2*

### **Direct page,Y**

**zop,Y**

**ZY zop**

This is the same as the above mode, except that the Y register is used instead of X. It is equivalent to the 6502 zero page, Y mode. Only the LDX and STX instructions may use this address mode. An example is

**STX &23,Y      STXZY &23**

When running software from a 6502 machine on the 65SC816, beware instructions of the form:

**LDA &70,Y**

Although this looks like a direct page (or zero page) instruction, it is in fact an absolute one (direct page, Y may not be used with LDA or STA). On the 6502 this does not matter as the address referred to is the same. However, quite a different address is being referred to, unless both the D and B registers contain zero. The potential danger is easier to spot in MASM as LDAZY is an illegal instruction, and LDAAY clearly implies absolute addressing.

### **Absolute,X**

**op,X**

**{B|A}X op**

This mode takes a two-byte address. The contents of X is added to this to obtain the address of the final one or two-byte operand. The operand lies in the data bank given by the B register (or the one after it if operand+X is greater than &FFFF). Example:

**EOR &1232,X      EORAX &1232**

### **Absolute long,X**

**{L} op,X**

**{L}AX op**

This is related to the above mode, but the address following the opcode is a three byte one containing the data bank to use. As usual, 'L' appended to the instruction indicates explicitly a long address mode, as in:

**ORAL &1236,X      ORALAX &1236**

### **Absolute,Y**

**op,Y**

**AY op**

This is similar to absolute,X except that Y is used as the index register.

### **Relative**

**op**

**op**

This address mode is the same as the one on the 6502. It is only used by branch instructions. The one-byte (signed) operand is added to the current value of the program counter to give the destination address. Note that programs may not branch over bank boundaries.

### **Relative long**

**PER/BRL op**

**PER/BRL op**

This is similar to the previous mode, but has a two-byte operand. This gives a displacement in the range -32768 to +32767 from the instruction after the branch (-32765 to +32770 bytes from the branch itself). Again, long branches cannot cross a bank boundary. The only instructions to use this mode are BRL (branch always long) and PER (push effective relative address).

**Absolute indirect****JMP (op)****JMI op**

This addressing mode is used only by jump instructions. It takes a two-byte operand and uses this as an address in bank zero. At this address is the final two-byte destination address in the current program bank. For

**JMP (&7612)      JMI &7612**

will take a two-byte value at locations &007612,&007613 and load this into PC low and PC high. K remains the same.

Unfortunately, because this address mode uses bank zero, it is impossible to allocate a fixed location to be used for the indirect address (for example, several processes might all be using the location 'at once'). It is therefore strongly recommended that indirect jumps are avoided in new programs, and removed from existing ones.

The absolute indexed indirect addressing mode has an operand in the current program bank, as opposed to bank zero, so this may be used instead of the current addressing mode. See the description of indexed indirect addressing for details.

The alternative to using indirect addressing is to push the address on to the stack and perform an RTI. For example to jump to and address held in the BHA registers (MSB in B):

PHB	PHB	:Push the MSB
WRD &20	RMODE &20	:Ensure sixteen bit memory mode
RSP #&20	RSPIM &20	
PHA	PHA	
PHP	PHP	:Push a dummy P register for the RTI
RTI	RTI	

Note that the address is pushed high byte first, and that a dummy PHP is pushed as one is pulled by the RTI. Note also that this code has nothing to do with interrupts; it is just a way of getting the desired address into the program counter.

**Absolute indirect long****JMPL (op)****JML op**

This is a 'long' version of the previous mode. It takes three bytes from the bank zero locations, and therefore loads the program bank register in addition to the PC. An example is:

**JMPL (&3229)      JML &3229**

See the comments above about the limitations of this mode.

**Direct page indirect****(zop)****I zop**

This corresponds to zero page indirect on the 65SC12. The single byte operand is added to the D register. The sum gives the address in bank zero at which the address of the final operand is found. The final operand is in the current data bank. An example is

**CMP (&70)      CMPI &70**

**Direct page indirect long**

**L (zop)**

**LI zop**

This relates to the previous mode. However, the address stored in bank zero is a three byte one which gives address low, address high and data bank for the final operand. Note that the instruction size is the same as direct page indirect; only the size of the final operand differs. An example is:

**SBCL (&AF)**

**SBCLI &AF**

**Absolute indexed indirect**

**(op,X)**

**IX op>**

This addressing mode is only used by the JMP and JSR instructions. The two-byte operand gives an address in the current program bank. X is added to this to give an address in the program bank which holds the destination address. PC low and PC high are loaded with this address. For example, suppose X=10. Then

**JMP (&3465,X) JMIX &3465**

would load PC low with the contents of &3475 and PC high with the contents of &3476 in the current program bank.

**Stack**

**op**

**op**

This address mode covers several types of instruction. Examples are the push and pull operations, eg PHP, PLA, jumping to and returning from subroutines (JSR, RTS) and pushing effective addresses (PEA).

**Stack relative**

**op,S**

**S op**

This mode is used with the 'group one' instructions, ORA, AND, EOR, ADC, STA, LDA, CMP and SBC. The one-byte operand is added to the stack pointer. The final one or two-byte operand is found at the address given by this sum.

**Stack relative indirect**

**(op,S),Y**

**SIY op**

This is also restricted to group one instructions. The one-byte operand is added to the stack pointer. At this address is a two-byte pointer. The contents of Y are added to this to give the address in the current data bank (or the one after) of the final one or two-byte operand.

**Block move**

op,op

op,op

This is a very specialised addressing mode, used only by two instructions. These are MVP (block move forwards) and MVN (block move backwards). The instructions copy a block of bytes from one address to another. MVP increments the pointers after each move and MVN decrements them.

Before the instruction is executed, the registers C (ie A and H), X and Y have to be set up with: the byte count minus 1, the source address and the destination address respectively. The two bytes following the instruction give the source bank and destination data bank respectively. After the instruction has finished, the data bank register B is loaded with the destination bank value.

The example code below moves &1000 bytes from address &128452 to &297623, assuming 16-bit M and X modes.

PHB	PHB	;Save data bank register
LDA #&1000-1	LDAIM &1000-1	;Byte count
LDX #&8452	LDXIM &8452	;Source address
LDY #&7623	LDYIM &7623	;Destination address
MVP &12,&29	MVP &12,&29	;Move instruction, with banks
PLB	PLB	;Restore our data bank.

Note: the order of the operands in the MVN and MVP instructions is the opposite to that specified in the Western Design Centre/GTE documentation for the 65SC816. The order used by the assemblers provided with the system is consistent with the TAX-type instructions: the first item (&12) is the source, the second (&29) the destination.

#### **2.5.4 The 65SC816 instruction set**

There are 93 mnemonics (instruction types) available and 24 addressing modes. This gives a total 2232 instructions. As there are only 256 possible eight-bit opcodes it is clear that not all instructions work with all addressing modes. This section describes the instructions available and their effects.

A list of the addressing modes which may be used with each instruction is given after the description in terms of MASM mnemonics. Where the mnemonic for an instruction differs between MASM and the BASIC assembler, a note is given about the BASIC form.

### **ADC - add memory to A with carry.**

This performs the operation  $A=A+operand+C$ . The add may be an eight-bit or sixteen-bit one, depending on the M flag. The C, N, V and Z flags are set to reflect the result.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

### **AND - logical AND accumulator with memory**

This performs the operation  $A=A \text{ AND } \text{operand}$ . The operation may be eight or sixteen bits. The N and Z flags reflect the result.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

### **ASL - Arithmetic shift left**

This shifts the operand (memory or accumulator) one bit to the left. Eight or sixteen bits may be shifted depending on the M flag. Zero is shifted into the least significant bit of the operand and the most significant bit of the operand is shifted into C. The N and Z flags are also affected by the instruction.

Modes: Accumulator; direct page; direct page,X; absolute; absolute,X

## *Chapter 2*

### **BCC - Branch if C clear**

This adds a one-byte displacement to the program counter if C=0 (ie relative addressing is used). The displacement is signed and gives an offset of between -128 and +127 bytes from the instruction after the branch (ie between -126 and +129 bytes from the opcode byte of the branch instruction). No flags are affected. Branches may not cross bank boundaries. If C is not 0 the next instruction is executed.

Modes: Relative

### **BCS - Branch if C set**

This branches if C=1. In other respects it is identical to BCC.

Modes: Relative

### **BEQ - Branch if Z set**

This branches if Z=1. In other respects it is identical to BCC.

Modes: Relative

### **BGE - See BCS**

### **BIT - Bit test**

This performs a one or two-byte AND operation between the operand and the accumulator, without affecting either of them. The Z flag is set to reflect the result of the AND. In addressing modes other than immediate, the N flag is set to bit 7 (or bit 15 if M=0) of the operand and the V flag is set to bit 6 (bit 14 if M=0). In immediate mode these flags are unaffected.

Modes: Immediate; direct page; direct page,X; absolute; absolute,X

### **BLT - See BCC**

### **BMI - Branch if N set**

This branches if N=1. In other respects it is identical to BCC.

Modes: relative

### **BNE - Branch if Z clear**

This branches if Z=0. In other respects it is identical to BCC.

Modes: Relative

### **BPL - Branch if M clear**

This branches if M=0. In other respects it is identical to BCC.

Modes: Relative

### **BRA - Branch always**

This performs a branch irrespective of the settings of the flag. In other respects it is identical to BCC.

Modes: Relative

### **BRK - Force a break**

This pushes the program counter on to the stack in the order K, PC high, PC low. It then pushes the status register, P. It then loads K with zero, and the PC with the vector stored at address &00FPE6,7. The use of this instruction should be avoided as it clashes with the standard Communicator error handling mechanisms. Its use as shown in the example is a hangover from software developed for the BBC operating system. They are followed by a one-byte error code and a zero-terminated error string. For example:

BRK	BRK
EQUB errNum	= errNum
EQUUS "Bad error"	= "Bad error"
EQUUB 0	= 0

Error number zero is treated as a 'fatal' error, eg will not be trapped by BASIC's ON ERROR statement.

Modes: Implied

### **BRL - Branch always long**

This unconditional branch is one of only two to use the relative long addressing mode (the other being PER). The opcode is followed by a two-byte displacement which gives an offset of between -32765 and +32770 bytes. Bank boundaries may not be crossed by the instruction.

Modes: Relative long

### **BVC - Branch if V clear**

This performs a branch if V=0. In other respects it is identical to the BCC instruction.

Modes: Relative

### **BVS - Branch if V set**

This performs a branch if V=1. In other respects it is identical to the BCC instruction.

Modes: Relative

## **Chapter 2**

### **CLC - Clear the C flag**

This instruction sets the Carry flag to zero. This is the normal operation before, for example, performing an initial ADC. The carry flag is also used by the operating system to reflect the exit status of certain routines. It clears the carry for 'normal' conditions and sets it for 'error' conditions.

Modes: implied

### **CLD - Clear the D flag**

This instruction sets the Decimal flag to zero, ie puts the processor in binary arithmetic mode. The machine's operating system expects D to be zero when called.

Modes: Implied

### **CLI - Clear the I flag mask**

The Interrupt flag in the P register masks IRQs if it is set and enables them if it is clear. Thus the CLI instruction enables IRQs to interrupt the processor.

Modes: Implied

### **CLR - See STZ**

### **CLV - Clear the V flag**

The oVerflow flag is set by certain arithmetic operations to indicate an overflow has occurred. BIT also affects the flag. The CLV instruction resets it to zero.

Modes: Implied

### **CMP - Compare accumulator with memory**

This instruction subtracts its eight or sixteen-bit operand from the accumulator and sets the N, Z and C flags to reflect the result. Neither the operand nor the accumulator is altered. The interpretation of the flags after a CMP is:

N=1	A<operand	(Signed comparison)
N=0	A>=operand	(Signed comparison)
C=1	A>=operand	(Unsigned comparison)
C=0	A<operand	(Unsigned comparison)
Z=1	A=operand	
Z=0	A<>operand	

The N flag is only valid if no overflow occurred, ie  $A-operand > -128$ . If an overflow did occur, the state of N will be reversed. As V is not affected by CMP, SBC must be used to perform rigorous signed comparisons.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

### **COP - Coprocessor instruction**

This has an identical effect to BRK, except that the vector is taken from addresses &00FFE4,5. It is used extensively by Communicator to provide calls to the operating system.

Modes: Implied

### **CPX - Compare X with memory**

This performs an operation similar to CMP except that the operand is subtracted from the X register instead of the accumulator. The size of the operand is determined by the X flag - one byte if it is set, two if it is clear.

Modes: Immediate; direct page; absolute

### **CPY - Compare Y with memory**

This performs an operation similar to CMP except that the operand is subtracted from the Y register instead of the accumulator. The size of the operand is determined by the X flag - one byte if it is set, two if it is clear.

Modes: Immediate; direct page; absolute

## **DEA - See DEC**

### **DEC - Decrement accumulator or memory**

This instruction decrements by one the value held in the operand byte(s). The N and Z flags reflect the state of the operand after the decrement. Note that a synonym for DEC A is DEA.

Modes: Accumulator; direct page; direct page,X; absolute; absolute,X

### **DEX - Decrement X**

This decrements the X index register. One or two bytes are affected depending on the value of the X flag.

Modes: Implied

### **DEY - Decrement Y**

This decrements the Y index register. One or two bytes are affected depending on the value of the X flag.

Modes: Implied

### **EOR - Exclusive-OR memory and accumulator**

This instruction performs the operation A=A EOR operand. One or two bytes are affected depending on the M flag setting. The N and Z flags reflect the result.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

## **INA - See INC**

### **INC - Increment accumulator or memory**

This instruction increments by one the value held in the operand byte(s). The N and Z flags reflect the state of the operand after the increment. Note that a synonym for INC A is INA.

Modes: Accumulator; direct page; direct page,X; absolute; absolute,X

### **INX - Increment X**

This increments the X index register. One or two bytes are affected depending on the value of the X flag.

Modes: Implied

### **INY - Increment Y**

This increments the Y index register. One or two bytes are affected depending on the value of the X flag.

Modes: Implied

### **JMI - Jump indirect**

This jumps to a location within the current program bank whose address is stored in two bank-zero locations. The bank-zero address is given by the two-byte operand of the instruction.

Modes: Bank zero indirect

Note: This instruction is written JMP (addr) in the BASIC assembler.

### **<JMIX - Jump indirect,X>**

This jumps to a location within the current program bank whose address is stored in two bank-zero locations. The bank-zero address is given by the two-byte operand of the instruction plus the X register.

Modes: Bank zero indirect,X

Note: This instruction is written JMP (addr,X) in the BASIC assembler.

### **JMPL - Jump long**

This instruction jumps to a long address which follows the opcode immediately. The three bytes of the operand are PC low, PC high and program bank. Use of this instruction is discouraged because it creates position dependent code.

Modes: Absolute long

### **JML - Jump indirect long**

This instruction jumps to a long address stored in three bank-zero locations. The bank-zero address is given by the two-byte operand of the instruction. Use of this instruction is discouraged because it creates position dependent code.

Modes: Bank zero indirect long

Note: This instruction is written JMPL (addr) in the BASIC assembler.

### **JSL - Jump to subroutine long**

This jumps to a subroutine using an absolute three-byte operand. The current value of the program counter is pushed on to the stack in the order program bank, PC high, PC low. Then the program counter is loaded with the three bytes following the opcode in the order PC low, PC high, program bank. Use of this instruction is discouraged because it creates position dependent code, except when accessing Communicator operating system.

Modes: Absolute long

Note: This instruction is written JSRL addr in the BASIC assembler.

### **JSR - Jump to subroutine**

This jumps to a subroutine. PC high, PC low is pushed on to the stack before the new two-byte address is loaded into PC low and PC high. Use of this instruction is discouraged because it creates position dependent code, except when accessing Communicator operating system.

Modes: Absolute, absolute indexed indirect (JSRAX)

Note: MASM's JSRAX addr is written JSR (addr,X) in the BASIC assembler.

### **LDA - Load the accumulator from memory**

This instruction loads a one or two-byte operand from memory into the accumulator, the size being determined by the M flag. The N and Z flags are altered to reflect the value of the operand.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

### **LDX - Load the X index register from memory**

This instruction loads a one or two-byte operand from memory into the X register, the size being determined by the X flag. The N and Z flags are altered to reflect the value of the operand.

Modes: Immediate; direct page; direct page,Y; absolute; absolute,X

### **LDY - Load the Y index register from memory**

This instruction loads a one or two-byte operand from memory into the Y register, the size being determined by the X flag. The N and Z flags are altered to reflect the value of the operand.

Modes: Immediate; direct page; direct page,X; absolute; absolute,X

### **LSR - Logical shift right memory or accumulator**

This causes the operand to be shifted one bit to the right. Zero is placed in the MS bit of the operand and the LS bit of the operand is transferred to the C flag. The N and Z flags are set to reflect the new value of the operand. The M flag determined whether one or two bytes are shifted.

Modes: Accumulator; direct page; direct page,X; absolute; absolute,X

### **MVN - Move block negative**

This instruction was described in the section on the block move addressing mode above. Before it is executed, HA is loaded with the number of bytes minus one to be moved. X points to the source block and Y points to the destination block. The two bytes after the opcode give the source and destination banks respectively. After each byte is transferred, X and Y are incremented and HA (the count) is decremented. The data bank is set to the destination data bank after the instruction has terminated. The flags are unaffected.

Note: the areas affected by the MVN/MVP instructions cannot cross bank boundaries. If an attempt is made to do so, the pointer 'wraps' around within the same bank.

Modes: Block copy (MVN sourceBank,destBank)

### **MVP - Move block positive**

This has the same effect as MVN except that after each transfer, X, Y and HA are all decremented.

Modes: Block copy (MVP sourceBank,destBank)

### **NOP - No operation**

This single byte instruction has no effect whatever except to use two processor cycles.

Modes: Implied

### **ORA - Logical OR memory and accumulator**

This performs the operation A=A OR operand. The operand may be one or two bytes long, depending on the state of the M flag. The N and Z flags reflect the state of the accumulator after the instruction has executed.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

### **PEA - Push effective address**

This instruction pushes the two-byte quantity following the instruction on to the stack. This number may be interpreted as either an immediate number or an absolute address, depending on how it is used by subsequent instructions. The flags are not affected by the instruction. The stack pointer is decremented by two.

Modes: Immediate

### **PEI - Push effective address direct page indirect**

This instruction also pushes a two-byte address on to the stack. The address is taken from bank zero. The operand of the instruction is a single-byte offset added to the direct page pointer D to give the address of the word to be pushed. The stack pointer is decremented by two.

Modes: Direct page indirect

### **PER - Push effective address relative long**

This is followed by a two-byte program counter-relative displacement. The displacement is added to the address of the instruction following the PER to obtain the value to be pushed. The flags are unaffected. The stack pointer is decremented by two.

The assemblers act such that if a label is used as an operand, the address pushed when the PER is executed will be that of the operand. For example

```
PER      lab1  
        ...  
lab1    ;
```

The address of lab1 will be pushed when the PER instruction is executed. Because this is derived from a relative value, the instruction is position independent.

Modes: Relative long

### **PHA - Push the accumulator**

This instruction pushes the accumulator on to the stack. One byte is pushed if M=1 (just A) and two bytes if M=0 (H then A). The flags are unaffected. The stack pointer is decremented by one or two.

Modes: Implied

### **PHB - Push the data bank register**

This pushes the B register (one byte) on to the stack, decrementing the stack pointer by one. The flags are unaffected.

Modes: Implied

## *Chapter 2*

### **PHD - Push direct page register**

This pushes the D register (two bytes) on to the stack, decrementing the stack pointer by two. The flags are unaffected.

Modes: Implied

### **PHK - Push the program bank register**

This pushes the K register (one byte) on to the stack, decrementing the stack pointer by one. The flags are unaffected.

Modes: Implied

### **PHP - Push the processor status register**

This instruction pushes the P register (one byte) on to the stack, decrementing the stack pointer by one. The flags are unaffected.

Modes: Implied

### **PHX - Push the X index register**

This pushes the X register (one or two bytes, depending on the X flag) on to the stack, decrementing the stack pointer by one or two. The flags are unaffected.

Modes: Implied

### **PHY - Push the Y index register**

This pushes the Y register (one or two bytes, depending on the X flag) on to the stack, decrementing the stack pointer by one or two. The flags are unaffected.

Modes: Implied

### **PLA - Pull the accumulator**

This instruction pulls the accumulator from the stack. One byte is pulled if M=1 (just A) and two bytes if M=0 (H then A). The N and Z flags are affected. The stack pointer is incremented by one or two.

Modes: Implied

### **PLB - Pull the data bank register**

This pulls the B register (one byte) from the stack, incrementing the stack pointer by one. The N and Z flags are affected.

Modes: Implied

**PLD - Pull direct page register**

This pulls the D register (two bytes) from the stack, incrementing the stack pointer by two. The N and Z flags are affected.

Modes: Implied

**PLP - Pull the processor status register**

This instruction pulls the P register (one byte) from the stack, incrementing the stack pointer by one. All of the flags are affected.

Modes: Implied

**PLX - Pull the X index register**

This pulls the X register (one or two bytes, depending on the X flag) from the stack, incrementing the stack pointer by one or two. The N and Z flags are affected.

Modes: Implied

**PLY - Pull the Y index register**

This pulls the Y register (one or two bytes, depending on the X flag) from the stack, incrementing the stack pointer by one or two. The N and Z flags are affected.

Modes: Implied

### **ROL - Rotate left memory or accumulator**

This instruction rotates its operand (one or two bytes, according to the M flag) by one bit left. The MS bit of the operand is transferred into the C flag. The previous state of the C flag is transferred into the LS bit of the operand. The N and Z flags reflect the state of the rotated operand.

Modes: Accumulator; direct page; direct page,X; absolute; absolute,X

### **ROR - Rotate right memory or accumulator**

This instruction rotates its operand (one or two bytes, according to the M flag) by one bit right. The LS bit of the operand is transferred into the C flag and the previous state of the C flag is transferred into the MS bit of the operand. The N and Z flags are also affected to reflect the state of the rotated operand.

Modes: Accumulator; direct page; direct page,X; absolute; absolute,X

### **RSP - Reset status bits**

This instruction takes an immediate operand and performs the logical operation P=P AND NOT operand on the status register. The effect is to reset the bits in P which are set in the operand and leave the other bits unaffected. A typical usage is:

**RSPIM &30**

to reset the X and M flags, putting the processor into 16-bit data and index mode.

Modes: Immediate

### **RTI - Return from interrupt**

This instruction may be used to return from COP and BRK instructions, and ABORT, IRQ, NMI and RES interrupts. It pulls the following four bytes from the stack: P (status) register, PC low, PC high, K (program bank) register. All of the flags are affected.

Modes: Implied

### **RTL - Return from subroutine long**

This is used to return from a subroutine which was called using JSR. It pulls three bytes from the stack: PC low, PC high and K (program bank) register. The address of the next instruction is one greater than the address on the stack. No flags are affected.

Modes: Implied

### **RTS - Return from subroutine**

This instruction returns execution from a subroutine which was called using JSR. Two bytes are pulled from the stack: PC low, PC high. The program bank register is unaltered. The flags are unaffected.

Modes: Implied

### **SBC - Subtract from accumulator**

This performs the operation A=A-operand-NOT C. One or two bytes may be affected, depending on the state of the M flag. The C flag should be set before performing an initial subtraction, and will be set after the instruction if no borrow occurred. The N, Z and V flags are also affected to reflect the result.

Modes: immediate; direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

### **SEC - Set C flag**

This instruction sets the Carry bit in the status register. A set carry on exit from operating system routines usually implies that an exception has occurred.

Modes: Implied

### **SED - Set D flag**

This instruction sets the Decimal mode flag in the status register. In this mode, the ADC and SBC instructions treat the accumulator as two (or four) binary coded decimal digits. The D flag should be cleared when calling operating system routines.

Modes: Implied

### **SEI - Set I flag**

This sets the Interrupt disable mask in the status register. While this flag is set, IRQ interrupts will not affect the operation of the processor.

Modes: Implied

### **SEP - Set status bits**

This instruction performs the operation P=P OR operand. The operand is always immediate. The bits which are set in the operand will be set in the status register; the other bits will be unaffected. A typical usage is:

SEPIM &30

to set the X and M flags.

Modes: Implied

### **STA - Store accumulator in memory**

This instruction stores the one or two-byte accumulator into the address given by the operand. No flags are affected.

Modes: direct page; direct page,X; direct page indirect; direct page indirect,X; direct page indirect,Y; indirect long; indirect long,Y; absolute; absolute,X; absolute,Y; absolute long; absolute long,X; stack relative; stack relative indirect,Y

*Chapter 2*

**STP - Stop the clock**

This instruction stops the processor clock, and therefore the program, until a RESET occurs. This instruction should not be used within Communicator code.

Modes: Implied

**STX - Store the X index register memory**

This stores the contents of the X register (one or two bytes) into the address given by the operand. No flags are affected.

Modes: Direct page; direct page,X; absolute

**STY - Store the Y index register memory**

This stores the contents of the Y register (one or two bytes) into the address given by the operand. No flags are affected.

Modes: Direct page; direct page,Y; absolute; absolute,Y

**STZ - Store zero in memory**

This instruction zeroes the one or two bytes addressed by its operand. It has a synonym CLR. No flags are affected.

Modes: Direct page; direct page,X; absolute; absolute,X

**SWA - Swap A and H**

This exchanges the high and low bytes of the C register, regardless of the state of the M flag. The N and Z flags are altered to reflect the state of A (the low half) after the swap.

Modes: Implied

### **TAD - Transfer A to D**

This instruction copies the Accumulator (both bytes, regardless of the state of the M flag) into the Direct page register. The flags N and Z are affected by the operation.

Modes: Implied

### **TAS - Transfer A to S**

This instruction copies the Accumulator (both bytes, regardless of the state of the M flag) into the Stack pointer. The flags unaffected.

Modes: Implied

### **TAX - Transfer A to X**

This transfers the contents of the Accumulator into X. The N and Z flags are set to reflect the contents of X after the transfer.

Modes: Implied

### **TAY - Transfer A to Y**

This transfers the contents of the Accumulator into Y. The N and Z flags are set to reflect the contents of Y after the transfer.

Modes: Implied

### **TCD - See TAD**

### **TCS - See TAS**

### **TDA - Transfer D to A**

This instruction copies the Direct page register into the Accumulator. Both bytes of A are affected. The N and Z flags are affected.

Modes: Implied

## *Chapter 2*

### **TDC - See TDA**

#### **TRB - Test and reset bit**

This instruction performs the operation  $\text{operand} = \text{operand AND NOT A}$ . That is, the bits which are set in A are cleared in the operand, and the bits which are clear in A are unaffected in the operand. In addition, the Z flag is set to reflect the result of operand AND A.

Modes: Direct page; absolute

#### **TSA - Transfer S to A**

This instruction copies the Stack pointer into the two-byte Accumulator. The N and Z flags reflect the state of A and H after the transfer.

Modes: Implied

#### **TSB - Test and set bits**

This instruction performs the operation  $\text{operand} = \text{operand OR A}$ . That is, the bits which are set in A are set in the operand, and the bits which are clear in A are unaffected in the operand. In addition, the Z flag is set to reflect the result of operand AND A.

Modes: Direct page; absolute

#### **TSC - See TSA**

#### **TSX - Transfer S to X**

This copies the Stack pointer into the X index register. The X flag should be clear (sixteen bit index mode) for this operation to be meaningful, otherwise the high byte of S has to be assumed. The N and Z flags are set to reflect the X register after the operation.

Modes: Implied

#### **TXA - Transfer X to A**

This transfers the contents of the X index register to Accumulator. The N and Z flags are altered to reflect the contents of the accumulator.

Modes: Implied

### **TXS - Transfer X to S**

This transfers the contents the X index register to the Stack pointer. If the X flag is set, the high byte of S will be zeroed. The flags are unaffected by the transfer.

Modes: Implied

### **TXY - Transfer X to Y**

This transfers the X index register into the Y index register. The N and Z flags are set to reflect the state of Y after the transfer.

Modes: Implied

### **TYX - Transfer Y to X**

This transfers the Y index register into the X index register. The N and Z flags are set to reflect the state of X after the transfer.

Modes: Implied

### **WAI - Wait for interrupt**

This instruction pauses operation of the processor until an interrupt occurs. When the interrupt routine returns with an RTI instruction, execution resumes at the instruction after the WAI.

Modes: Implied

### **XBA - See SWA**

### **XCE - Exchange carry and emulation bits**

This instruction swaps the C and E bits in the status register. It is used to set or clear emulation mode. The machine always runs in native (non-emulation) mode, so the XCE instruction should not be used.

Modes: Implied

### **2.5.5 General advice for programmers**

This section contains several 'hints and tips' for programmers converting or writing programs in 65SC816 assembly language. Although some of the points have been mentioned previously, they are gathered together here for easy reference.

The examples are given in MASM format only to allow comments to be included where appropriate. Conversion to BBC BASIC assembler is straightforward - see sections 4.2 and 4.3 for details.

### **2.5.6 Bank-independent code**

An important property that programs should exhibit is bank independence. A bank-independent program will run in any of the 256 banks in the memory map without alteration.

In order to be bank-independent a program should not contain any absolute long references to data or program areas. A long reference implies that the position of the data or routine being referenced is known at assembly time. As code may be moved between banks (for example, by changing the socket occupied by a ROM), this requirement cannot be met.

The exception to long references is when the operating system is being called. In this case, long references (eg JSRL/JSL) are always used. This is to enable operating system entry points (which reside in bank zero) to be accessed from any bank.

In MASM the NOLONG directive may be used to fault all long references except those to bank zero. If a program assembles correctly with NOLONG in force, it should be bank-independent (this is not a fool-proof guarantee, however). Below are some further aids to achieving bank independence.

If it is necessary to access hardware addresses, enclose the access between LONG/NOLONG, eg:

**LONG STAL hwaddr NOLONG**

Assemble two images at two different banks, with the address within the bank constant. These images should be identical. A quick test (before a file compare program is run on the files) is to see if they are the same length.

If the BNK directive is used, the argument should only be the bank of the code location counter, ie use should only use:

**BNK :MSB: .**

Note that it is still possible to generate code that is bank dependent and still pass the above tests.

To access tables in the program, use code of this form:

**PHK  
PLB  
LDABX table**

This assumes a

**BNK :MSB: .**

directive (or equivalent) has been executed.

Do not reference :MSB: (an address in the code), eg. do not use:

**LDAIM :MSB:code**

where code is a label in the program, or equivalent.

Do not JSL to routines in your program bank that RTL. Instead use:

**PHK  
JSR codeAddr**

Always access modules outside your own via an internal call to a separate source module (xxxEXT or EXT). This means that changes to external interfaces may be restricted to examining one source file.

### 2.5.7 Position independent code

Writing bank-independent code is relatively straightforward, especially when the NOLONG feature of the MASM assembler is used. Making a program position independent is more difficult, but has several advantages.

A position independent program is one which may be loaded and executed at any address in any bank without alteration. One implication is that all changes of program control (eg jump and subroutine calls) must be relative to the current program counter value. Similarly, data accesses must be either program counter-relative, or refer to the direct page.

It is not envisaged that major applications programs will be written in a position independent way, but small utilities (or 'transients') benefit from the ability to be loaded at any convenient memory location.

Here are some examples of position dependent instructions. They all have the property of requiring re-assembly for each address at which the program is loaded:

LDAAX table	:Table's position varies according to the origin
JMP loop	:Loop's position varies according to the origin
JSR init	:Init's position varies according to the origin

The instructions below are position independent, as the effective address does not depend on the origin of the program:

BNE found	:All branches are relative to the program counter
LDAZX parms	:Direct page accesses depend only on the D register
TAX	:All implicit instructions are position independent
LDAIM ">"	:As are immediate ones
RORA	:And accumulator ones.
JSRL (\$)OSB	:Calls to the OS are position independent

First, position independent changes of program control will be discussed. Conditional and unconditional branches within -32765 and +32770 bytes of the current instruction are available. Any branch instruction may be used in the range -126 to +129 bytes. Examples are:

```
BRA lab
BNE loop
```

For greater displacements, only the unconditional branch is available. To do a long conditional branch, invert the condition and branch over a long branch. For example, to perform:

BNEL test

(where BNEL is an illegal instruction), use:

```
BEQ ntest
BRL test
ntest
...
```

The code sequence above could be put in a MASM macro definition. Below is a list of the pairs of complementary branch instructions:

BNE	BEQ
BPL	BMI
BVC	BVS
BCC/BLT	BCS/BGE

## *Chapter 2*

Up to eight macro definitions would be required (one for each condition). A typical one is shown below:

```
MACRO
$lab BNEL $dest
$lab BEQ #FT01
$lab BRL $dest
01
MEND
```

Note the use of FT is the local label reference. This ensures that the instruction uses the local label 01 in this macro invocation only, not any global label 01 that may have been defined. It also only searched forward for the label.

To perform a relative JSR, the return address (minus one) is pushed using the PER (push effective relative address) instruction and a BRA or BRL used to reach the destination. The sequence may be defined as a macro in the MASM assembler:

```
MACRO
$lab BSR $dest
$lab PER #FT01-1
BRA $dest
01
MEND

MACRO
$lab BSRL $dest
$lab PER #FT01-1
BRL $dest
01
MEND
```

Conditional BSRs may be coded using the 'short branch over' technique described above for emulating long conditional branches.

Accessing absolute data in a position independent manner also requires the use of the PER instruction. This may be used to push the address of the data item on the stack, which may then be accessed through the stack relative indirect addressing mode. For example, to copy ten bytes of data from 'table' into the direct page:

```
PER table
LDYIM 9      ;Ten bytes to copy
copylp LDASIY 1 ;Address is at stack + 1
            ;Can't STAZY
TAX          ;Save it in direct page
STXZY buffer ;Next byte
DEY          ;Remove address
BPL copylp
PLA
PLA
...
```

If many accesses are to be made to a data block, it might be more efficient to transfer the pointer into direct page, and use direct page indirect addressing instead of stack relative indirect. As a rough guide, LDASIY takes eight cycles, whereas LDAIY takes six (or seven if the direct page is not on a page boundary).

A task which requires both position independent program control and position independent data access is performing a jump through a table of addresses. To illustrate the technique, the usual position dependent method is shown below. On entry, A contains an index between 0 and 127 of the entry to be used.

disptch	ASLA	;Scale A to word
	TAX	;Get index to table in X
	JMIX jtab	;Jump indirect through jump table
	...	
jtab	& rout1	;Table of routine entry points
	& rout2	
	...	

There are two stages in making this code position independent. First, the addresses in the jump table must be made relative to some label in the code, so that they may be added to this to find the actual address. Second, access to the addresses in the table must be made position independent. The new version is shown below (assuming eight-bit mode):

disptch	ASLA	;Scale A to word
	TAY	;Get index to table in Y
	PER jtab	;Get table addr and relative base
	CLC	;Add table entry to base
	LDAS 1	;Low byte of relative base
	ADCSIY 1	;Plus low byte of displacement
	SWA	;Save low byte of final address
	LDAS 2	;Now high byte
	INY	
	ADCSIY 1	
	STAS 2	
	SWA	;Restore low byte of final address
	STAS 1	
	RTS	;Jump to the routine
	...	
jtab	& rout1-jtab-1	
	& rout2-jtab-1	
	...	

To save on stack manipulation, the relative base is the same as the jump table start address. The reason for the '-1' in the table entries is that a RTS is used to perform the jump, so the required address minus one must be on the stack. A typical call to the dispatch routine, using the BSR macro defined above, would be:

LDA reason	;Get reason code from direct page
BSR disptch	
...	

### **2.5.8 Jumping to OS entry points**

A common code-saving technique in 6502 programs is to replace the code sequence:

JSR routine  
RTS

with the single instruction

JMP routine

The RTS at the end of 'routine' will act as the RTS to return to the calling program. This technique may be adopted on the 65SC816, except when 'routine' is an operating system entry. All OS routines return with an RTL instruction. This pull three bytes from the stack; one more than the return address put there by the calling JSR. Thus, the form below must be used instead:

JSL osRoutine  
RTS

If, in a existing program, there are many instructions of the form

JMP osRoutine

these may be changed to:

JMP myOsRoutine

and myOsRoutine defined as

myOsRoutine      JSL osRoutine  
                      RTS

### **Indirect jumps**

The instructions:

JMI addr          JMP (addr)  
JML addr          JMPL (addr)

perform indirect jumps through addresses in bank zero. Due to the transient nature of bank zero locations, it is not possible to define fixed bank zero locations which may be used. Use of these instruction should therefore be avoided.

However, the instructions:

JMIX addr          JMP (addr,X)  
JSRIX addr          JSR (addr,X)

use indirect vectors in the current program bank, and may therefore be used with impunity to perform indirect jumps.

### The LDA zop,Y instruction

There are two instructions which, in the BBC BASIC assembler, may look like direct page ones, but are actually absolute. These are instructions of the form:

```
LDA &70,Y  
STA &70,Y
```

There is no direct,Y addressing mode, so these actually assemble as three-byte instructions. This has no ill effect on the 6502, as the 'direct' page starts at address &0000 and there is no data bank register. However, consider the two interpretations of

```
LDA &70,Y
```

on the 65SC816. If this were a direct,Y instruction, the effective address would be D+&70+Y, where D is the direct page register. This is in bank zero. However, it is actually an absolute,Y instruction, and the effective address is B+&0070+Y, where B is the data bank - a very different address. The programmer should therefore ensure that when she writes instructions of the form LDA zop,Y she realises that an absolute, not direct page, instruction will be generated.

This problem does not occur in MASM as the addressing mode is stated explicitly, as in:

```
LDAAY &70  
STAAY &70
```

It is unlikely that mis-interpretation will occur here.

### Direct page indirect instructions

Many data-access instructions operate through pointer in the direct page. The pointer is to an address in the current data bank. This may cause problems when converting programs from the 6502.

To illustrate this problem, suppose a routine is entered with the address of a parameter block is X and Y (low byte, high byte). The routine might load the first two bytes of the parameter block using the following (6502) code:

getblk	STX vec	.getblk	STX vec
	STY vec+1		STY vec+1
	LDYIM 1		LDY# 1
	LDAIY vec		LDA (vec),Y
	SWA		SWA
	LDAI vec		LDA (vec)
	...		...

To pass the address of a parameter block which lies at address &1234 in the current data bank, this code might be used:

LDXIM &12	LDX# &12
LDYIM &34	LDY# &34
JSR getblk	JSR getblk

This would have the desired effect when executed on the 65SC816. Suppose, though, that the parameter block lay in the 6502's zero page, and was set-up using zero page-type instructions. This would translate into being in the direct page on the 65SC816. The code to pass an address of, say, &70 to 'getblk' would be:

LDXIM &70	LDX# &70
LDYIM &0	LDY# 0
JSR getblk	JSR getblk

The effect of this code now differs between the two processors. In particular, the 6502 takes the address as &0070 and the 65SC816 takes it as B+&0070, where B is the data bank. What was required on the 65SC816 was D+&0070, where D is the direct page.

The simple solution to this is to move the parameter block from the direct page to the program's data bank. The more tricky solution is to make sure that the correct locations are addressed. This involves setting the data bank to zero when the routine is called, and calculating the address in bank zero of the parameter block:

TDA	:Get direct page in A
CLC	:Add the par blk address
ADCIM &70	
TAX	:Low byte in X
SWA	:High byte in Y
TAY	
PHB	:Save the data bank
LDAIM 0	:Zero the data bank
PHA	
PLB	
JSR getblk	:Do the routine
PLB	:Restore the data bank

### Updating IO locations

Most of the IO devices in the machine are mapped into memory locations, and may be read from or written to using appropriate 65SC816 instructions. Some of these are also used by system interrupt routines. An example is the data port B of the 6522 VIA. This port is accessed at location &420000.

If you access it, eg to control the RS423 RTS line, you should disable interrupts around the update thus:

```
VIA    * &420000
DATAB  * 0
BIT0   * 1
BIT1   * 2
BIT2   * 4
BIT3   * 8
...
PHP      ;Preserve the interrupt state
SEPIM ($I+($)M ;Disable interrupts & 8 bit mode
LDAL VIA+DATAB ;Get the current contents
ORAIM BIT3      ;Set RTS (bit 3) high
STAL VIA+DATAB ;Save it back
PLP      ;Restore interrupt state
...
```

This prevents a service routine's update of the register being undone by the STAL.

Note also that all access to IO registers should be performed in eight bit mode, as many of these locations are 'read sensitive', and undesirable effects may occur if location n+1 is accessed in addition to the desired location n.

## **2.6 Creating Assembler Programs**

Assembler source files are created using the TWIN editor, assembled and linked via the 65816 Macro-Assembler (MASM). The resultant object files are concatenated into a single object image by the program ALOADER. Both MASM and ALOADER run under 65TURBO on the ARM second processor.

### **2.6.1 Creating Assembler Source Files**

When designing larger programs it is sensible to break the source down into small manageable modules which can be worked on separately and linked together by the assembler.

It is good practice to create separate working directories for an individual program. A program's working directory will contain its up to date source, object and image files.

A directory "X" should be created within a programme's working directory for use by MASM.

Assembler source files are input using the TWIN editor.

## 2.6.2 Creating Assembler Object files

Assembler Object files are created from assembler source files. Objects represent the binary equivalent of the source code assembler opcode mnemonics.

MASM the 65816 macro-assembler creates 65816 object-code files from assembly language source files.

### *Linking Object Files*

MASM provides very limited linking facilities so that program source can be sub-divided into separate files. Files are linked via the LNK assembler directive placed at the end of a source file. The LNK directive takes the name of the next source file to be assembled as its argument. This means that the order of assembly is hardwired at the source code level and it is not possible to link in objects assembled from other source files. The linking facility enables address information from one object file in an assembly to be accessed by another object file in the assembly.

In the following example the contents of two assembler source code files are listed. In file "Example1" there is a branch to an instruction labeled in file "Example2". When the assembler is running it will maintain a list of label addresses and on its second pass will place the address of the instruction at label "branch1" found in the object file "X.Example2" into the instruction referencing it in the object file "X.Example1".

```
; > Example1
::::::::::::::::::;;
;
;
;                               Example1
;
;
::::::::::::::::::;;
BRL      branch1           ; Branch to routine in another file.
LNK      Example2          ; Link directive to next file.

;
;
;
; > Example2
::::::::::::::::::;;
;
;
;                               Example2
;
;
::::::::::::::::::;;
branch1                         ; Label jumped to from Example1
BRL      branch1             ; Loop forever.
END                           ; End of assembly directive
                             ; place at the end of the last file
                             ; in a link chain.
```

### *Including Files*

MASM provides a simple facility to include files at assembly time. Included files normally contain things like system definitions and macros. Include files can contain links to other include files. The assembler can be instructed to include files via the assembler directive GET placed in assembler source files. GET takes the name of the file to be included as its argument. Alternatively macro files to be included can be specified from the MASM command line at assembly time.

In the following fragment of code the GET directive is used to include a file of definitions called "IncludeDefs".

```
    GET      IncludeDefs          ; Pull in the file IncludeDefs
branch1
    BRL      branch1
    END
```

**Note:** always use GET at the beginning of the first source file in an assembly. Scattering GETs throughout a program will lead to *unpredictable* behaviour on the part of MASM.

### *Invoking the Assembler*

The 65816 macro-assembler is invoked from the command line by entering "NEWTUR". Remember that MASM runs under 65TURBO. The user is presented with the MASM prompt "Action :". "ASM" is entered from the MASM command line to invoke the assembly action. The user is then prompted for the name of the first source file the assembly and an optional include file. Assembly then takes place, the resultant object files being located in the directory "X".

The following example demonstrates some possible user interaction with MASM. The bold text indicates output from MASM, the underlined text input from the user.

```
newtur
Action : asm
Source file : srcfile
Macro library : $s.getsystem
```

In the above example MASM is invoked and instructed to assemble the source file list starting with "srcfile" including the macro file list starting with "\$s.getsystem".

For more information on the assembler and how to invoke it see The MASM Guide.

### 2.6.3 Creating an Executable Module Image.

#### *The START macro*

When creating an assembler program to run on Communicator, it has to be made to conform to the module format expected by the Communicator MOS. A system macro "START" is provided to simplify the process. Start should be invoked at the begining of the source code.

The START macro takes the following format:

```
START modname,version,flag,code,args,sif,prog
```

Where:

modname	- the name of the module. If this field is blank, the module name and arguments are assumed to follow immediately after the START macro if the code field is valid. Otherwise an error is flagged.
version	- the module's version number. Takes the format xxxx which translates into the format xx.xx. Optional, defaults to 00.00.
flag	- Extended module flags. Optional.
code	- module flags. This field takes flags values specifying module attributes.
args	- Module arguments. Optional. If no module name is specified, this field ignored.
sif	- Offset to special info field. Not used, leave empty.
prog	- start of code. Put label for module's entry point here.

The following fragment of code sets up the header for a module called "card". The start macro is called with no module name, the version number is set to "10.01", flags are not set up, code sets the module's attributes to a bank independent filing system, no module arguments are set up and SIF is left blank. The final argument is the entry point to the module code as an offset from the begining of the module header.

After the start macro the module name and arguments are defined. Note they are null terminated.

```
mycode *      ($)MHCBNK+($)MHCFS

START ,1001,,mycode,,,entry

: _____;

: Command names
=     "CARD/ACCESS/FORMAT/FREE/PROTECT"
=     null

entry
PLD
PHD
```

For more detail on modules and module attributes see the Communicator Systems Manual.

**The Loader**

Once a program has been assembled the object files must be concatenated into a single check-summed module object image file. To facilitate this process the OEM development system is supplied with a generic object loader program called "ALOADER". ALOADER runs under 65TURBO on the ARM second processor.

ALOADER builds an image file from a set of objects specified by a "build\_data" file within a program's working directory.

A build\_data file takes the following format:

```
> build_data
<directory where X.objects can be found - optional>
<object 1>
<object 2>
.....
<object n>
• • • • • • • •
<BANK address to be saved with image - optional>
<final object name>
```

User defined fields are bracketed by "<>". Fixed data is in bold.

A build file consists of the file name followed by a blank line. The build file name is followed by the name of a directory where the object files are to be found. This can be left blank and the directory will default to "X" in the current directory. There then follows a list of object files each on a separate line. The object files should be given in assembly order. The object file list is terminated by a line starting with a star (\*) character. Following the object file list is an optional bank load address for the image file. If left blank the bank load address defaults to that of the first object file. Finally, the last line specifies the name of the image file.

Every entry in the table can be followed by a comment, but make sure they are separated by a space!

Example:

```
> build_data
    We default to the X directory.
    file1    object file 1
    file2    object file 2
    file3    object file 3
    endfile last object file
    *
        We default to the bank of the first object file.
    Outfile image output file.
```

#### **2.6.4 Summary**

To summarise, creation of an executable assembler module goes through the following stages:

- 1) Enter the source files using TWIN. Use the start macro to create the module header.
- 2) Assemble the source files using MASM.
- 3) Create a build\_data file and create an executable module image using ALOADER.

## **2.7 Coding Conventions and Library Support**

This section gives a brief description of basic procedures to be followed when designing applications to be run from the Main Menu top-level coroutine.

It is assumed that the reader is aware of and understands the nature and role of coroutines and task management within the Communicator system. Those requiring more information are referred to the Communicator Systems Manual and the BlueScreen Chapter.

### **2.7.1 BASIC**

All the BASIC Applications written so far for Communicator are so-called "Blue Screen" software. This means that they all use a standard mode 0 screen layout with white text on a blue background, a title line, one large and one smaller window for information, and a row of boxes depicting the purposes of the Fkeys.

While there is no reason why the programmer should not write applications with his own screen design (for example, using the teletext mode or a multi-colour lower resolution mode), there is a lot to be said for maintaining a consistent appearance over a large range of application tasks. To this end, all the screen handling routines, several other useful procedures and a standard "HELP" support system are available for incorporation into applications. These routines are fully documented in Chapter 6, as are the coding conventions imposed on any application using them.

If the programmer wishes to write his own screen utilities and help support, there are a number of conventions to observe, mainly connected with the way BASIC supports entry codes ENHELP and ENKILL and with the way the Menu program handles applications' screens and contexts.

#### ***PROC@HELP conventions***

When the user presses HELP in an application, BASIC will allow the preempt to occur after executing the first statement when the preempt key reaches the head of the keyboard buffer (this is true for all the task keys). This preempt will ripple up in a similar manner through every level of coroutine until the main Menu is reached. In the case of the HELP key, the Menu will then call the BASIC coroutine which was running the application immediately below it, as a subroutine with entry code ENHELP which arrives at BASIC as reason ENHELP. The Menu provides the application with the default screen and context. BASIC's reaction to this is to create a new coroutine which attempts to run PROC@HELP.

If the programmer has created more than one coroutine level below the Menu, only the top level task is recognised by the Menu, and this task needs to handle the ENHELP calling mechanism itself if help is to be provided by child coroutines. The PROC@HELP provided in the Blue Screen Utilities does this, and is a useful example if the programmer wishes to write his own.

Running PROC@HELP is slightly different from running normal programs, in ways which it is important that the application programmer appreciates.

- 1) PROC@HELP has access to all program variables and may create new variables. If the procedure exits abnormally after creating new variables, however, the main program is liable to crash sometime later, since important pointers in the HELP coroutine's zero page are not copied back to the main task. "Exits abnormally" includes executing a BRK instruction (i.e. a BASIC error, ON ERROR does not work in this situation, since the software stack is important), executing a RETURN with no corresponding GOSUB, calling @CWT% or allowing a preempt.
- 2) During the running of PROC@HELP, BASIC does not allow preempts. This is to prevent a "backdoor" exit from the HELP coroutine with effects as above. It is important that PROC@HELP, and any procedures or functions which may be called, do not contain any code which would allow preempts to occur outside BASIC's control. This means that the programmer should not use GET, and should not use INKEY unless he is certain that the head of the keyboard buffer contains a key other than a task key. This can be checked by using the system COP call @OPXKC% (eXamine Keyboard Character). If a task key has been pressed, either flush the buffer using OSCLI"fx 15,1" or unwind the stack and exit through the ENDPROC of PROC@HELP, so that the Menu can process the key.

#### *PROC@KILL conventions*

If the user instructs the Menu to Kill a task, the coroutine is first called with reason COKILL, which reaches the BASIC as reason ENKILL. This is treated in much the same way as ENHELP, except that a procedure PROC@KILL is used. In this case, the Menu does not provide the task with a screen or context, which has important consequences for the programmer (see (3) below). Similar comments as for PROC@HELP apply to programs with child coroutines.

Again, running PROC@KILL is a little different from normal program text:

- 1) In this case an abnormal exit would not appear to matter, since the program will never have the chance to crash later if it has been killed. Unfortunately, an abnormal exit may result in a pending Escape event which prevents the Menu from destroying the coroutine. This would make the task impossible to kill.
- 2) Again, allowing preempts to occur could provide a route to an abnormal exit. This should not be a problem, however, since PROC@KILL should never need to seek keyboard input.
- 3) PROC@KILL is called "invisibly", i.e. it runs with the screen and context of the caller, which will normally be the Menu (although if tasks have child coroutines it may be some other "ancestor"). It is therefore very important that PROC@KILL does nothing to change its context or screen. i.e. avoid:
  - a) All VDU calls. This includes things like CLS, CLG, MODE n etc.
  - b) All OSCLI"fx ..." calls

For a graphic example of why context changes are a bad idea, enter and run the following BASIC program, press STOP to return to Menu, and kill the task.

```
10 REPEAT : U.FALSE
20 DEFPROC@KILL
30 VDU19,0,3I19,II
40 ENDPROC
```

## **2.7.2 Assembler**

Although applications written in assembly language do not have access to the the BASIC BlueScreen utilities, it is sensible for them to handle the HELP and KILL preemption keys in the same way as BASIC programs.

In order for the HELP key to function properly, the code must be designed so as to allow preemption. Preemption is explicitly allowed during calls to one of the three functions

<b>OPPRE</b>	- allow preemption
<b>OPRDC</b>	- read a character from the keyboard
<b>OPRLN</b>	- read a line from the keyboard

When a coroutine calls one of the above functions the MOS checks the head of the keyboard buffer for the presence of a preempt key code. If a preempt key code is found, the MOS suspends the coroutine and returns control to its parent. Control is eventually passed back to the Main Menu which is the top-level coroutine. The Main Menu notes which preempt key was pressed and invokes the appropriate action.

### ***HELP Conventions***

The following subsection discusses the actions taken by the system on receipt of a HELP key preempt event. It is assumed that applications only span one coroutine. The methods to be adopted when an application spans several coroutines have not been properly decided to date.

When an assembler application is executing and the HELP key is pressed, the coroutine is suspended at the first opportunity and control passed back to the Main Menu. The Main Menu detects that the preempt event was triggered by the HELP key and invokes the suspended coroutine as a subroutine passing it the reason code ENHELP.

Although re-invoked as a subroutine of Menu, the coroutine is run with its own context.

The ENHELP reason code vectors the coroutine call to the help routine supplied by the designer. The routine invoked should perform the necessary actions and exit via an RTL instruction as opposed to the RTS instruction normally employed.

Although the coroutine is suspended and the HELP function run as a subroutine of the Menu, execution takes place within the context of the coroutine. This leaves the coroutine open to context corruption. Should the HELP routine change any sensitive data or variables, the coroutine will probably malfunction when it is resumed. It is the responsibility of the designer to ensure that this does not occur.

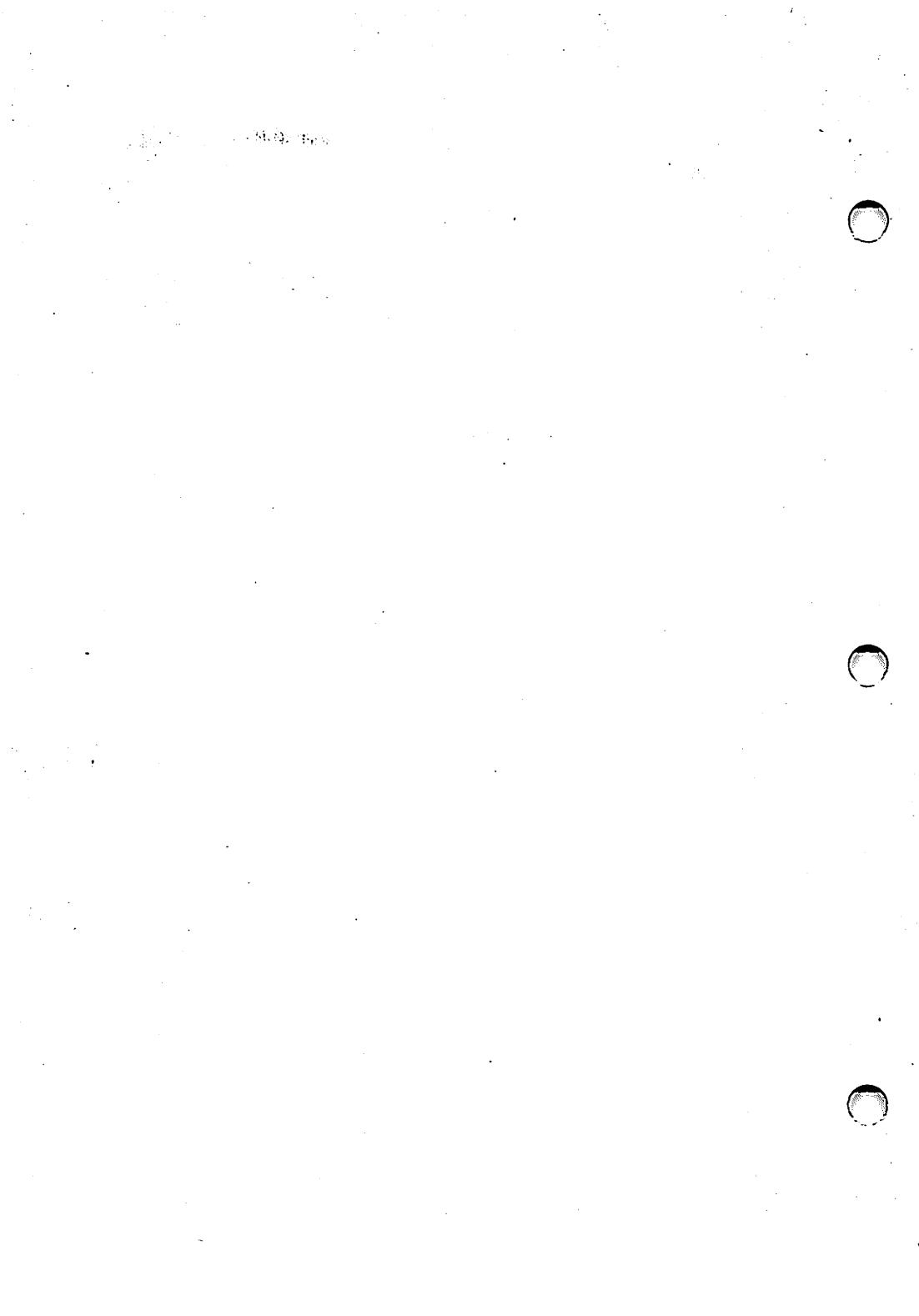
Preempt keys have no effect during execution of the HELP function as it is run as a subroutine of the Main Menu and so is the top-level system coroutine.

### **2.7.3 KILL conventions**

If the user instructs the Menu to kill an assembler task, the assembler coroutine is called as a subroutine with the reason code ENKILL.

The ENKILL function runs with the menu's screen and context but with the coroutine's direct page. The function should release all resources (memory, handles etc) allocated to the coroutine before exiting. The ENKILL function should exit via the RTL instruction as opposed to the RTS normally employed.

The ENKILL subroutine should under no circumstances change the context or screen of the caller.



## 3. TWIN

### 3.1 Introduction

TWIN is a split-screen editor designed specifically for the ACORN RISC MACHINE (ARM). Particular attention has been paid to its ease of use and interaction with assemblers, compilers and high-level languages, all of which can be run directly from TWIN.

Compilation errors, reported by their line number, can be diverted to TWIN files and shown on one of the screen windows. The other screen window can then be used to display the offending line of source, and the programmer can make any required alterations, deletions or insertions in the text. Work on the text may continue while some other task is running concurrently.

#### 3.1.1 Facilities

- Edits text from one of ten internal memory buffers.
- Views portions from two buffers simultaneously.
- Runs background tasks, the results of which can be seen on a screen window.
- Copies text from one window to another.
- Transfers copied text as an input to a background task.
- Goes to specific lines in the text.
- Quickly finds, replaces or counts text specified in a variety of ways.
- Views on-screen help and status information, if required.
- Uses filing system and operating system commands.
- Maintains a software clock and date and time-stamps document files.
- Prints text.

### *Chapter 3*

When TWIN is first loaded, a window is set to the same line and column dimensions as the screen, and it takes its text from buffer 0. The window size can be reduced, enabling the screen to display both static and dynamic help information. Alternatively, a second window which takes its text from a different buffer may be called on to the screen. TWIN can organise as many as 10 memory buffers, of which any two can be viewed by screen windows at any given time.

#### **3.1.2 How this chapter is arranged**

The guide has five main sections. This introduction shows how to enter and leave TWIN and how the screen is arranged. The second section describes the different sorts of file you can edit, and how to use TWIN concurrently with other programs. The third section covers how to edit files, and the fourth describes TWIN's powerful find-and-replace command. The final main section describes how to save, load and combine files.

At the end of the chapter, some useful reference information is included. You'll find section 3.6, which summarises TWIN's function-key commands, particularly useful.

#### **3.1.3 Conventions used in this chapter**

Text printed like this is text that you type or see on the screen:

**colour**

Keys you press are shown like this:

**[Return]**

When you need to press two keys together, we show the instruction like this:

**[Shift]-[F1]**

We use a set of standard names for keys, though they may be labelled differently on your computer. The standard names we use are:

- **[Esc]** for the escape key
- **[Tab]** for the tab key
- **[Ctrl]** for the control key
- **[Shift]** for the shift key
- **[Del]** for the delete or rubout key
- **[Return]** for the carriage return key.

Text printed like this shows the sort of item to type:

*filename*

Here you would type a filename, not the word filename.

{, numeric}

This example shows that you may type a comma then a numeric. The curly brackets indicate optional items in a syntax line.

To give a TWIN command, you press a function key. Each key can give up to four different commands, depending on whether you press it on its own, with **[Shift]**, with **[Ctrl]** or with both the shift and control keys. In this guide, we show these commands as command names in boxes, rather than as particular numbered function keys. For example, pressing function key one is shown as:

**[command]**

Pressing **[Ctrl]** and function key three together is shown as:

**[next line]**

In addition, each function key can have two further meanings, available when you have pressed **[find and replace]** or **[global replace]**. All the function key commands are shown on a standup keycard, supplied with TWIN. We suggest you set it up to the left of your keycard.

### **3.1.4 Starting TWIN**

TWIN is supplied on a 5.25-inch floppy disc in DOS format.

TWIN may be loaded in three ways from the ARM command line:

**TWIN [Return]**

creates a blank document in buffer 0

**TWIN filename [Return]**

loads TWIN and the named file into buffer 0

**TWIN filename filename [Return]**

loads TWIN, the first file into buffer 0, and the second file into buffer 1. Both windows are displayed.

Any file may be loaded into TWIN; it need not be an ASCII file.

### **3.1.5 The screen layout**

When TWIN is entered by typing:

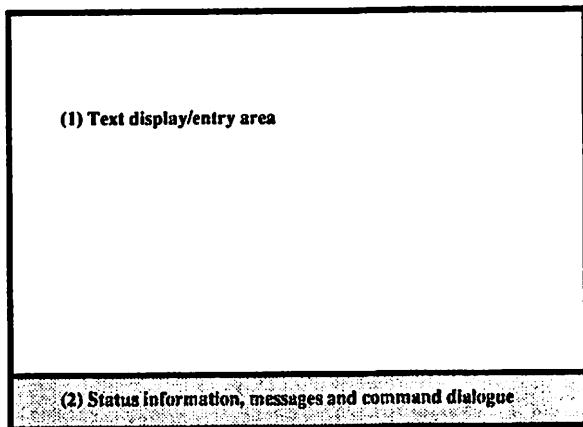
**TWIN [Return]**

or:

**TWIN filename [Return]**

only buffer 0 is active and one screen window is displayed.

The screen format is as shown below:

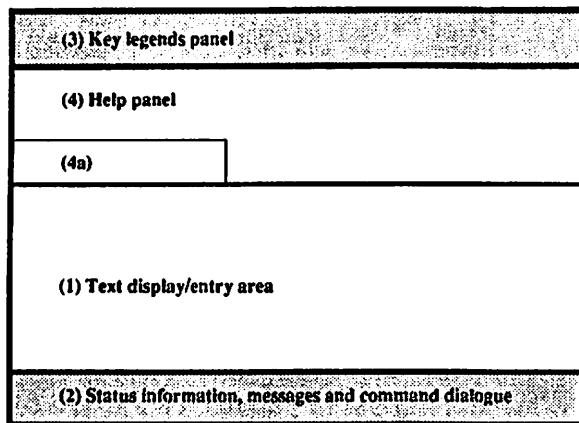


*Chapter 3*

This screen layout is only one of several available. Until you are familiar with TWIN, the descriptive mode screen will probably prove more helpful. This mode is selected by pressing:

**[set mode] D**

There are now four areas on-screen:



### **Text display/entry area**

Displays a portion of the text in the buffer, including the text being typed. This area may contain one or two windows, which need not necessarily be identical in size.

### **Status information, messages and command dialogue**

Usually indicates the current text entry mode and number of marks set, but is also used for information, warning and error messages, command prompts and replies to prompts. The status line has six fields:

<i>example field</i>	<i>explanation</i>
01	left digit: buffer number of current file right digit: number of pushed buffers, or blank
Insert	characters will be inserted into the text
Over	characters will overtype the existing characters
Tab cols	tabulate by eight characters
Tab words	move to under the next word on the above line
One mark	one mark has been set into the text
Two marks	two marks have been set into the text
Original	the current text has not been modified
Modified	the text has been altered since it was loaded or saved
Discarded	the text has been thrown away after replying Y to the safety prompt
LF <-> CR	original text, except that linefeed and carriage-return have been interchanged
"filename"	the name of the current file
date	the date of a dated file
**Command File**	a file which has a load address of 0 and an execution address of FFFFFFFF and which will be read as a series of commands by a filing system
address>	the load and execute addresses of a binary file.

Miscellaneous information, such as the results of global replacements, is also shown in this line.

### **Key legends panel**

Describes the TWIN command provided by each function key.

### **Help panel**

By default this area describes the functions of **Tab**, **COPY** and the cursor keys. In addition to this information, the bottom left-hand corner of this area (marked 4A in the diagram) describes the action of the TWIN commands when they are selected.

### *Increasing the text area*

The text display area may be increased at the expense of the help messages by selecting the key legends (K) mode. This is achieved by:

**set mode K**

The key legends may be removed with:

**set mode 3**

The D and K modes provide helpful information about the editor, but have the disadvantage that they leave less room on the screen for text than the corresponding 3 display mode.

### *How special characters are shown*

Control codes are displayed as their corresponding letter highlighted. For example, **Ctrl A** is shown as an inverse video A.

Linefeed characters are normally hidden from view unless they need to be shown. To display linefeeds, press:

**show new lines**

Linefeeds appear as an inverse video left-arrow rather than a inverse video J. The carriage-return character (CTRL M) cannot be hidden. The end of the buffer marker is displayed as an inverse video asterisk.

### **The top status line**

If you value an on-screen time display, but do not want to use mode D, there is mode T, obtained by pressing:

**set mode T**

This gives a status line similar to the following abbreviated version:

TWIN' T' mode Shift f5 to change mode TWIN at 6001D0000 10:16:46 15-May-86

Amongst other things, it shows the load address of the TWIN in use, and the time and date.

A screen print-out of a full TWIN help screen with two windows in use is shown in Appendix D.

### 3.1.6 The keyboard

The letter, number and punctuation keys have their normal functions within TWIN, but some of the others have modified functions:

<i>key</i>	<i>action</i>
<b>Return</b>	types new-line character (ASCII &0A), moves cursor or completes a command
<b>Tab</b>	moves the cursor (2 modes)
<b>Shift - Tab</b>	switches tab modes
<b>Ctrl - Tab</b>	expands tabs in text
<b>F1</b> to <b>F10</b>	invokes TWIN commands
<b>Shift - F1</b> to <b>F10</b>	invokes TWIN commands
<b>Ctrl - F1</b> to <b>F10</b>	invokes TWIN commands
<b>Ctrl - Shift - F1</b> to <b>F10</b>	invokes TWIN commands <i>key</i>
<b>COPY</b>	deletes the character at the cursor
<b>Shift - COPY</b>	switches cursor to cursor copying mode
<b>Ctrl - COPY</b>	deletes the line marked by the cursor
cursor keys	moves cursor a character/line at a time
<b>Shift</b> with left or right cursor	moves cursor left or right by a word
<b>Shift</b> with up or down cursor	moves cursor up/down by a window-page
<b>Ctrl</b> with left or right cursor	moves cursor to the start or end of line
<b>Ctrl</b> with up or down cursor	moves cursor to beginning or end of text
<b>Ctrl - Shift</b> with up or down cursor	moves both windows up/down by a page
<b>Esc</b>	cancels an incomplete command or leaves cursor editing mode.

The function keys are specially defined by TWIN. Their functions will be explained as and when necessary. A brief description of all of them is given in Appendix A.

### **3.1.7 Leaving TWIN**

To go back to the operating system, press:

**[exit to language]**

**[Return]**

## **3.2 Working with TWIN**

### **3.2.1 Types of file**

TWIN is capable of creating:

- documents which may be any text or language-related source-code. These documents can be date-stamped automatically by TWIN. The current date and time is obtained either by asking you at initialisation time, or by interrogating a network. Both procedures can be incorporated into the early morning start !BOOT file. Should the time not be set, TWIN uses the \*\*Command file\*\* form of document, explained below
- command files can be used in the same way as a document or date-stamped file, but they are marked as having a load address of -1 and an execution address of 0, enabling the operating system to use them as a series of input instructions. Command Files can therefore be executed under the TWIN\* or ARM\* prompt and will perform automatic actions
- pure binary files. The file may be inspected, and it is permissible to edit it by overtyping. Usually, it would only be sensible to edit recognisable ASCII strings. If edited in insert mode, the file would be increased or decreased in length with fatal consequences upon branch instructions. The load and execution addresses of the loaded binary file are shown on the status line.

### 3.2.2 Date stamping TWIN documents

In order to allow advanced tools to run, files can be date stamped. The date is maintained in the software clock on the I/O processor.

TWIN is loaded with the correct time and date on a stand-alone system by using the file DATE. On a system connected to a network, you can type Date -net. The filename DATE expects to be followed by parameters in the form:

DATE DD-MMM-YY HH:MM:SS

For example:

DATE 14-May-87 10:37:08 [Return]

In this example, the system has been dated as 14th May 1987, at a time of 10:37am and 8 seconds.

TWIN starts up with the current date and time shown on the status line. The date of the file will be updated when the file is saved. If the date has not been set, TWIN will initialise a command file. However, a dated file may be loaded into a running version of TWIN which has not had its date set. If an attempt is made to save a dated file from an undated TWIN, a prompt message tells you that the file cannot be marked with the current date and time and will seek permission to save it as a command file.

### 3.2.3 TWIN and the operating system

Any of the filing and operating system commands may be used from the editor, by first pressing:

[command]

The bottom of the screen displays the TWIN\* prompt, and the operating system can be used. There are two options to return to TWIN:

[Esc]

or

[Return]

when the TWIN\* command line is blank.

**Command files**

Although TWIN is normally used directly by the keyboard operator, it can also be driven by command files which have been created for the purpose.

Command files are best used when you wish to repeat the same editing operations on several files or when the editing is such a complex sequence that it is best done in stages.

Since command files execute without your interaction, their construction and testing should be done in stages so that careful checks can be made to ensure that a command file does what it is supposed to do and no more.

The first line of a command file should be blank.

The following example sequence of commands makes a language the background task in a TWIN-window environment.

Enter TWIN in command file mode and type:

[Return] [enter character] [toggle window] [enter character] [push] [enter character] [task status] *language name*  
[Return] [enter character] [toggle window]

This enters six characters (plus the number of characters in the language name) into the TWIN command file. These commands are typed into TWIN using the special [enter character] function.

For example, [enter character] then [toggle window] places the toggle window character into the command file, and so on. The screen will show whatever symbols are associated with these characters: they are likely to be blank, or an assortment of graphic, mathematic and foreign language symbols.

Note the use of [push] to ensure that the language file doesn't overwrite any TWIN file when it is loaded.

Such command files don't always have to run under TWIN, as shown in the example given above. They always start in the operating-system environment, but may call, enter and leave language environments. TWIN will often be involved because the command sequence is likely to want to make use of TWIN's editing features.

This type of command file is often used as a filing system !BOOT file, to select the editor and set options (such as display carriage returns, release scroll margins, select text colour, obtain date and time from a network and so on) at the start of editing.

The text of TWIN can be placed into a language by pressing [exit to language] then the language name. After checking to see if any altered text needs to be saved to the filing system, TWIN will go to the named language. Alternatively, you can go to the operating system by pressing [exit to language] then [Return].

Loading and saving with the filing system is described in section 3.5.

### 3.2.4 TWIN concurrency

To run a task concurrently with TWIN, it is necessary to select a buffer to run it in. Pressing [task status] displays a command prompt on the new window's status line, and a task can be called – it may be a language or a utility. All output from the task prints to the buffer for as long as the task runs, and the contents of the buffer can be viewed in the TWIN window. Editing on another of TWIN's buffers can take place while the task is running, but the task will stop occasionally as TWIN loads and saves files, or gives prompt and error messages which require some immediate response from you. TWIN's clock will not be updated.

When the task's buffer is on-screen, the window is the output screen for the task, and keyboard information (characters typed in the range &00 to &7F) can be passed to the program. All the editing and cut-and-paste facilities of TWIN are retained, and are available to assist in entering commands into the task program. A [copy block] operation can be performed between the TWIN window and the task window which directs all the copied characters to the task. The task window normally shows the information flowing into the end of the task buffer – in this way the latest output is always on view. This is a task-linked situation, the default. However, pressing [auto bottom] unlinks the task so that the task window no longer follows the data streaming into the buffer. The task-unlinked state allows you to move the cursor all over the task buffer.

[auto bottom] toggles between the task-linked and task-unlinked state. The task window can be re-sized, and may be removed from the screen altogether. If you are in a TWIN window and it is inconvenient to go into the task window, pressing [task bottom] forces the task-buffer cursor and window to the end of the task buffer.

When a task is started, half of the available memory is allocated to it for the output: this will be used up gradually, but there is no check for running out of memory. Only one task may run at one time, and a new task cannot be started until the one running has finished or is cancelled.

TWIN can read the error messages from compiler output and go to the offending lines in the source file. With the source code in your current window, and the compiler output in the other window, press [next line]

TWIN now looks for the first occurrence of `line` (in any case of characters) in the compiler output. For example, it might find:

`Error in line 98`

The cursor now moves straight to that line in the source file. In this example, it moves to line 98. You can now correct the error in line 98 and repeat the process to find the next line which the compiler has identified as containing an error.

In the task-linked mode of operation, TWIN must redraw its window to display any information arriving from the task: this slows down the task in hand. If faster operating speeds are required, the task window may either be reduced in size to its smallest dimension (five lines), or turned off completely. Repeated pressing [expand window] while in the TWIN window (as opposed to the task window) reduces the size of the task window, and pressing [close window] while in the task window removes the task window from the screen.

A simple worked example of how to run the ARM assembler from TWIN is given in Appendix B.

## 3.3 Editing

### 3.3.1 Insert and overtype modes

TWIN offers the choice of typing in overtype or insert mode. In overtype mode, the characters typed replace those currently at the cursor position on the screen. In insert mode, they push existing text to the right to make room for themselves. The mode in use is displayed in the bottom-left corner of the screen (the status area). To switch between these modes, use [ins/over]. Certain keys behave differently in the two modes – these differences will be mentioned as the functions of those keys are introduced.

When the text you are typing reaches the right-hand side of the screen, it overflows on to the next screen line; no special character (such as hard or soft returns) is placed in the text at the point where the text breaks to start a fresh line on the screen. Therefore, to start a new line, it is necessary to press [Return]. TWIN imposes no limit on the number of characters that can be typed on one line, other than the maximum buffer size, dictated by the available memory.

Pressing [Return] in insert mode puts a new-line character (ASCII 10) at the current cursor position and the cursor moves to the start of the next screen line. The return character is important as it marks the end of a line to the editor. Return characters are normally invisible. To make TWIN display return characters, press [show new lines]. Return characters show as left-arrows on a highlighted square.

Pressing [Return] in overtype mode moves the cursor to the start of the next screen line.

To correct minor errors while typing, use [Del] to erase characters to the left of the cursor. In insert mode, this removes the character from the screen and buffer; in overtype mode it doesn't remove it, but replaces it with a space.

### 3.3.2 Moving around the text

The cursor keys have a repeat action if they are held down. The screen window can only show part of the program text; this is regarded as a page. To see another page, use the cursor keys as below:

- to move through the TWIN buffer a line at a time (scrolling), use the up or down cursor keys – the screen will scroll when the cursor comes within a few lines of its top or bottom (this distance is the scroll margin, and can be changed using [new margins])
- to move through the text buffer a page at a time, use the up or down cursor keys with [Shift]
- to move to the start or end of the text buffer, use the up and down cursor keys with [Ctrl]
- to move the cursor under the start of words on the preceding line, press [Shift]-[Tab] until TAB cols is displayed on the status line at the bottom of the screen, then use [Tab]
- to move the cursor to the next tab position to the right, press [Shift]-[Tab] until TAB cols is displayed on the status line at the bottom of the screen, then use [Tab]
- to move to a particular line in the buffer, use [go to line]. This displays the current line number (At line nnn, characterxxx, new line:), where nnn is the number of returns between it and the start of the buffer. You may now type a line number, or + then a number to move forwards that number of lines, or - then a number to move backwards
- to move to a known piece of text in the buffer, use [find and replace] (see section 3.4.1).

#### The scroll margins

The screen will normally scroll when the cursor is moved to within four lines of the top or bottom of the screen (the scroll margin) to ensure that text surrounding the cursor can be seen properly. It is possible to alter the action of scrolling by the [new margins] key. Using [new margins] displays the prompt:

```
set Bottom margin, set Top margin or Remove margins ? [B/R/T]
```

```
[new margins] B
```

sets the bottom margin to the cursor position

**[new margins] T**

sets the top margin to the cursor position

**[new margins] R**

removes the scroll margins altogether.

The margins will be reset to TWIN's preferred positions if the size of the window is altered by pressing **[expand window]** **[toggle window]** **[close window]** or changing screen modes. Toggle window operations which do not reset the size of the windows do not reset the margins.

### 3.3.3 Changing the text

#### *Small alterations*

Small changes to text in the buffer are made using the **[Del]** or **[COPY]** keys for single-character deletions backwards and forwards, or by overtyping in overtype mode.

In insert mode, **[Del]** deletes characters to the left of the cursor, **[COPY]** deletes characters to the right of the cursor, and typing inserts the new text at the cursor.

In overtype mode, **[Del]** replaces characters to the left of the cursor with spaces and moves the cursor to the left. **[COPY]** behaves as it does in insert mode, and typing replaces existing text with new text.

To remove a blank line or join two lines, the new-line character, together with any spurious spaces which might be present, should be removed. This is often easiest to do when the new-line character is made visible by pressing **[show new lines]**.

Whole lines may be removed by pressing **[Ctrl]-[COPY]** simultaneously.

#### *Cursor editing*

To copy short sequences of text from the screen, place the cursor at the position where the copied text is required and press **[Shift]-[COPY]** to select cursor editing mode. Next, move the cursor to the text which is to be copied. Pressing **[COPY]** now copies the text character by character. Extra text may be typed in and existing text may be deleted using the **[Del]** key. The copying process cannot continue past a new-line character. Press **[Esc]** to return to insert or overtype mode.

Two drawbacks to cursor editing limit its usefulness: control characters (including the new-line character) cannot be entered as part of the copy, and, if the destination line is before the source line, copying may invoke a line-scroll and so impair a successful copy from that point onwards.

The singular advantage of cursor editing is that it disables the editor commands on keys **[H]** to **[T]** so that you can use those keys to type complete words, phrases or ASCII codes 127-255 with a single keystroke.

Although these techniques are very useful, even more powerful editing commands are provided to delete, move and copy blocks of text.

### **Cutting and pasting text**

These blocks are only limited in size by the text in the buffer. To use any of these commands, you must first define the block of text to be used. This is done by moving the cursor to the start and/or end of the block and marking its position with [mark place].

To delete a block of text, mark the start or end, then move the cursor to the other end of the block and use [delete block].

To move a block of text, mark the start and the end (in either order), then move the cursor to the destination outside the block and use [copy block].

To copy a block of text, mark the start and the end (in either order), then move the cursor to the destination and use [mark place]. From now on, each time [copy block] is pressed, a further copy will be placed at the cursor position, until the marks are erased using [clear marks].

To copy a file from the filing system into existing text in the TWIN buffer, position the cursor where the text is to go, then use [insert file], typing the file name in response to the prompt Type filename to insert: TWIN removes marks automatically after the block delete or move commands. Marks are not removed automatically after the copy command, to allow multiple copies of marked text to be made easily. Marks can be cleared at any time with [clear marks]. It is best to do this as a matter of good practice, as unwanted marks can modify the action of certain commands such as [global replace] and [save file], and can prevent operation of some commands, causing the error message Bad marking.

The number of marks currently set is displayed in the status line at the bottom of the screen. A maximum of two marks is allowed in any of TWIN's buffers: buffers have their own set of marks. This allows copying between one buffer and another, providing that marks have first been cleared in the destination buffer. The text marked should be in the source buffer, and the cursor placed in the destination-position of the current buffer. A copy (but not a move) is then made between the source and destination buffers.

TWIN will use as its source buffer either the buffer which occupies a screen window, or, in cases where only the destination window is on the screen, the buffer which has just been removed from the screen using the [close window] key.

### **3.3.4 Clearing and restoring text**

To delete the entire contents of the text buffer, use [clear text], then use one of the options in response to the following prompt:

Clear text Y (dated), shf-f9 (auto-exec), D (discard)/N

Y kills the current file and begins a fresh one which will be date-stamped.

[clear text] kills the current file and begins a fresh one which will be executable as a command file.

D removes the altered status from the current file. The file remains in memory but it will now be possible to remove it by a [load file] command which does not ask permission before overwriting the file with the new file.

N means do not clear text. [Esc] may also be pressed, in which case the text is not cleared. If no action is taken, the command times-out and automatic escape occurs.

Use [clear text] very carefully. There is no command to restore lost text.

## **3.4 Finding and replacing text**

TWIN provides two very powerful search-and-replace commands, which can perform the following types of functions:

- rapidly changing all or selected occurrences in the TWIN buffer of one string into another string – correcting spelling mistakes, improving consistency, expanding or shortening variable names in programs, expanding abbreviations
- finding the next time a text string appears in the TWIN buffer – moving to a line where the content is known, but the line number is not, checking procedure/function parameters in programs and so on
- counting the number of times a string occurs in the TWIN buffer – checking the size of a document, guarding against over use of particular phrases, analysing style, looking for under-used variables/procedures/functions in programs and so on.

The two commands are:

- **[find and replace]**, which you use to look for or replace a phrase from the current cursor position to the end of the file
- **[global replace]**, which you use to look for or replace a phrase throughout the whole file, or within a block in that file; or to count the number of times a phrase occurs in the file.

You use both commands in the same way. After pressing the function key, you type the text you want to look for – the target string. The function keys have now taken on a new set of meanings, shown in white type on the key card. These keys let you specify the target string in very precise ways. One key gives you the option of entering a replacement phrase – the replacement string.

To search and replace within a block, move to one end of the block and press **[mark place]**, then move to the other end and press **[global replace]**.

### 3.4.1 Simple finding and replacing

To find a phrase, press:

**[find and replace]**

The screen displays the prompt **Find and replace::**. Type the target string, followed by **[Return]**. TWIN now looks for the string, ignoring the case you used to type it. For example, if you specify **book**, TWIN looks for **book**, **BOOK**, **Book** and so on. If it finds it, it shows the prompt:

**C(ontinue)**, **E(nd of file replace)**, **R(eplace)** or **ESCAPE/RETURN**

Pressing **C** causes TWIN to look for the next occurrence of the string.

Pressing **E** causes TWIN to prompt for a replace string and then the command behaves like a global replace from the present position to the end of the file.

Pressing **R** causes TWIN to prompt for the replace string which is used only to replace the string at the cursor position.

Instead of typing a replacement string, you can press:

- **[Return]** to delete the target string
- **[COPY]** to enter the target and replacement strings you specified last time you used **[find and replace]**
- **[Shift]-[COPY]** to enter the target and replacement string you specified last time you used **[global replace]**

You can look for the same target string again by pressing **[find and replace]** then **[COPY]**.

You can repeat the whole find-and-replace operation by pressing **[find and replace]** then **[Return]**.

### 3.4.2 Counting occurrences

To find out how often a phrase occurs in your file, press:

**global replace**

The screen displays the prompt **Find and replace:**. Type the target string, followed by **Return**. TWIN now counts the occurrences of the string.

### 3.4.3 Specifying a target string

This section describes the more sophisticated ways in which, using either **find and replace** or **global replace**, you can specify the text you want TWIN to look for. To do this, you use the special function key commands that become available when you press **find and replace** or **global replace**.

To find:

- a phrase in a specified case, press **exact** then the phrase
- a phrase, part of which must be in a specified case, use **toggle case**. For example, acorn **toggle case** Computers **toggle case** limited would find Acorn Computers Limited and ACORN Computers LIMITED, but not ACORN COMPUTERS LIMITED
- a new-line character, press **new line**
- a control character, press **control** then the appropriate character
- ASCII 128-255, press **set top bit** then **control** and the appropriate character – for example, for ASCII 129, you'd press **set top bit** then **control** A
- ASCII 257, use **set top bit** A

### *Chapter 3*

- any letter, digit, or underscore, use **alpha-numeric**
- any digit (0 to 9), use **digit**
- any character (ASCII 0-255), use **any character**
- any one of a set of characters, for example, xyz, press **[start set] xyz [end set]**. Note that this command is case-sensitive: in this example, TWIN would find y but not Y
- any one of a range of characters, for example, f to i, press **[sub range] f-i**. This command is also case-sensitive: in this example, TWIN would find h but not H
- any character other than that specified, use **[not item]** – for example, to find anything other than n, type **[not item] n**.

You can use **(start set)**, **(end set)**, **(sub range)** and **(not item)** with other single-character specifiers, so that the following examples are all valid specifications of single characters:

**[start set] alpha-numeric \$ [end set]**

matches any alpha-numeric character or the dollar sign

**[start set] a [sub range] z [end set]**

matches any lower-case letter or @

**[not item] [start set] [digit] [end set]**

matches any non-numeric character

**[not item] 3 [sub range] 6**

matches any character other than 3, 4, 5 and 6

**[start set] a [sub range] c-e [sub range] z [end set]**

matches any lower-case letter other than d.

### ***Specifying variable-length target strings***

Two extra function key commands let you specify strings that don't have a fixed length:

[most items]

[many items].

#### **Most items**

This command lets you search for strings made up of one or more of the same character. For example:

[most items] 1 finds any occurrence of 1, 11, 11, and so on

[most items] [alpha-numeric] finds any word

[most items] [digit] finds any number (6, 66, 3454545)

[most items] [not item] [new line] matches a non-blank line.

This command is particularly useful for finding multiple spaces.

#### **Many items**

This command lets you specify the start and end of a target string, without specifying what happens in between. You normally use it in conjunction with [not item], [any character], [alpha-numeric], or [digit]. For example, to find every phrase that starts Acorn and ends Limited, you would type:

Acorn [many items] [any character] Limited

To delete ends-of-lines from a ; character up to the end of the line, you would search for:

; [many items] [any character] [not item] [new line] [new line]

and replace each occurrence with nothing. This particular target-string specification asks TWIN to look for every phrase starting with a semi-colon, then including a series, of unspecified length, containing any characters except the new line character, and ending with a new line character.

The [many items] key is most useful for specifying unimportant filler characters, that is, characters that need not be there at all to match the target string.

### 3.4.4 Specifying a replacement string

Straight after typing your target string, you can specify a replacement string by typing **replace by** then the string.

As before, you can use any of the special search-and-replace function key commands.

You can transfer the target string into the replacement string using the command **found string**. For example, to replace all occurrences of micro with microcomputer the command would be:

**global replace** micro **replace by** **found string** computer **Return**

The whole find string **micro** is incorporated into the replace string.

Selected characters from an ambiguous string specification may also be used in the replacement string. To do this, you use **found section** and specify their position in the ambiguous field:

**global replace** **alpha-numeric** **alpha-numeric** **alpha-numeric** abc  
**replace by** **found section** 0abc

In this example, the found string refers to the ambiguous field **alpha-numeric** **alpha-numeric** **alpha-numeric**. The 0 indicates that the character in the first position (0) is the one to be incorporated in the replacement string. The characters in the second and third positions are thrown away. If, for example, the third character was also needed, then the line:

**global replace** **alpha-numeric** **alpha-numeric** **alpha-numeric** abc  
**replace by** **found section** 0 **found section** 2abc

would capture it.

Therefore, the parts of the found string which can be used in the replacement string are those that were specified by ambiguous character or string specifications in the target string. To identify which parts of the found string to use, the ambiguous elements are numbered according to their position in the target string from left to right starting at 0. For example, to bracket words (one or more alphabetic/numeric characters, preceded and followed by anything else), the syntax would be:

**global replace    most items    alpha-numeric    replace by    found section**

or

**global replace    most items    alpha-numeric    replace by    found section.**

## 3.5 Files

### 3.5.1 Saving and loading files

#### *Saving text files*

The **save file** key is used to save a TWIN document to the filing system. **save file** produces a bleep to warn you that a file of the same name in the current directory of the filing system may be about to be overwritten. This is in case **save file** was selected by mistake. There are a number of ways in which to give a filename to the document being saved:

- type a filename then press **Return** in response to the prompt **Type filename to save:**
- save the document using the same filename as last typed for save, load or insert by pressing **COPY** **Return** after **Type filename to save:** (After pressing **COPY** but before pressing **Return** the filename may be altered)
- place **> filename** on the first line in the buffer and then press **Return** when prompted for the filename. This line must be less than 129 characters long but **> filename** need not be at the start of the line.

To save just part of the buffer in a file, use **mark place** to mark one end of the part required, move the cursor to the other end and then use **save file**. The prompt:

**MARK TO CURSOR save:**

will be given instead of:

**Type filename to save:**

No bleep will be given.

### *Chapter 3*

#### *Loading text and program files*

The contents of the currently selected buffer may be replaced at any time using [**load file**]. The prompt:

Type filename to load:

will appear and the filename followed by [**Return**] should be typed.

The file named should be an existing file on the current filing system. Any type of file currently supported by the filing system or language will load into TWIN. The wildcard \* character may be used where the filing system allows: for example, [**load file**] P\* instead of [**load file**] Paintpr4. To ease the typing of load names which are complex due to directory routing, the special [**load via \* path**] command is available (see below).

Some types of file, such as ARM object code, will load correctly but will not be very readable, as they do not consist of many ASCII strings. They can be edited, however. Some programming languages convert keywords in their programs into ASCII codes above 127. Programming languages may provide commands to pass their tokenised programs to the editor, expanding them to readable plain text in the process: for example, ARM BASIC. Refer to the language's documentation for guidance.

A file can also be loaded using the first and second methods outlined in section 5.1.1.

#### *Combining files*

Files can be added to the text in the buffer rather than completely replacing it by using [**insert file**]. The file is inserted at the cursor position.

### The load via \* path facility

[load via \* path] uses a text file containing directory information, giving TWIN the ability to search for files in places other than the current directory. To understand the working of [load via \* path], imagine there exist three files, *filea*, *fileb* and *filec* which are regularly used. Creating a TWIN file called *path* in your currently selected directory similar to the example below will then allow TWIN to load any one of these files without the need to move out of the currently selected directory, or to give a full file path name in response to [load file].

```
[Return]  
$.arm.library [Return]  
$.arm.peter.utils [Return]  
$.armdoc.current [Return]
```

This example assumes *filea* has already been saved to \$.*armdoc.current*, *fileb* to \$.*arm.peter.utils* and *filec* to \$.*armdoc.current*. The first line is blank thus also allowing TWIN to look in the currently selected directory.

Once the path file has been set up, press [load via \* path] and enter the filename in response to the prompt

Type filename to load (via \*path):

When the file has been found, it will be displayed and the full path name will be placed in the file name position on the status line. If the file path cannot be found, the file system's error message will report Not Found. If, on the other hand, one of the three files cannot be found, then TWIN will report File not Found.

Names and delimiters can be built up as required. Use only I, ASCII 10 or ASCII 13 as delimiters. Using I (which can be read as or), the example path file would look like this:

```
|$.arm.library|$.arm.peter.utils|$.armdoc.current
```

There is still the null name before the delimiter, so the current directory would be searched first. The directory names given can be relative to the current directory. The path file must not be longer than 1024 bytes.

### 3.5.2 Printing files

TWIN can print the contents of the currently selected buffer directly to a printer and the output will simultaneously appear on the screen, scrolling at a rate determined by the printer's baud rate or the size of the printer's internal buffer or both. If a single mark is in the text, the printed output will consist only of the text between the mark and the cursor. If two markers exist, the output will be the text between the marks.

TWIN normally makes no attempt to format lines and consequently words tend to become split at the right-hand edge of screen lines and printed lines. Word-splits tend not to occur very often in source code where the average line length is less than 80 columns, but for sections of explanatory text or for documents, word-splitting is not acceptable. [format] will act on a paragraph of text and remove any split words by selectively replacing some spaces with a new-line character. The new line is initially placed after the 80th character, but if this is not a suitable point to break the text, a search is made backwards until a space is found. The paragraph is taken as the text between the cursor position and the next new line which is followed by a character, a space, full stop, comma, semi-colon or colon.

TWIN cannot right justify the text.

To print text:

- use [format] to tidy your text paragraphs
- use [mark place] to define a block within the buffer, if you don't want to print the whole buffer
- press [print text] to start printing.

## 3.6 TWIN function keys

[go to line]

TWIN computes line numbers by counting new-line characters and three different goto jumps are allowed:

- move to a given line-number. For example, [go to line] 765
- move to a mark by using m1 or m2. For example, [go to line] m1
- move by a number of characters by + n or - n. For example,  
[go to line] -50.

[show new lines]

The new lines can be shown as a special character, a highlighted left-arrow, so that they can be seen clearly. Press this key to either reveal or hide the new lines.

[toggle window]

Toggles between windows. If the second window is not already on screen, then the screen is split and the second window is opened. The selected window's status line is highlighted.

[connect buffer]

Connects a window to a different buffer from the one currently using it, by specifying the buffer's number. There are 10 buffers, called 0 to 9. Alternatively, the plus and minus sign (+ and -) will increment or decrement the buffer attached to the current window. Any response other than 0 to 9, + or - lists the buffer directory on the screen. Buffers cannot be connected while a pushed buffer is present (see [push]).

[command]

Commands to the computer's operating system can be given following the TWIN\* prompt which appears on the screen; for example, obtain a CATalogue, run BBC BASIC, and so on. Afterwards, re-enter TWIN by pressing [Return].

**[insert/over]**

Toggles between insert and over modes for the particular buffer in use.

**[expand window]**

Enlarges the current window by one line, if there is enough room. The other window is reduced by one line. Cursor positions in both windows remain on screen.

**[load file]**

Loads a text file into the current buffer. If modified text will be erased in the process, TWIN asks you to confirm this.

**[insert file]**

Inserts a text file at the current cursor position.

**[close window]**

Closes the current window (the text is not harmed) and displays the other window, which may or may not have been on the screen.

**[load via "path"]**

This loads a text file like **[load file]** but uses a file path such as *m2path*, *cpath*, or *TWINpath* provided by the filing system. The path file must not be longer than 1024 bytes.

**[save file]**

There are a number of variations:

- save all text
- save text from a mark to the cursor
- save using the current filename
- save using a name supplied in the text.

TWIN date-stamps the file and a beep sounds to remind you that save has been pressed. **[Esc]** cancels the save if it isn't wanted.

## *Chapter 3*

### **[new margins]**

Controls the scrolling action of a window:

- **B** to set the bottom scroll margin to cursor line
- **R** to remove the top and bottom scroll margins
- **T** to set the top scroll margin to cursor line.

A mode change or window command resets the margins to their default value.

### **[next line]**

TWIN can look for occurrences of **line** or **LINe** in the error messages of a compiler's output and use the information to go to that line in the source file's buffer.

### **[find and replace]**

Begins a find or a selective find and replace.

### **[exit to language]**

After checking if there is any altered text to be saved, TWIN will exit to a named language (or to the operating system if just **Return** is pressed).

### **[global replace]**

Performs an automatic count of occurrences (with no replace) of a non-selective find and replace. The whole text or just a block of text may be specified.

### **[set mode]**

A number of screen modes are available:

- **3**: ordinary screen mode
- **D**: Descriptive: shows all keys and details commands
- **K**: Key legends: shows the function keys names
- **T**: Time: shows the time on the top line.

**[enter character]**

Inserts the character from the keyboard directly, or allows the character's decimal code to be typed.

**[pop]**

A pushed buffer will be recovered from its stack. See **[push]**.

**[mark place]**

Puts a marker into the text at the cursor position. Each buffer and each pushed buffer (see **[push]**) may have up to two marks. When you set a mark, the status line shows One mark or Two marks.

**[clear marks]**

Clears all place marks in the current buffer.

**[format]**

Formats lines to a width of 78 characters or less, starting from the line the cursor is on, and ending at the first new line that is followed by a non-alpha-numeric character.

**[push] .**

Pushes the current contents of the buffer into stack memory. Only buffers currently selected may be pushed, so two (represented by each screen window) may be pushed at any one time. A new file may be loaded into the buffer vacated by the push. The newly loaded file may also be pushed. In this way files can be loaded and viewed without having to open a new buffer to receive them. TWIN can push any number of files, subject only to limitations of memory.

**[copy block]**

Copies the text between two marks to the cursor. If there are no marks in the current buffer, text is copied from the alternate buffer (provided it has two marks set in it).

**[move block]**

Moves the text between two marked places to the current cursor position which should be outside the marked area. The marks are then cleared.

*Chapter 3*

**[print text]**

Prints out the whole, or marked, text.

**[delete block]**

Deletes the text between the cursor and the marked place. The mark is then cleared.

**[toggle LF <-> CR]**

Exchanges all carriage return |M and line feed |J characters currently in the file.

**[auto bottom]**

Toggles between task-linked and task-unlinked states. With task-linked, whenever the task window is redrawn, the cursor is moved to the end of the buffer. With task-unlinked, no change is made to the cursor position.

**[bytes free]**

The number of bytes free in TWIN's section of memory. It is possible to adjust TWIN's load address to obtain more space.

**[clear text]**

Deletes all text in the buffer.

**[task status]**

Starts and finishes a single background task. It will also show whether a task is currently running. The output of the task is appended to the end of the current buffer. While it is running, editing can be done on the appended part of the current file, or in the whole part of a file in a different buffer. Keyboard characters are sent to the task.

**[task bottom]**

Moves the cursor position to the end of the task window.

## 3.7 Concurrency: a worked example

### 3.7.1 TWIN and AAsm, the ARM assembler

Switch on the computer. When the ARM prompt appears, set the time. Load TWIN by typing TWIN.

Type the following into TWIN:

```
; -> test
ORG #1000
SWI 1
- "Hello World",10,13,0
SWI 17
END
```

This short piece of text is an AAsm file, now residing in TWIN's buffer 0. The source file can be saved using the name taken from the ; -> line at the top of the file using [load file] [Return].

Next, obtain a second buffer by pressing [toggle window]. A second, blank, window is now on the screen. AAsm can be called from this window by pressing [task status]. This brings the prompt Command: on to the status line of buffer 1. If AAsm is not in the currently selected directory or library, the full pathname must now be given. However, if AAsm does reside there, the response should be:

\*AAsm

AAsm will load and in the window of buffer 1 you will see:

```
ARM stand alone Macro Assembler Version x.xx
Entering interactive mode
Action:
```

The status line will show:

1 Task running in this window

### *Chapter 3*

The response to the prompts should be:

Action: ASM

Source file name: test

Code file name: testc

whereupon the file test will be assembled.

Pass 1

Pass 2

Assembly complete

No errors found

Action:

Both passes ran, with no errors reported.

The program can now be executed by leaving AAsm:

Action: quit

This ends the task and leaves buffer 1 as a TWIN buffer, rather than a buffer shared by AAsm. The message Task running in this window is removed from the status line. To run testc from TWIN, it is made another Task. Press:

**[task status]**

In response to the prompt Command: type \*testc . The task will run, printing "Hello World" to the buffer and causing the status line to report briefly:

1 Task running in this window

before the task ends and control passes back to TWIN.

### 3.7.2 Using a task

The command needed to recompile a C source, for example, would be:

```
cc bench.sieve -link
```

Execute the command file as a task by typing:

```
[task status] bench.sieve
```

Now files can be edited in the usual way in the other window, the text in the task window being updated almost as quickly as output arrives. As errors appear, the relevant source file can be loaded and [next line] used to move to the error line in the source.

## 3.8 Error messages

TWIN always reports detected errors and the command which is in error is not obeyed. The text in the buffer will be left unchanged but the editor may clear marks or move the text cursor to the start of the buffer or screen line.

### Bad mark number

You have tried to set more than two marks or have not set the right number of marks for use with this command. For example, you are trying to delete a block with 2 or 0 marks set rather than 1. The marks will be cleared.

### Bad number

You have typed a number outside the allowed range of letters rather than digits, in response to an editor prompt for a number. Note that numbers are always decimal - you cannot use & to specify a hexadecimal number.

### Bad use of stored name

While using [insert file], you have either pressed [Return] or [COPY] [Return] in response to the prompt for the file to insert, instead of supplying a filename.

### Line not found

You have specified a destination line with [go to line] which is beyond the end of the file. (To count the number of lines in the buffer, use [global replace] [newline] [Return].)

### No infile name found

While using the load, save, or insert file commands, you have responded to the prompt for filename by pressing [COPY] or [Return] without previously typing a filename. If you did type a filename before pressing [Return], the file is either missing or an invalid name was used.

### *Chapter 3*

#### **No previous string**

After using the [find and replace] or [global replace] keys, you have responded to the prompt by pressing [Return] but have not previously specified a find-and-replace string for that command. (Note that the string previously used with [find and replace] is available for use with [global replace] or vice versa, by pressing Shift - [COPY].)

#### **No room**

There is not sufficient room in the text buffer for the file that you have tried to load or insert, or the text buffer is full so that there is no room left for you to type more text. To release more memory, the contents of a buffer can be saved and the buffer closed.

#### **Insufficient memory to (re)start TWIN at this address**

This error message is most likely to occur when you have forgotten that the language in use is running under TWIN and you attempt to edit the program seen as an obstacle. To recover from this error, reload the language.

#### **Relative move out of file**

On a goto command, the forwards or backwards jump specified would take the cursor out of the file.

#### **No more lines in other window**

[next line] has been pressed, but there are no more error lines specified for TWIN to go to.

#### **Unsplittable line**

A line has been found which has more than 77 characters, none of which are spaces or new-line symbols.

## **3.9 Tasks and events**

The swapping between the task and the main TWIN functionality is performed by events from the BBC I/O system. Consequently, the task is denied the use of events. TWIN swaps the environments created by Control and SciEnv as swapping between itself and the task occurs: thus escape codes will be sent to the task as will errors.

When a task is running, the CSD or filing system should not be changed because this could confuse the task.

## **3.10 Moving TWIN**

The release version of TWIN runs at &1D0000 in 2- or 4-Mbyte ARM, &D0000 in 1-Mbyte ARMs, or &10000 in 1/4-Mbyte ARMs, the Executive ROM performing the translation automatically. Note that the Executive cannot deal with the &3D0000 address in a 2-Mbyte machine. This gives either 3/4 or 1/4 of the machine to background tasks. The load and execute addresses of TWIN can be altered by using a program on the release disc called MOVEFILE. For the ARM\* command line, the instruction

**movefile TWIN address**

should be given. The address given should be the new address for TWIN, for example, &2D0000. The address range accepted by the movefile program is &1000 - &3FFFFFF, but a high-memory address should be chosen with care since MOVEFILE doesn't check whether there is sufficient physical memory to hold the TWIN program.

### 3.11 TWIN called from ARM BASIC

If the language BASIC is running on the ARM, the command TWIN or EDIT will cause the following actions to take place:

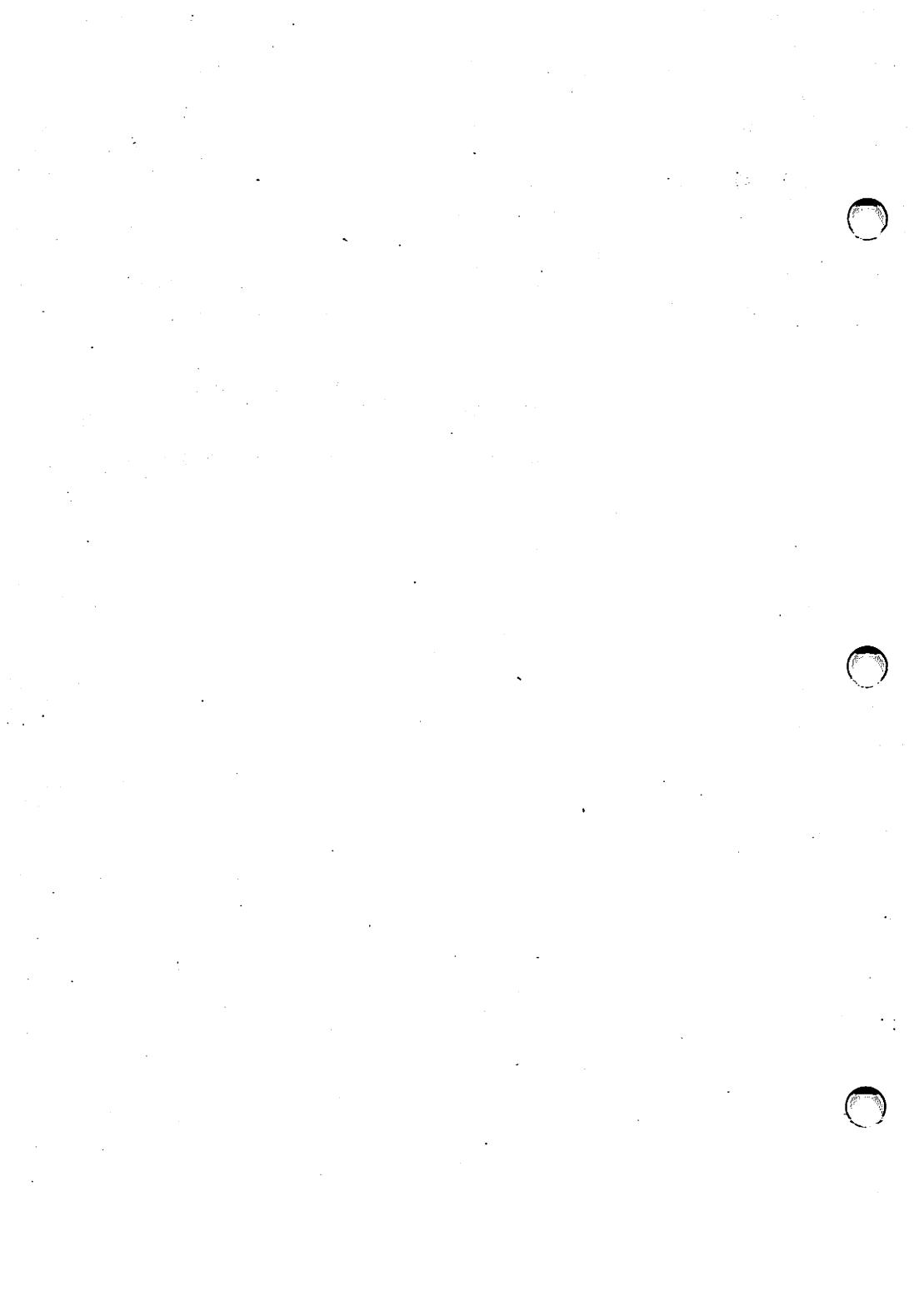
- TWIN will be loaded into memory from the filing system
- the AB program will be copied into memory, the copy being an ASCII file version of the program
- TWIN will be informed of the location of the ASCII file and will start up with it displayed in buffer 0.

Return to AB is by the normal [exit to language](#).

If the start and end address of the ASCII file is known, it can in fact be loaded directly into TWIN by a command from the ARM\* prompt. The syntax to do this is:

**TWIN *s*ssssss,fffffff;filename**

where *s* is the hexadecimal start address and *f* is the hexadecimal end address. The filename is the name the ASCII file will have as an AB program.



## 4. The External ROM Expansion Card

### 4.1 Introduction

The Communicator External ROM Expansion Card consists of a cased circuit board with internal ROM and RAM and four externally accessible EPROM ZIF sockets. In addition there is a user accessible hardware Reset switch. Protruding from the left hand side of the case is an edge-connector terminated ribbon cable for attaching the Expansion Card to Communicator.

The internal ROM contains a set of general purpose software utilities and a trace facility to assist in the development of applications software.

The 32KB of internal battery-backed CMOS RAM allows the developer to load data into Communicator without occupying normal Communicator RAM or the need to blow EPROMs.

The external ZIF sockets allow up to four user-developed EPROMs to be installed in Communicator's memory map.

## **4.2 The External ROM Expansion Card**

### **4.2.1 Introduction**

This section deals with:

- The connection and operation of the External ROM Expansion Card.
- The Reset button.
- The installation of external EPROMs.
- Use of the internal battery-backed RAM.

### **4.2.2 Connecting the External ROM Expansion Board**

Before attempting to connect the External ROM Expansion Card to Communicator, ENSURE THAT COMMUNICATOR IS POWERED DOWN.

The Expansion Board is connected to Communicator via a single wide Ribbon Cable terminated by a female Edge Connector. To connect the Expansion Board, locate the protruding male Edge Connector on the right hand side of Communicator and push the female Edge Connector onto it. A location slot on the Edge Connector ensures that the cable cannot be attached incorrectly.

Once the Ribbon Cable has been attached, Communicator can be powered on.

To check that the External ROM Expansion Card is correctly installed, invoke BASIC and at the BASIC prompt enter:

\*HELP SYSUTILS

This should result in the following output:

```
SYSUTILS      (v01.00)
  *cbfree
  *handles
  *map
  *modules
  *irqlist
```

If the above, or something very similar is NOT displayed then either the Expansion Board has not been properly installed, or there is a hardware fault.

Repeat the installation procedure ensuring that the Ribbon Cable is properly attached.

If the \*HELP SYSUTILS command still fails, contact Acorn Software Support.

#### **4.2.3 The Reset button**

Located on the top of the External ROM Expansion Card is a RED button. This button allows the user to execute a Hardware Reset on Communicator, without having to power it down.

#### **4.2.4 The External EPROM ZIF Sockets**

There are four ZIF sockets provided on the External ROM Expansion Card. These allow the developer to install EPROM based software into the Communicator's memory map. The sockets are configured to operate with EPROMs of type 27256, 27512 or 27101.

The ZIF sockets are numbered zero to three and map onto the following contiguous areas of Communicator Memory:

ZIF-SKT 0	&B00000 to &B1FFFF
ZIF-SKT 1	&B20000 to &B3FFFF
ZIF-SKT 2	&B40000 to &B5FFFF
ZIF-SKT 3	&B60000 to &B7FFFF

#### *Installing EPROMs*

Before attempting to install an EPROM in the External ROM Expansion Card, ENSURE THAT COMMUNICATOR IS POWERED DOWN.

Select an empty ZIF socket and confirm that it is "open" - the ZIF socket "arm" should be in the UP position.

Ensure that the EPROM is correctly oriented - the notched end of the EPROM should be at the same end as the marked PIN 1 of the ZIF socket.

Insert the legs of the EPROM into the ZIF socket.

Push the ZIF "arm" DOWN to lock the EPROM in place.

Communicator can now be powered up.

If you encounter difficulty accessing the EPROM, repeat the installation procedure ensuring that:

- The EPROM has been correctly blown.
- The EPROM is of the correct type (27256, 27512 or 27101).
- The EPROM is correctly oriented.
- The "arm" on the ZIF socket is DOWN.
- The External ROM Expansion Card is properly connected.

If you still encounter problems, contact Acorn Software Support.

### 4.2.5 The Internal RAM

The External ROM Expansion Card contains 32KB of battery-backed CMOS RAM, located at &BA0000 to &BA7FFF.

The internal RAM can be used to store software or data that the developer does not wish to load into normal Communicator RAM. The CMOS RAM can also be used as a pseudo-ROM; being battery-backed, the contents are retained after Communicator has been powered down.

Software can be installed in the CMOS RAM via the "\*LOAD" command. For example:

\*LOAD PROG &BA0000

would load the file "PROG" from the start of the CMOS RAM area. It would then be linked into the Communicator module structure by

\*RESET

## 4.3 The System Utilities

### 4.3.1 Introduction

The Communicator System Utilities package provides the systems programmer with five tools to assist with the installation and maintenance of Communicator software.

The Utilities Software is located in the Expansion Board internal ROM.

The five utilities provided are:

CBFREE	Displays the number of free control blocks.
HANDLES	Lists the handles currently assigned by the system.
MAP	Provides a map of system RAM allocation.
MODULES	Lists the currently installed modules.
IRQLIST	Lists the entries on the system interrupt queue.

### 4.3.2 Invoking the System Utilities

Entering

\*HELP SYSUTILS

from the BASIC command line will result in a list of the available Communicator System Utilities being displayed, thus:

SYSUTILS (v01.00)

\*cbfree  
\*handles  
\*map  
\*modules  
\*irqlist

To invoke a Utility from the command line, enter \*<utility-name> where utility-name is one of "cbfree", "handles", "map", "modules" or "irqlist". For example to invoke the MAP Utility enter

\*map

from the command line.

### **4.3.3 The CBFREE Utility**

The CBFREE System Utility is used to obtain information on the number of free CONTROL BLOCKS within the system.

Entering

\*cbfree

from the command line results in a message of the following format being displayed:

xxx/yyy Control Blocks free.

where xxx is the number of sixteen bit control blocks available and yyy is the number of eight bit control blocks available.

For information on the use and format of Control Blocks refer to the Communicator Systems Manual.

### **4.3.4 The HANDLES Utility**

The HANDLES system utility provides information on module handles currently assigned by the Communicator MOS.

Entering

\*handles

from the command line causes the current number of handles and a title line followed by one or more lines of handle information to be displayed. If no handles are currently allocated then the message

No handles.

is returned.

Handle information is displayed on a line per handle basis and is columnnated to simplify interpretation.

The following is an example of possible output from the HANDLE Utility:

1 handle.			
Address	Control block	Direct page	Module name
E450	E45C 94FF FFFF 10FF	D100	NET

In this case only one handle is in use.

The value E450 in the "Address" column is the address of the the Handle Control Block.

The values E45C 94FF FFFF 10FF in the "Control block" column are the contents of the Handle Control Block.

The value D100 in the "Direct page" column is address of the Direct Page of the handle's associated Module.

The string "NET" in the "Module name" column is the name of the handle's associated Module.

For information on HANDLES, CONTROL BLOCKS and MODULES see the Communicator Systems Manual.

For information on DIRECT PAGE see the 65816 Programmers's Guide.

#### **4.3.5 The MAP Utility**

The MAP Utility provides a break-down of Communicator system RAM allocation.

System RAM allocation is displayed on a line per "chunk" basis, a "chunk" being a contiguous block of allocated memory of arbitrary size.

Line entries are in columns to simplify interpretation.

The first column contains the length of the chunk in bytes.

The second column contains the name of the module that owns the chunk, (e.g. MOS). Where there is no name, the memory was allocated below the level of the MOS.

The third column contains the start and end addresses of the chunk in brackets. Entries in the third column are indented to illustrate the tree structure of memory allocation. For example, the first entry in the map describes the WHOLE of the Communicator System RAM and all other start-end address entries are indented relative to it.

The fourth column is used to display the name of a RAM-resident or transient module occupying the chunk if applicable. Where this is the case, the second column will contain "MOS" as the owner.

To invoke MAP utility enter:

\*map

from the command line.

The following display fragment is taken from the output produced by the MAP Utility.

FFFF00	(000000,FFFF00)
080000 RAM	(010000,090000)
010000 RAM	(010000,020000)
000C00 ViewData	(020000,020C00)
004700 BASIC	(020C00,025300)

The first entry in the above example describes the whole of the Communicator System Address space. There is no owner. The address space is from &000000 to &FFFF00. Notice that all following column-threes are indented relative to this entry, reflecting the fact that they lie within it.

The second entry describes the whole of allocatable RAM, from &10000 to &90000. Again notice that all following column-threes are indented relative to this entry's because they have been allocated from within it.

The third entry belongs to the RAM filing system and is 64KBytes in size.

The fourth entry belongs to the Viewdata module and is &C00 bytes in length.

The fifth entry belongs to the BASIC module and is &4700 bytes in length.

Notice that for start-end address columns of the last three entries are all in line indicating identical levels within the memory allocation tree.

For information on the MOS, MEMORY ALLOCATION and MODULES see the Communicator Systems Manual.

#### **4.3.6 The MODULES Utility**

The MODULES Utility provides information on all modules resident in the system, both in ROM and RAM. In addition checksum data on the system ROMs is displayed along with a calculated Release-ID.

Module data is presented on a module per line basis. Each line being in columns to simplify interpretation.

To invoke the MODULES utility enter:

\*modules

This causes a title line to be displayed followed by the module information. After the module information has been output, the system ROM checksums are displayed followed by the calculated Release-ID.

##### ***Module Entry Fields***

The "Name" field corresponds to string found in the name field of the Module header. A name followed by a slash (/) indicates that the module supports star (\*) commands.

The "Version" field corresponds to the value found in the version field of the Module header.

The "Addr" field gives the start address of the Module.

The "length" field gives the length of the Module in bytes.

The "DP" field gives the address of the direct page, (if allocated), assigned to the module.

The "Descript" field displays the contents of the Least Significant Byte of the Module header flags field in bits.

The "Xflags" field give displays the contents of the Most Significant three bytes of the Module header flags field in Hexadecimal.

The "UC" field displays the USE COUNT for the module. The USE COUNT reflects the current number of references, via OPRFR, to the module, (Sometimes referred to as the INDIRECT COUNT). This will only appear after the Module has been referenced at least once.

The "MD" field displays the address of the Module Description Block.

The "Indn" field displays the address of the indirection block associated with the Module. This will only appear if the Module has been referenced at least once.

The "Csum" field is the calculated checksum for the Module. If the calculated value does NOT match the checksum field of the Module header, it is followed by an asterisk (\*).

##### ***ROM Entry Description***

The module entries are followed by the ROM entries.

Each ROM entry consists of the ROM name, (ROM A -> ROM D), followed by a calculated checksum. If the calculated checksum does not match the actual ROM checksum, it is followed by an asterisk (\*).

Following the four ROM entries is the Release-ID which is calculated by Exclusive ORing the four ROM checksums.

***Example Output from the MODULES Utility***

Name	Version	Addr	Length	DP	Descript	Xflags	UC	MD	Indn	Csum
BASIC.	01.00	F84B00	52EF	.....B.	000001	E21C	8ED2*			
E.VT100	01.00	FC6400	0F0A	D600	.D....BP	000000	01	E264	E2A0	EEBC
MOS/	01.00	FFAE00	5065	0000	.....B.	000000	02	E0C0	E318	7966

ROM A	461F
ROM B	D7C9*
ROM C	3FEE
ROM D	E383
RelID	4DBB

In the above example the first Module entry is for the BASIC Module, version 1.00. The Start Address of the BASIC Module is at &F84B00 and it is &52EF bytes in size. The Module has no Direct Page assigned, has not been referenced by any other Modules and has no associated Indirection Block. The calculated checksum is &8ED2 which does not tally with the checksum in the Module header.

In the above example the third Module entry is for the MOS module, version 1.00. The Start Address of the MOS Module is at &FFAE00 and it is &5065 bytes in size. The Module has a Direct Page assigned to it at location &0000, is currently being referenced 2 modules and has an associated Indirection Block at &7966. The slash (/) after "MOS" indicates that the Module has some associated star (\*) commands.

The ROM entries in the example show that for ROMs A C and D, the calculated checksums agree with the ROM checksums. The checksum failed for ROM B. The Release-ID calculated for the system ROM is &4DBB.

For information on MODULES, CONTROL BLOCKS, INDIRECTION BLOCKS etc refer to the Communicator Systems Manual.

#### 4.3.7 The IRQLIST Utility

The IRQLIST Utility provides information on all of the entries on the Communicator's System Interrupt Queue.

To invoke the IRQLIST Utility enter

\*irqlist

from the command line.

This causes a title line to be displayed followed by the Interrupt Queue entries on a line per entry basis. Each entry is columnated to simplify interpretation.

##### *IRQ Entry Fields*

The "Cblk" field contains the address of the entry's associated Interrupt Control Block.

The "Hwaddr" field contains the address of the hardware status register associated with the entry.

The "Enaddr" field contains the address of the interrupt queue entry itself.

The "MSK" field contains the initial value to be masked (i.e logically AND'd) with the contents of the associated hardware status register.

The "EOR" field contains a value to be Exclusive OR'd with the contents of the associated hardware status register after it has been masked.

The "AND" field contains a secondary value to be AND'd with the contents of the associated hardware status register after it has been masked and EOR'd.

The "DP" field contains a value to which the D-register of the 68516 should be set prior to entering the associated Interrupt Handling routine.

The "P" field contains a value to which the 65816 status register should be set prior to entering the associated Interrupt Handling routine.

The "Irqrtn" field contains the entry point to the associated Interrupt Handling routine.

The "Irqcnt" field contains a count of the number of times the associated Interrupt Handling routine has been invoked.

The "Module" field contains the name of the Module which installed or last modified this entry on the Interrupt Queue.

##### *Example Output from the IRQLIST Utility*

Cblk Hwaddr Enaddr Msk EOR AND DP Pri P Irqrtn Irqcount Module

E078 45FE00 00A55B 0C 00 08 AC00 21 B4 FFCD00 000000D MOS  
E0A8 42000D 42000E 90 00 10 AC00 10 B4 FFCCBA 0000000 MOS

In the example the first entry on the Interrupt Queue has an Interrupt Control Block at &E078. The associated hardware status register is at address &45FE00. Data read in from the hardware register is AND'd with &0C then EOR'd with &00 and finally AND'd with &08. If the result of the logical operations is non-zero the 65816 D-register is set to AC00, the 65816 status register is set to B4 and the interrupt routine at location &FFCCBA is invoked. The entry has a priority of &21 and has been invoked &0D times. The entry was installed or last modified by the MOS module.

For information on MODULES, CONTROL BLOCKS and INTERRUPTS refer to the Communicator Systems Manual.

## 4.4 The Trace Utility

### 4.4.1 Introduction

The TRACE program provides debugging facilities for 65SC816 object programs. These will usually have been created using the MASM or BASIC assemblers. TRACE takes the form of a 'language' with the prompt commands as two letters and all numbers in hexadecimal. Facilities include dumping the contents of memory, interpreting memory (disassembly), patching memory and registers, and the tracing of instructions.

Except in special circumstances, instructions executed by the program are traced. In most cases this is done by placing them in a 'scratch pad' and allowing them to execute; in other cases (for example, JSR and BRK), execution is performed by simulation.

Trace information is reported before the execution of each instruction; the amount of this information is determined by the current reporting level. Errors are reported at the global reporting level (defined by the 'RT' command) and this may differ from the current reporting level.

It is possible to define an interrupt key. This can be used to break into tracing or lengthy print operations. The interrupt key defaults to CTRL.

The following Chapters describe the various types of TRACE commands. In the descriptions, items in angled brackets < > stand for classes of items, and square brackets [ ] indicate optional items.

### 4.4.2 Use of breakpoints and reporting

This group of commands includes the following:

BS	Set breakpoint
BC	Clear breakpoint
BO	Clear all breakpoints
DB	Display breakpoints
RT	Set global reporting level
RH	Set reporting high memory point
RL	Set reporting low memory point
PH	Display reporting high memory point
PL	Display reporting low memory point
ST	Stack trust
TC	Clear trust address
TS	Set trust address
DT	Display trusts
IK	Set interrupt key
EN	Set entry point for trace
CO	Continue tracing
SS	Snapshot

Each of the commands is described below.

#### *BS - Set breakpoint*

This command has the following format:

BS <address> [<optional count>]

It will set a breakpoint at <address>. If the count is not specified, zero will be assumed. If specified, it may be in the range 0-&FFFF.

When <address> is reached and the count is zero, TRACE will 'break-in' and report the current processor status; otherwise it will decrement the count and continue.

**BC - Clear breakpoint**

This has the format:

BC <address>

It removes the breakpoint at <address>.

**BO - Clear all breakpoints**

This clears all the breakpoints in the table.

**DB - Display breakpoints**

This gives a list of the currently-set breakpoints, with their current counts, in descending order of address, for example:

03191D 0000  
031914 0020

The next time the break-point at address &03191D is encountered, trace will break-in. The break-point at &031914 has another 32 (&20) executions before break-in.

**RT - Set global reporting level**

This command has the format:

RT <reporting options>

It defines the 'level' of tracing information output. The following reporting options are available:

Option	Meaning
NONE	Report nothing
AB	Report all but the following options (defined)
ALL	Report everything described below
ADDR	Report the execution address
HEX	Report the opcode in hexadecimal form
OP	Report the opcode in mnemonic form
EA	Report effective address of operand
A	Report contents of A register
B	Report contents of B register
D	Report contents of D register
X	Report contents of X register
Y	Report contents of Y register
P	Report contents of flag register
S	Report contents of stack pointer

The default reporting level is 'ALL'. The output from TRACE when everything is being reported looks like this:

001900 B1 00 LDAIY &00  
HA=010A X=FF Y=20 B=03 D=3291 S=1C23  
P=N,MX.... Eff Add = 031297

The first line is the disassembly of the current instruction. The second line gives the contents of registers HA, X, Y, B, D and S. The last line gives the status flags and the effective address of the instruction.

## **Chapter 4**

### ***RH - Set reporting high memory point***

This command defines an address in memory above which reporting will not take place. It has the format:

**RH <address>**

Reporting will be suppressed if the program counter equals or exceeds <address>. The default value is &FFFFFF, the top of the address range, so reporting always occurs at default value.

### ***RL - Set reporting low memory point***

This command defines an address in memory below which reporting will not take place. It has the format:

**RL <address>**

Reporting will be suppressed if the program counter is less than <address>. The default value is &000000.

### ***PH - Display reporting high memory***

This displays the current value of the reporting high memory point, as set by the last RH command, or &FFFFFF by default.

### ***PL - Display reporting low memory***

This displays the current value of the reporting low memory point, as set by the last LH command, or &000000 by default.

### ***ST - Stack trust***

This command suppresses reporting in subroutines if the stack pointer goes below a specified level. Its format is:

**ST <address>**

It is used in conjunction with the TS command: TS suppresses reporting in a given subroutine whereas ST suppresses ANY subroutine when the stack pointer has gone below the specified level.

Reporting resumes when the stack pointer becomes equal to or greater than the specified Stack Trust Level.

### ***TS - Set trust***

This command adds a trust address to the 'trust table'. It suppresses the trace output in a particular subroutine. The format of the command is as follows:

**TS <address>**

If, on tracing a JSR or JSL instruction, <address> is found to be in the trust table, reporting will be temporarily turned off for this and all subsequent subroutines. Reporting is resumed when the subroutine is exited. The subroutine is deemed to have exited when the stack pointer is raised to level of the stack immediately prior to entering the subroutine.

### ***TC - Clear trust address***

This removes a trust address from the trust table. Its format is as follows:

**TC <address>**

**DT - Display trust addresses**

This command displays the current trust addresses. They will be given in descending order of address and will look something like the following:

```
34043  
23002  
21952
```

**IK - Set interrupt key**

The key used to interrupt tracing may be redefined from its default of CTRL. The format of the command is as follows:

```
IK <byte value>
```

<byte value> corresponds to the internal value of the key as defined in the section INKEY in the BBC BASIC User Guide. It must be given in the hexadecimal form. For example, to use the '@' key to interrupt TRACE:

```
IK &B8
```

When TRACE is interpreting or printing store, the interrupt key will be checked before the start of each line of output. During tracing, it will be checked at most once every 256 instructions. When the key is found to be depressed, a return will be made to command level (the '+' prompt).

**EN - Set entry point**

This command is used to define the starting point for tracing. Its format is as follows:

```
EN <address>
```

EN does not cause tracing to start; this operation is performed by the CO command.

**CO - Continue tracing**

The CO command continues tracing from the current address; it is also used to start tracing, once the start point is defined using the 'EN' command. The format of the command is as follows:

```
CO [<optional reporting level>]
```

If a current address has not been defined, the command will fail and an error message will be given.

The reporting level, if present, will be made the current reporting level; otherwise, the global reporting level will be used.

**SS - Snapshot or Single Step**

This instruction displays the full status at the start of the current instruction, or single steps through a given number of instructions. The format is:

```
SS [<parameter>]
```

If the parameter is not supplied, a 'snapshot' of the current processor status is printed, but the instruction is not executed. If the parameter is supplied, that number of instructions are executed at full reporting level, followed by the snapshot of the next instruction to be executed.

### 4.4.3 Looking at memory

This group of commands includes the following:

IT Interpret (disassemble) store  
PT Print store

#### *IT - Interpret store*

This command interprets (disassembles) store from the given address; it displays the addresses and opcodes in the following way:

31190F 91 70	STAIY &70
311911 D1 70	CMPIY &70
311913 EA	NOP
311914 D0 23	BNE &23

When interpreting immediate instructions (eg LDAIM ">") TRACE uses the current values in the M and X status flags to decide how many bytes of operand to take.

Standard MASM mnemonics are used. The format of the command is as follows:

IT <address> [<end address>] +<length>

Store will be disassembled from <address> and the process will stop at <end address> or <address> plus <length>. If the second parameter is not present, disassembly will stop at <address>+&20. To produce the above example, the following command would have been used:

IT &31190F &311914

The address will be interpreted as a displacement in the current direct page if its value is less than &100. For example:

IT &20

will disassemble bytes from address D+&20, where D is the contents of the direct page register. To specify an absolute address in the range &00-&FF, at least one leading zero must be included, as in:

IT &020

#### *PT - Print store*

This command prints store from the given address; it displays the contents in the following way:

310070 00 1A 00 00 00 00 36 00
310078 00 00 36 00 00 00 36 00
310080 02 FF 52 DF FF FF FF FF

The format of the command is as follows:

PT <address> [<end address>] +<length>

Store will be printed from <address> to <end address> or <address> plus <length>. If the second parameter is not present, printing will stop at <address>+&20. To produce the above example, the following command would have been used:

PT &310070 &310087

or

PT &310070 + &17

The address will be interpreted as a displacement in the current direct page if its value is less than &100. For example:

PT &7F

will print bytes from address D+&7F, where D is the contents of the direct page register. To specify an absolute address in the range &00-&FF, at least one leading zero must be included, as in:

PT &07F

#### 4.4.4 Patching memory and registers

This group of commands includes the following:

PS	Patch store location
PR	Patch register
RM	Reset M bit in P register
SM	Set M bit in P register
RX	Reset X in P register
SX	Set X bit in P register

Each of these commands is described below.

##### PS - Patch store

This command modifies data and instructions in the computer's memory. Its format is as follows:

PS <address>

If <address> is less than &100 then the contents of the D register will be added to give an address in the direct page. For example, if D=&0200 then:

PS &FD

will patch address &2FD. If the address &00FD is required, then a leading zero must be included:

PS &0FD

the contents of <address> will be displayed and can then be modified. The command is interactive, in that the user can modify a number of locations in succession with the minimum amount of effort. If the user types '+' after a byte, or in place of a byte, the contents of the next address will be displayed and can be modified also. Similary, if the user types '-', the contents of the previous location will be displayed and can be modified.

To return to command level, the RETURN or ESCAPE key is pressed.

As an example, the following dialogue will change locations &311911 and &311913 to &D1 and &EA, respectively:

```
+PS &1911
311911 D9 D1+
311912 70 +
311913 00 EA <RETURN>
```

Here, the underlined items would be typed in by the user. Any characters except hexadecimal digits, + and - and RETURN are ignored.

##### PR - Patch register

This command allows modification of the contents of the H, A, C, X, Y, D, B, P or S registers; only one of these can be modified with a single command. The format of the command is as follows:

PR <regname> <value>

For example, the following commands will alter the X and A registers to &70 and &FF:

```
PR X &70
PR A &FF
```

## *Chapter 4*

### ***RM - Reset M bit in P register***

This clears the M bit in TRACE's copy of the P register, which means that subsequent instructions will be traced in 16-bit memory/accumulator mode. Note that to patch the high byte of the accumulator, the name H is used, and to affect both bytes the name C is used.

### ***SM - Set M bit in P register***

This sets the M bit in TRACE's copy of the P register, which means that subsequent instructions will be traced in 8-bit memory/accumulator mode.

### ***RX - Reset the X bit in the P register***

This clears the X bit in TRACE's copy of the P register, which means that subsequent instructions will be traced in 16-bit index register mode. In addition, patching the X or Y registers will result in both bytes being affected.

### ***SX - Set the X bit in the P register***

This sets the X bit in TRACE's copy of the P register, which means that subsequent instructions will be traced in 8-bit index register mode. In addition, patching the X or Y registers will only affect the lower byte of the registers.

#### **4.4.5 Memory protection**

This group of commands includes the following:

SP Store protect  
DP Display store protections  
SA Store allow (unprotect)

Each of these commands is described below.

##### ***SP - Store protect***

If a memory location in a program is being corrupted for some non-obvious reason, the instruction doing the corrupting may be tracked down using the SP command. The format of the command is:

**SP <address> <value>**

Suppose, for example, that the location &311900 should contain the value '&55', but it is being changed somewhere in the program. To find the offending instruction, this would be used:

**SP &311900 &55**

Each time a store-modifying instruction is obeyed, TRACE would check if it is altering address &311900. If it is, a message similar to the following will be displayed and you can inspect the code that is causing the problem:

Protection failure at 1900 was 88 should be 55

##### ***DP - Display protections***

This command gives a list of protected locations, in descending order of address. The output of the command looks something like the following:

31191D D0  
311918 91  
31190C 48

##### ***SA - Store allow***

This command removes a given location from the table of protected addresses. Its format is as follows:

**SA <address>**

For example, the following command would remove &311900 from the protection table:

**SA &311900**

#### 4.4.6 Real-time tracing

Single-stepping through time-critical subroutines to find errors will cause complications. The commands described below are intended to help in this situation. They include the following:

RS	Set real-time point
RC	Clear real-time point
CR	Set real-time Chapter
CC	Clear real-time Chapter
DR	Display real-time points
RN	Enable real-time mode
RF	Disable real-time mode

##### *RS - Set real-time point*

This command has the following format:

RS <address>

<address> will be added to the table of real-time addresses. Whenever a JSL instruction is to be executed, its address will be checked against those in the table. If it is found, then instead of simulating the JSR instruction, the call will be made from within TRACE and the subroutine will run at real time speed.

##### *RC - Clear real-time point*

This command removes an address from the real-time table. It has the following format:

RC <address>

##### *CR - Set real-time Chapter*

This takes a Chapter number in the range &00-&FF. All JSL instructions to that Chapter will be executed instead of traced. Example:

CS &04

By default, routines in Chapters &00 and &FF are real-timed.

##### *CC - Clear real-time Chapter*

This clears the Chapter given from the real-time Chapter table. After a CC all JSLs to the specified Chapter are traced.

#### ***DR - Display real-time points***

This command gives a list of real-time points, with their expected values, in descending order of addresses. For example:

```
311950 01  
311102 FF  
311100 81
```

#### ***RN - Enable real-time mode***

Apart from the real-time routines and Chapters described above, TRACE can also be made to real-time all instructions. Following the command RN, all instructions will be executed directly instead of by interpretation. This will continue until a break point whose count is zero is met.

While real-time is active, TRACE will not enforce any stack or location checking, or break-in key checking. A typical example of real-timing a routine is:

```
+RN  
+EN &20200  
+CO
```

The continue will call the real-time routine, which will execute until the break point is encountered. Real-time mode continues until disabled by RF.

#### ***RF - Disable real-time mode***

After an RF command, only those routines and Chapters explicitly set to be real-timed will be executed directly. All other will be interpreted with any necessary checking being performed.

### **4.4.7 Miscellaneous**

#### ***TO - Restrict tracing of operation codes***

This command restricts the class of operation codes which will be reported. It has the following format:

**TO <integer>**

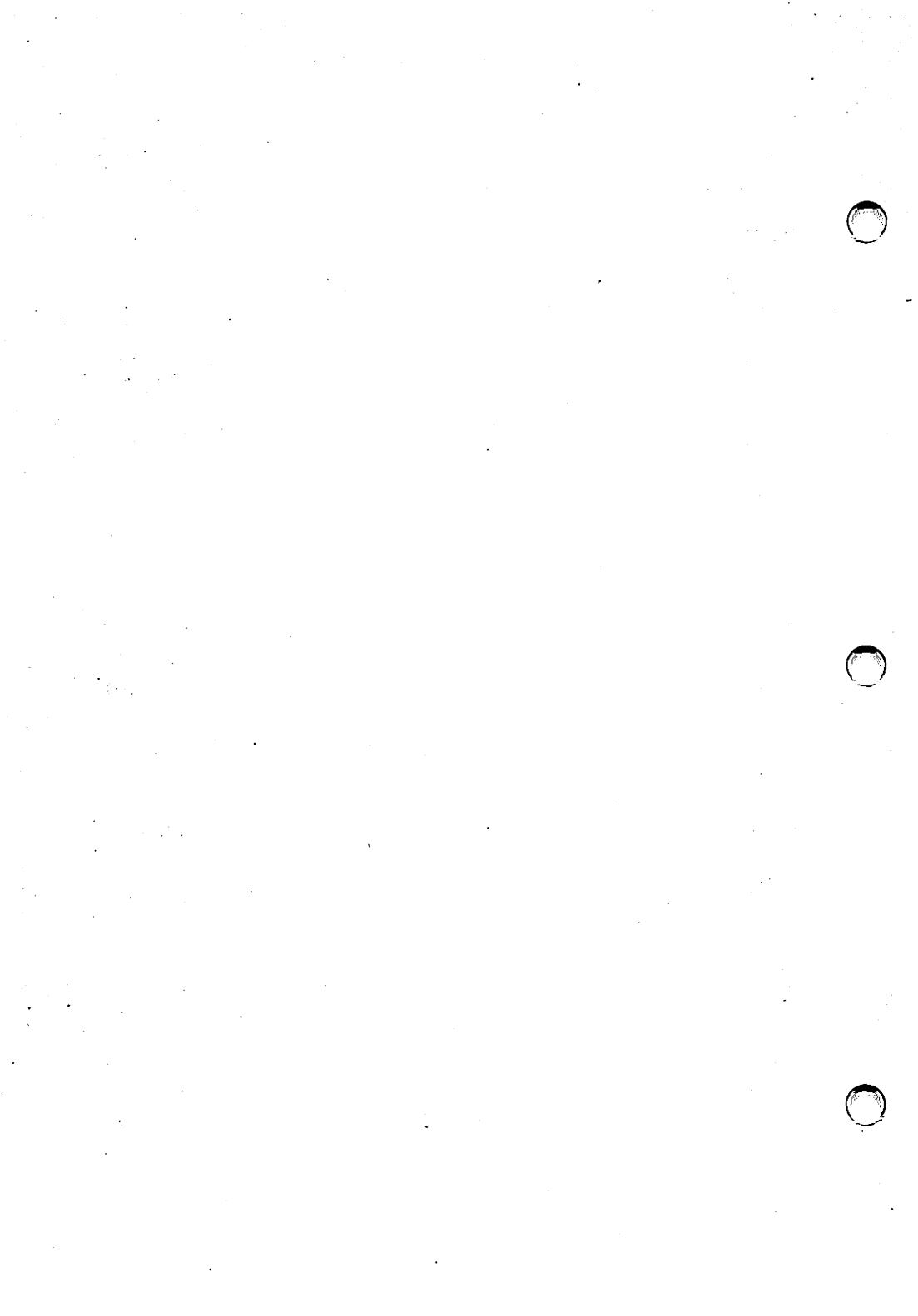
the contents of <integer> have the following significance:

**<bit> Class of operation codes reported**

- |   |                         |
|---|-------------------------|
| 0 | Control codes           |
| 1 | Loops                   |
| 2 | Stacks and tests        |
| 3 | Arithmetic and the rest |

Thus,

TO &F traces all instructions TO &3 traces loops and control codes only TO &8 traces arithmetic instructions.



## 5. Support Utilities

### 5.1 Introduction

This Chapter describes the software tools available to aid in the construction of Communicator software modules.

These utilities are either BASIC programs designed to run on the ARM second processor under 65TURBO, or TWIN command files.

The following utilities are covered:

<b>TCRUNCH -</b>	BASIC program compressor
<b>UNCOMMENT -</b>	Strips comments from a BASIC program
<b>EDCONVERT -</b>	Converts a BASIC constant definitions file into a DEMANIFEST TWIN command file.
<b>MAKEMOD -</b>	Turns a BASIC program into a module
<b>ALOADER -</b>	Generic loader. Creates a single loadable object ROM image from a set of assembler object files.

More information on the use of these utilities is provided in the Introduction to Communicator Programming.

### 5.2 UNCOMMENT

The UNCOMMENT utility, as its name suggests, is used to strip comments from BASIC source files, (i.e. before they have been exec'd into BASIC).

UNCOMMENT is a twin command file and is invoked on the current file from TWIN thus:

**<F1> UNCOMMENT**

Comments starting with "REM" are removed, as are those starting with "I\*". Lines consisting entirely of asterisks ("\*\*") are also stripped.

UNCOMMENT does not remove OS comments with a space between the asterisk ("\*") and the bar ("I").

### 5.3 ED CONVERT

EDCONVERT is a utility which creates DEMANIFEST files from BASIC constant definition files.

EDCONVERT is a TWIN command file and should be invoked on the current file from TWIN thus:

**<F1> ED CONVERT**

All symbolic constants used in a BASIC program should be defined in a single file, one constant per line. The first line of the file should be blank and the second line should contain the title of the file.

EDCONVERT creates a new TWIN command file from the constant definitions file which should be saved as "DEMANIFEST" in the programs's source directory.

The DEMANIFEST file can then be used to substitute actual values for manifest constants within a BASIC source file.

## 5.4 TCRUNCH

The TCRUNCH utility is used to compress BASIC programs.

TCRUNCH is a BASIC program and runs on either the ARM second processor under 65TURBO.

TCRUNCH is invoked thus:

**CRUNCH filename1 filename2**

where

filename1 is the input file

filename2 is the output file

All BASIC sources for a program must be exec'd into BASIC before running it through TCRUNCH.

TCRUNCH will not accept input greater than 64K bytes in size.

Programmers should avoid the use of short variable names and using variable names with the EVAL statement.

## 5.5 MAKEMOD

The MAKEMOD utility adds a module header to the start of a BASIC program and a module checksum to its end.

MAKEMOD is a BASIC utility and runs on either the ARM second processor under 65TURBO or on a Communicator.

MAKEMOD requires that a "build\_data" file be present in the program's source directory along with a working directory "MODULES".

The build\_data file contains information in the following format:

```
> build_data for <module name>
<source directory>
<destination directory>
<flags>
<module name>
<file name>
```

any other text - comments etc.

The maximum file size is 512 bytes. Note that the module name does not have to be the same as the file name (with or without its /A). Either of the source directory or the destination directory may be null, but not both. It is recommended that the source directory is null (ie use the current directory) and the destination directory is Modules.

The flags are in the form : (\$MHCPOS% + (\$MHCBNK% + ... which is EVALed.

The program makes decisions about the contents of the header on the basis of the flags specified and the module name (in the case of Menu and certain other programs designed for use in "Space/Break" systems). There are four cases handled by \*MAKEMOD:

(1) For most programs, designed to run directly below the menu, set the following flags:

(\$MHCPOS% + (\$MHCBNK% + (\$MHCBS% + (\$MHCMEN%

(2) For a Menu module (not necessarily the main Menu program), the module name must be "Menu" (case as shown) and the flags :

**(\$MHCPOS% + (\$MHCBNK%**

This ensures a result compatible with the old MAKEMODM.

(3) For BASIC programs to run deeper in the structure than Menu-visible ones :

**(\$MHCPOS% + (\$MHCBNK%**

(4) For service programs that are not to appear on the menu, but which can be invoked by \*<modulename> in a space/break system, whereupon they will take over the machine :

**(\$MHCPOS% + (\$MHCBNK%**

modulename must be preceded by an asterisk (eg. \*SERVICE).

For more information on MAKEMOD and an example build\_data file see the Introduction to Communicator Programming.

## 5.6 ALOADER

The ALOADER utility is a generic object file loader. ALOADER creates an executable module object file from a set of assembler object files.

ALOADER is a BASIC utility and runs on the ARM second processor under 65TURBO.

ALOADER builds an image file from a set of objects specified by a "build\_data" file within a program's working directory.

A build\_data file takes the following format:

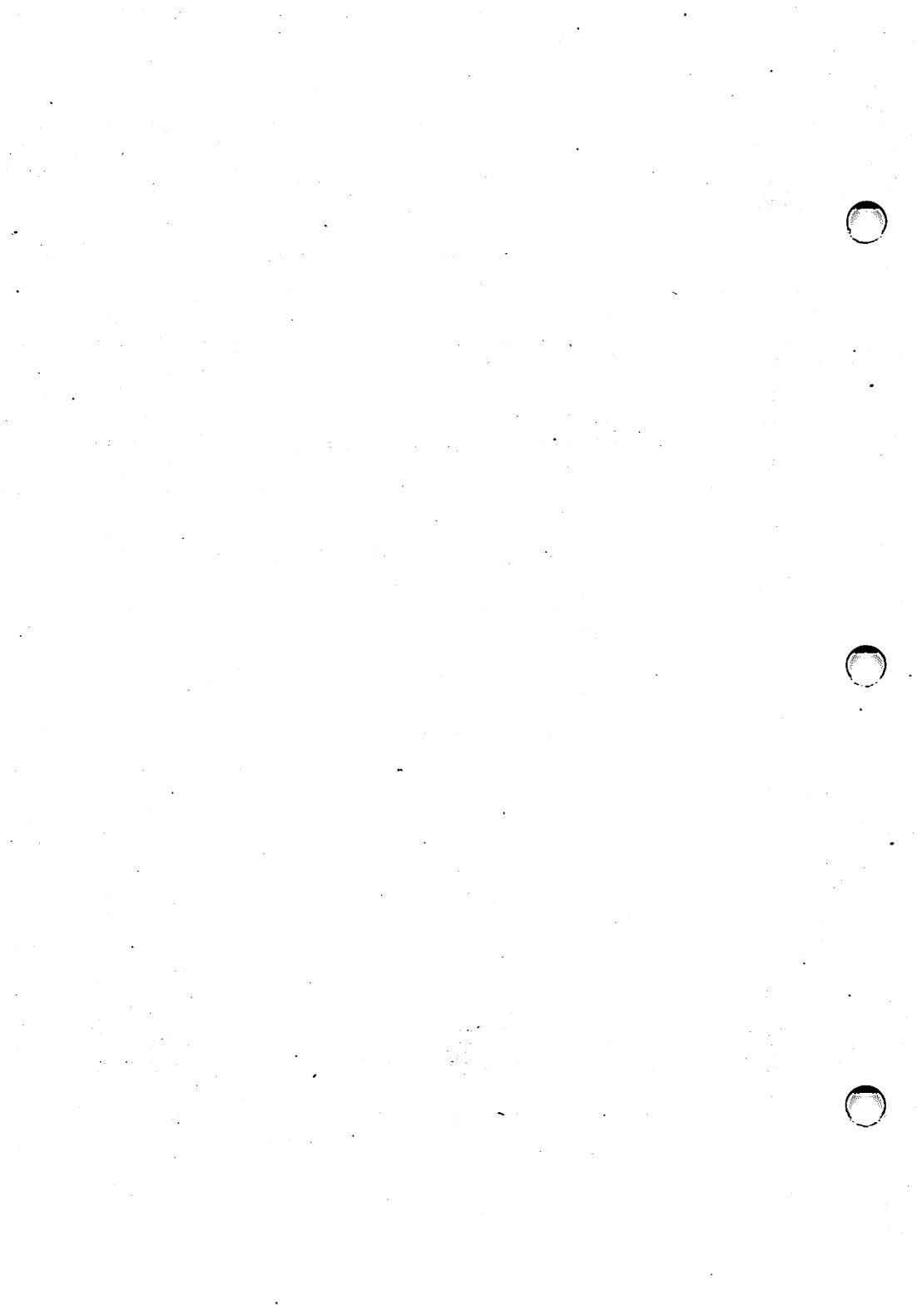
```
> build_data
<directory where X.objects can be found - optional>
<object 1>
<object 2>
.....
<object n>
* *****
<BANK address to be saved with image - optional>
<final object name>
```

User defined fields are bracketed by "<>". Fixed data is in bold.

A build file consists of the file name followed by a blank line. The build file name is followed by the name of a directory where the object files are to be found. This can be left blank and the directory will default to "X" in the current directory. There then follows a list of object files each on a separate line. The object files should be given in assembly order. The object file list is terminated by a line starting with a star (\*) character. Following the object file list is an optional bank load address for the image file. If left blank the bank load address defaults to that of the first object file. Finally, the last line specifies the name of the image file.

Every entry in the table can be followed by a comment, but make sure they are separated by a space!

For an example build\_data file see the Introduction to Communicator Programming.



## 6. BlueScreen Software

### 6.1 Overview

The BlueScreen utilities are a set of BASIC procedures and functions which are provided for incorporation in programs. They provide a consistent user interface and display style for BlueScreen software. This includes most of the main application programs which are unique to the Communicator, such as the Menu, Phone Directory and Carousel. These applications are characterised by a blue and white display screen in which similar information appears in the same place for each program. The standard utilities save the user from needing to write his own routines to achieve this display standard, and ensure consistency between each piece of software. The routines fall into 3 main areas :

Declaration support

Screen formatting - including display of prompts and error messages

HELP support

The Declaration support package must be called as early as possible in the user's program to ensure that data structures used by the rest of the utilities are declared. The package also calls a number of procedures which must be provided by the user in order to supply HELP information, error messages, prompts etc.

The Screen formatting support includes procedures to draw boxes, turn inverse video on and off, display the time on-screen, display a module title etc. and also higher-level functions such as drawing a basic BlueScreen layout, displaying active Fkeys. In general, screen support routines must not be called before the declarations are completed.

The HELP support consists of a procedure PROC@HELP which may be run by the system if the user presses HELP. It uses the screen formatting routines, and relies on the declarations being completed. In order to avoid problems if the user presses HELP before the declarations are complete, however, it checks to make sure and does nothing if they are not finished.

## 6.2 Specification

### 6.2.1 Screen format

The BlueScreen utilities support a mode 0 screen with Blue background and white foreground with the following format :

```
Communicator Module version (c) Acorn Computers Ltd 1986 Day nn Mon 19nn hh:mm
-----
|  

|  

| < Display box : 78 characters by 20 lines >  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

|  

| < prompt box : 78 chars by 3 lines >  

|  

|  

|  

| F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 |
|Caption1|Caption2|Caption3|Caption4|Caption5|Caption6|Caption7|Caption8|
```

The top line of this screen (containing title information and a continuously updated time and date readout) is displayed in inverse video (blue on a white background) as is the function key display at the bottom of the screen. The remainder of the screen is set up in normal video (white on blue ground).

The main user routines are intended to support a screen exactly as above, but lower level routines also exist which enable the user to customise the display in cases where the standard format is not flexible enough. For example, the Menu layout is different in the display box and title line, while the Calculator has extra boxes within the display box.

### 6.2.2 Help support

Any BASIC program module running below Menu level has the opportunity to provide useful help information to a user who presses the HELP key. The interface to this facility is through a procedure called PROC@HELP. If the BASIC program is running, and has such a procedure, it will be run by a subroutine call from the menu program. BASIC starts a separate coroutine in which to run the HELP process, and then provides it with a copy of the user's program environment. Note that the help process may change the

user's program environment (for example by creating new variables).

The BlueScreen utilities provide a standard PROC@HELP which handles all screen formatting and keyboard transactions while the help process is active. Errors or preempts during the help process can be particularly dangerous, and the use of the standard help utilities is recommended to avoid these possible problems. Note that the use of task keys within the standard help procedure is supported, and the effect is the conceptually obvious one.

Help is provided on a standard screen layout as defined in the preceding section, and is divided into "Contexts" and "Pages", which are defined by a user supplied procedure PROCdefine\_help\_info. Each activity within the user's program may be associated with a context, and there may be one or more pages of help information associated with each context.

## 6.3 User interface

### 6.3.1 BlueScreen

In this context, "User" means a programmer writing a piece of BlueScreen software in BASIC. Unlike a module with a single entry point and a number of entry reason codes, the BlueScreen utilities are incorporated in the user's program and form part of its structure. This means that there are a lot of individual interfaces to be described. These fall into two classes :

#### Required interfaces

- every program needs to call certain routines and provide certain other routines before the utilities can be used. These interfaces will be described in this section.

#### Invocation interfaces

- The detailed interface to each routine in the package that the user may wish to call. The present section will merely list those routines which the programmer may wish to call directly, with a brief note on the purpose of each. The interfaces to other routines are described in the 'Procedures' section below.

### 6.3.2 Required interfaces

These fall into two classes: calls the user must make to start the package, and procedures that the user must provide for the package to operate.

The first statement in the user's program should be :

#### PROCdeclarations

This must be called just once, at the start of a program. Causes the package to declare all its internal variables and arrays. It will also call a number of routines which must be provided by the user. These are :

PROClocal\_declarations  
PROCdefine\_help  
PROCdefine\_prompt\_info  
PROCdefine\_error\_messages  
PROCdefine\_fkey\_captions

***PROClocal\_declarations***

The purpose of **PROClocal declarations** is to allow the user to declare anything he wants - which may be very little. Typical uses for this routine include setting up arrays or strings, making FX calls to change the program's environment, or assembling some machine code. One required use, however, is to declare a string title\$ which contains the title and version number to be displayed on the basic BlueScreen. There is a maximum length for this string of sixteen characters. A typical title\$ declaration :

```
title$="New Module V1.05"
```

***PROCdefine\_help***

**PROCdefine\_help** is required to define the help information which will be read by PROC@HELP. This must comprise at least one page of information (even if the page only contains a null line). The information is accessed by a two-dimensional array of labels, which must be executed in order to be set. This array is help\_info%(x,y). The x dimension specifies the number of contexts for help information, while the y dimension specifies the maximum number of pages in any context (note that the indexing origin for BASIC arrays is zero). Since not all contexts will have the same number of pages of information, each context also needs an array of page counts maximum\_help\_page%(x), each element of which must be assigned a value less than or equal to the y above. Each page of help information is defined by DATA statements, terminated by the string consisting of exactly four asterisks. For example :

```
DIM help_info%(2,1)      :REM three contexts of maximum 2 pages
...
maximum_help_page%(2)=1  :REM context 2 has two pages
...
^help_info%(2,1)
DATA"This is the help information for the second"
DATA"page of the third context of this program"
DATA"*****"
```

The set of help information which is displayed at any time when the user presses HELP is defined by a global variable display\_help\_context%. No checking is currently carried out in PROC@HELP to ensure that this is in the permitted range, so if the user sets it to a value outside those for which help\_info% is declared, his program will fail with a BASIC "Subscript" error. PROC@HELP always displays page 0 of the context first. Again, no checking is carried out to ensure that the user has, in fact, declared all the help pages he might use, and it is particularly important that these declarations should have been completed since a BRK in PROC@HELP can leave the main program's heap corrupt.

***PROCdefine\_prompt\_info***

**PROCdefine\_prompt\_info** fulfils a similar function of declaring textual data, this time to be used in the prompt box. The labels are in a one-dimensional array prompt\_data%. The data for each prompt context is contained in three strings, each of 74 or less characters. For example:

```
DIM prompt_data%(5)      :REM six contexts for prompts
...
^prompt_data%(2)
DATA"To see PROC@HELP in operation      Press HELP"
DATA"To return to the Menu      Press STOP"
DATA"To simulate a BBC micro BREAK      Turn the machine off"
^prompt_data%(3)
..."
```

The prompt which is displayed at any time is controlled by a global variable `display_info_context%`. No checking is carried out to ensure that this value indexes a label which has actually been declared. Note that after setting the global variable, `PROCprompt` must be explicitly called to display the prompt. This variable also controls the prompt which is restored when a "Print failed because..." error message is cleared. For this reason, it is very important that prompts should be displayed this way, and not using `PROCinfo` directly (See the interface details for these routines under 'Procedures').

#### *PROCdefine\_error\_message*

`PROCdefine_error_messages` is very similar in format to `PROCdefine_prompt_info`, except that as well as the three strings, there is a numerical datum which specifies the highlight state to be used to display the message. This is normally set to 1 to indicate inverse video. The labels used are in a one-dimensional array `error_data%`. There is a restriction on the length of the third string in that when a string `press_space$` is concatenated, the total length must be within the 74 character limit. `press_space$` is normally declared with a lot of leading spaces, but if the user's error messages are particularly verbose he may prefer to declare it with none. It is for this reason that declaration of this string is left to the user, and should be declared in this procedure. An example:

```
press_space$="          press SPACE to continue."
DIM error_data%(21)   :REM lots of possible errors
...
^error_data%(13)
DATA 1, "You were unlucky"
DATA "You shouldn't press Fkeys like that"
DATA ""               :REM leave room for press_space$
```

#### *PROC\_define\_fkey\_captions*

`PROCdefine_fkey_captions` is required to declare an array of short strings which specify the words to appear on the Fkey box display. These are the default values when the program is started, and there is nothing to stop the user from changing the captions later, if required. For example:

```
DIMcaption$(7)
:
caption$(0)=" Go Home"
caption$(1)="Stiff mc"
caption$(2)=" Useful"
...
...
```

### 6.3.3 Invocation interfaces

The main routines which the programmer may find useful are now described.

#### *PROCdraw\_base\_screen*

`PROCdraw_base_screen` takes no parameters. It initialises the screen into the standard blue/white format with a title, the time at the top right, a large display box and a smaller prompt box. This procedure does not set up the fkey display. It does not leave any sort of window set up, so `TAB(x,y)` will refer to the whole screen.

## *Chapter 6*

### *PROCdisplay\_fkeys*

*PROCdisplay\_fkeys* takes a single parameter which is an integer. This is interpreted as an array of nine bits. The most significant is set to indicate that the boxes should be drawn. It is not currently useful to clear this bit. The low eight bits control the display of the eight fkeys, bit (n) controlling fkey (n+1). It is usual to have a series of manifests set up with these bit values to make the program easier to understand, for example @box% = 256 @list% = 1 @sec/edit% = 2, ... @all% = 511. The current setting of these bits is stored in a global called *current\_options%*, so that an fkey may be forced on by :

```
PROCdisplay_fkeys(current_options% OR @print%)
```

or turned off by :

```
PROCdisplay_fkeys(current_options% AND NOT @print%)
```

Any combination of fkeys may be turned on or off simultaneously by these explicit calls. In addition, it is often useful to have something like

```
saved_options% = current_options%
... do something complex ...
PROCdisplay_fkeys(saved_options%)
```

Finally, if you want to turn one or more fkeys off, without turning any others on, a shorthand method is provided :

```
PROCdisplay_fkeys(NOT (@print% OR @quit%))
```

And if you want to get particularly complicated, and save screens away in memory within particular procedures, then :

```
... save your screen ...
LOCAL current_options%
PROCdisplay_fkeys(
any_old_options%
)
... restore your screen ...
ENDPROC      :REM this restores the old value of current_options%, which will
               REM therefore match up with the screen image.
```

NOTE: an fkey has in fact three states: on, off and undefined. The latter state applies if the caption associated with an fkey is the null string. In this case, it is meaningless to turn the fkey display on or off: it will always be displayed as an empty box.

### *PROCprompt*

*PROCprompt* takes no parameters. This procedure exists to display three lines of up to 74 characters of text in the prompt box. The text to be displayed is selected by the global variable *display\_info\_context%* which should be assigned to as needed by the programmer. The reason for doing it this way is that any procedure which needs to display a prompt temporarily can be structured with a call to an inner procedure with:

```
LOCAL display_info_context%
display_info_context% = @some_temporary_value%
PROCprompt
...
ENDPROC
```

The caller can then call PROCprompt without needing to know which context it was itself called from. This is obviously important in the case of PROCs which may be called from almost anywhere, eg. PROCscreen\_dump, PROC@HELP.

The resulting display is in normal video.

#### *PROCwarning*

This procedure exists to display an error message in the prompt box. It takes a single positive integer argument which indexes the array or .error\_data% labels declared in PROCdefine\_error\_messages. Each label points to DATA for four items, the first is numeric and is a parameter for PROChighlight. By convention this is 1, indicating inverse video for error messages. A zero here would give normal video. There follow three text strings. The first two may be up to 74 characters long, while the third must be shorter, since the string press\_space\$ will be appended to the end. This is normally set up to be "Press SPACE to continue", and if this is to be central on the third line, the third line of text needs to be padded out to a fixed length of 25 characters. In practice, some programs will have only short error messages, and in this case it will be easier to declare press\_space\$ to have the leading spaces included, and have the third line of error text in the DATA statements as the null string. This is why it is left up to the programmer to declare his own press\_space\$ in PROClocal\_declarations. After displaying the error message, the user must press space before this procedure ENDPROC back to its caller.

```
*****
***** NB: At present, PROCwarning does not restore the contents of ****
***** the prompt box on exit, after the user has pressed SPACE. ****
***** This is because it was thought that after an error, the old ****
***** prompt information would probably be invalid anyway, so the ****
***** programmer would replace it with something else. ****
*****
*****
```

#### *PROCinfo*

This is the routine used by PROCprompt, PROCsysprompt and PROCwarning to display their text. It may sometimes prove useful for the programmer to use it directly. However, it should be noted that if information is displayed this way then a call to PROCscreen\_dump is made, and the print fails, the information restored in the prompt box is determined by display\_info\_context%, which may not be what the programmer wanted or expected. It is recommended that PROCinfo is only used directly when there is no chance of the prompt box being corrupted, and that PROCprompt is used in the majority of instances.

It is called with one integer and three string parameters. The integer selects the highlight level. Zero selects normal video, and at present, anything else selects inverse. It is suggested that the actual values used are zero and one, so that any future enhancements may use higher bits of the highlight level parameter. The three string parameters are each limited to 74 characters, but no checking is carried out by the routine itself.

***PROChighlight***

This routine takes a single integer parameter which defines the highlight level to be used. Note that the comments made about highlight levels, under PROCinfo, apply here too.

Care should be used with this routine if used with TAB(x,y), since if you call:

`PROChighlight(1):P.TAB(X%,Y%);"a string"`

then part of the line BEFORE "a string" may be blocked in in inverse video, depending on where the cursor was previously. The recommended approach is:

`P.TAB(X%,Y%);:PROChighlight(1):P;:"a string";:PROChighlight(0)`

***PROCscreen\_dump***

***PROCscreen\_dump***

This is the standard interface to TXTDMP and SCRDMMP. The routine takes a single boolean parameter: if this is TRUE it will attempt a graphics dump, and if FALSE, a text dump. Note that if the attempt fails, the routine will try to tell the user why: in doing so, it will corrupt the prompt box. On exit it will attempt to correct the damage by calling PROCprompt, so display\_info\_context% should always be valid when the routine is called.

***FNinkey***

FNinkey is provided to get characters from the keyboard, while maintaining a realtime clock display in the top right of the screen. It assumes that the screen is in mode 0 or 3, which it will be for programs using these routines. There are three parameters, the first being the time in centiseconds between clock updates. This is conventionally set to 100 for an interval of 1 second. The second and third parameters specify the X, Y position where the cursor will appear if it is currently displayed. These parameters are conventionally set to zero if you aren't using the cursor at all.

Called with a non-negative update time, the routine will first display the time after the first failure to get a key from the keyboard (ie the time will typically appear after a one second delay from the routine being called - and not at all if the user types soon enough and fast enough), and will not return at all until a key has been pressed. A special form of the call, with the first parameter set to -1 forces the time to be displayed, and returns immediately.

***PROCclear\_buffer***

This call exists to replace OSCLI"FX15,1" which at one time didn't work. Its sole purpose is to clear the keyboard buffer of any extraneous guff, bumpf, stray characters, noise, preempts, etc. etc. that may have found its way in there due to bugs in the user or the system. Please note that if you are in a "No-preempts" routine such as PROC@HELP, IT MUST NOT BE USED, since it is eminently preemptible (it uses INKEY); use OSCLI"FX 15 1" instead.

## 6.4 Environment

The utilities package is provided in source and compressed source form to be included in the user's program. This should then be \*TCRUNCHED and MAKEMODded. The resulting module will be run as an interpreted BASIC program (usually living in its own coroutine). Such a module will typically support basereason and be permitted to appear on the main menu. (See the Communicator Systems Manual for a discussion on coroutines, task switching and the purpose of the basereason entry code.) It is perfectly possible, however, for a module which runs as a child coroutine of another module (ie not the Menu) to use the utilities (for example, Tshell issues a module: it cannot appear on the menu and is a child coroutine of the Phone module). In particular, the utilities are designed to support a module which may have child routines which themselves use the utilities and support PROC@HELP, etc.

Much of the code in the standard utilities is also used by the Menu module itself, but because of the unique requirements of the Menu, it uses a non-standard version. This version is documented separately, but refers to this document for most of its detailed information.

## 6.5 Structure

Because the utilities are mainly low-level service routines designed to be incorporated into the user's program, the structure is rather fragmentary. This section describes the assumed structure of some generalised program that might use the utilities, and it is hoped that this will usefully illuminate the relationships between the various utility routines. Only a small number of the routines are intended to be called by the user directly, while others form the common building blocks used by the main procedures seen by the user's own code. There is nothing to stop the user from calling the lower level routines himself, but he should be aware that by doing so he may undermine the assumptions made by some of the higher level routines, with potentially undesirable effects.

The utility routines are designed to be used by a program with a standardised structure as follows :

```
PROCdeclarations  
body of main program calling various utility procedures  
END
```

## 6.6 Data structures

### 6.6.1 Textual data structures

Three sections of the utilities use potentially large amounts of textual data. These are: the HELP system; the error message system and the prompt system. All three use similar, though not identical, data structures. In all cases, the textual data is stored in DATA statements, referenced by labels which can be RESTORED. To make it easier to reference these data, the labels declared are one- or two-dimensional arrays. This raises a small implementation difficulty (details below) which means that all of these textual data must be executed, i.e. they must lie inside a declarative procedure which the BASIC interpreter scans. These three procedures must be provided by the user, based on examples in the `utils2` file (see below under Source Code Management). For the prompt and error text, these labels are `prompt_data()` and `error_data()` and are one-dimensional arrays.

For the help text, the labels are `help_info()` which form a two-dimensional array, of which the first subscript is a context number and the second subscript controls the page within the context. Because the number of pages may vary from one context to another, each context defines an array element `maximum_help_page()` to ensure that the help system does not try to display non-existent frames.

In the case of the prompt and error data, there are always a fixed number of items, viz. three strings for prompts and a single integer followed by three strings for the errors. The HELP information is a little more flexible, since there may be many or few lines of information on a help page. Here there is an arbitrary number of strings, termination of a page being indicated by a string containing just four asterisks. It should be noted that a maximum of twenty lines may be displayed on a page, but no check is performed, so displaying extra lines will merely scroll the top lines out of the display box.

#### *Variables controlling display*

The HELP system is called asynchronously by action of the top level Menu (see Communicator Systems Manual), and which help information it displays is controlled by a (global) variable `display_help_context%`. This is used as the first subscript of the `help_info%` array.

The second subscript is generated internally by the help system, starting at zero and rising to `maximum_help_page%(display_help_context%)`, before looping round to the first page.

The prompt display is controlled by a (global) variable `display_info_context%` which is used as a subscript to the `prompt_data(%)` array of labels. For user-defined contexts, the value must be a non-negative integer (up to the maximum declared size of the `prompt_data()` array), but assigning a value of -1 will cause the system to use a "blank" context to fill the prompt box. Values from -2 to -4 are used.

## 6.7 Procedures

The BASIC utilities package contains the routines listed below. For each routine, the line "file" describes which part of the package contains the routine. "util1" is the standard package to incorporate into applications. "util2" routines are those presented in the 'skeleton' which the user must write himself to declare textual data as described under "User interface".

### 6.7.1 Util1 routines

#### ***PROC@HELP***

##### **PROC@HELP**

File :	util1
Called by:	Called externally and asynchronously by parent coroutine
Calls:	draw_base_screen, highlight, init_fkeys, display_fkeys, fkey_boxes, fkey_caption, display_help, ex_key_char, screen_dump
Purpose:	Entered asynchronously if user presses <HELP>. Screen and VDU context saved for us by MENU, and restored on exit.
Globals:	declarations_finished% - to check they are child_running% - to see if one is (and can provide its own help) help_fkeys_valid% - to see if they are
	No more used explicitly in @HELP, but see display_help()
Exit conditions:	Changes nothing global permanently. Screen should be restored to former state by main menu.
Description:	By assigning declarations_finished% variable to itself we declare it if it didn't exist, thus avoiding a BRK when we test it, but unless it was set up by PROCdeclarations, it will be initialised FALSE, so @HELP exits immediately.  Check child_running%. If user has not explicitly set this to a non-zero value, then it will be found FALSE, and we will provide our own help. If it is non-zero, however, we will call PROCchild1s_help which will interpret the value as a handle for a lower level coroutine which will provide help for itself.  If we are providing our own help, we need a base screen to work on, so draw one, set inverse video. Unless user has explicitly set help_fkeys_valid% to be FALSE, we will assume that current_options% and caption\$() are valid, and display appropriate fkey captions. If this global is FALSE, then we ignore current_options% and display only F7 (print), which is always valid in help. This option really exists for the benefit of the Menu when it is providing help for a program which has none, because the Fkey display which is appropriate to the menu itself is not appropriate to the help for a child. It could be used by the programmer to ensure that fkey captions are wiped if user presses help at particularly critical points of the code, without needing to define a whole new help context or change the fkey display of the running program. This usage is not, however, recommended.  Once the basic screen is set up, we go on to display page zero of the help information for this context, as defined by display_help_context%. All the paging of help information is handled by PROCdisplay_help (q.v.) and need not concern us here.  The remainder of PROC@HELP is a keyboard polling loop, which is careful to ensure that a preempt is not permitted. This is achieved by looking ahead in the keyboard buffer with FNex_key_char. If the user has pressed return, this is removed from the buffer, whilst if he has pressed a task key, this is left there for the menu to process. In either case, the loop exits and PROC@HELP returns control, which passes back to the main menu. If the user pressed space, then the help page is incremented, and PROCdisplay_help is again called to sort out which page to display. Any other keypress is removed from the buffer and ignored.

## *Chapter 6*

### ***FNalign***

**FNalign(N%)**

**File :** util1  
**Called by:** util\_assemble, USER if he wishes to assemble some machine code  
**Calls:**  
**Purpose:** To dimension some workspace for assembly code, ensuring that it doesn't cross a bank boundary.  
**Description:** DIM P% -1 so as to reserve no bytes but get the address where bytes would be DIMensioned. If this address plus the number of bytes we want would be in a different bank from the address itself, DIMension an area up to the end of the bank. Then return the size of the area we want, which caller uses to DIMension an area guaranteed to be in one bank.

### ***PROCchild�\_help***

**Called by:** @HELP  
**Calls:** Machine code entry point child�\_help%  
**Purpose:** If task has a child coroutine - let it provide its own help.  
**Description:** We pass a handle to the machine code routine, but do so without corrupting A%, X%, Y% etc., since these may be in use elsewhere, and we couldn't (at one time) make them LOCAL because of <HELP in HELP>. Consequently, the handle, from child\_running%, is passed to the machine code by indirection. The machine code has access to the address util\_internal\_data%, from where it loads Y.

### ***PROCclear\_buffer***

**File :** util1  
**Called by:** Warning, USER  
**Calls:**  
**Purpose:** Temporary but vital utility when OSCLI"fx 15 1" didn't work. Retained in current release for compatibility with earlier software. Uses less bytes in the calling sequence (after TCRUNCHing) than the OSCLI call, but is slower.  
**Description:** Keep on doing inkeys until we find there are no more keys to read.

***PROCdeclarations***

File : util1  
Called by: Main (user) routine  
Calls: define\_help, define\_prompt\_info, define\_error\_messages, define\_fkey\_captions, util\_assemble, local\_declarations  
Purpose: Procedure to isolate all global declarations.  
Description: Assumes no pre-existing globals, takes no parameters, this should be called from the main program before anything else is done.

We declare all the variables used internally by the utilities, and call a series of user-defined procedures in which the user is expected to define anything he himself wishes to use, plus those arrays, labels and DATA required by the utilities if they are to provide prompts, error messages and help information. Some of the internal variables are set up by further utility routines define\_system\_prompts and util\_assemble.

***PROCdefine\_system\_prompts***

File : util1  
Called by: Declarations  
Calls:  
Purpose: Defines strings for the first line of the system prompts used by PROCdisplay\_help.  
Description: Prompt values -4 to -1 reserved by system: -1 is a blank prompt, while the first line of each of the prompts -2 to -4 are pointed to by sysprompt\_data%(0 to 2)  
The three strings defined are :  
.sysprompt\_data%(0):DATA ""  
.sysprompt\_data%(1):DATA "Press SPACE for more information."  
.sysprompt\_data%(2):DATA "Press SPACE to return to start of help information."

## *Chapter 6*

***PROCdisplay\_fkeys(display\_fkeys\_internal\_option%)***

File : util  
Called by: USER  
Calls: highlight, init\_fkeys, fkey\_caption  
Purpose: Display currently available fkeys at screen bottom  
Globals: current\_options% defined by PROCinit\_fkeys before first call  
caption\$(0:7) defined through PROCdeclarations  
Parameters: display\_fkeys\_internal\_option% bits 0 to 8 & 31 significant  
calling sequence :  
general call :  
**PROCdisplay\_fkeys (current\_options% OR <new options> AND NOT <old options>)**  
or  
**PROCdisplay\_fkeys(<predefined set of new options>)**  
easy call to turn options off :  
**PROCdisplay\_fkeys( NOT <options> )**

Description: Note: if called with current\_options% bit 8 set, but the display actually clear, it won't draw the boxes for you - use PROCinit\_fkeys, but keep a copy of current\_options with which to call display\_fkeys.

In practice, manifests, such as @restore% = 128, would be set up, and used in the calling sequence (for more information on using manifests, see the BASIC programs Author's Guide). An example of this sort of call :

**PROCdisplay\_fkeys (current\_options% OR @connect% AND NOT @see\_edit%)**

Since the Fkey display is always in inverse video, we start by ensuring this state is set up. If parameter is negative, then user is using the simplified NOT <options> call, so translate this into a positive logic version.

If current\_options% indicates that the boxes are not drawn, and the parameter specifies that it should be, then start off with a PROCinit\_fkeys. This is slow, so the user should always ensure that current\_options% correctly reflects the state of the display.

Use a bit mask, shifted during each iteration, to check up on each fkey in turn. If the state we are specifying is different from the currently displayed state (and not otherwise), call PROCfkey\_caption to change the display. Note that if user has set the caption\$() string to null from something non-null, this call will not erase the present display.

Finally, restore normal video and set current\_options% to reflect the state we have just displayed.

***PROCdisplay\_help(page%)***

File : util1  
Called by: @HELP  
Calls: info  
Purpose: Read and display a page of help information.  
Globals: Assumes that globals have been set up by PROCdefine\_help, called from PROCdeclarations. This covers RESTORE labels and maximum\_help\_page%() for each context. display\_help\_context% is kept up to date by the various routines in the application program, and should be one of the first things they set when the context changes.  
Exit conditions: No text window, highlight level preserved.  
Description: Set up text window for the whole of the display box, and clear it to background colour in normal video (this must be assured by caller).  
Pages of help information are displayed cyclically, ie. if caller has incremented help\_page% beyond the number of pages in the current context, we wrap around to the first page (zero) again. The specific page of information to be used is accessed by RESTOREing the DATA pointer to the appropriate label in PROCdefine\_help\_info. The lines of text are printed preceded and not followed by <cr/lf> to ensure that we can print on the last line of the window. The formatting string is initially null, so that the first line is displayed on the first line of the screen, but is then set to <cr/lf> for all subsequent lines. The page is terminated by a data line containing just four asterisks.  
Once the help is displayed, we must choose a prompt on the basis of the page state. If there is only one page, then the message is pretty minimal, while if there is more than one, we need to know whether the current page is the last. If it is, tell user that pressing space will get him to the first page again. Otherwise, tell him that he can press space to get to the next page. These messages are generated by the sysprompt part of PROCprompt, using reserved values of display\_info\_context% in the range -4 to -2.  
Note that the VDU26 is totally superfluous, since after PROCprompt, we have default windows anyway.

## *Chapter 6*

### ***PROCdraw\_base\_screen***

**File :** util1

**Called by:** @HELP, USER : usually called early on in main program

**Purpose:** Sets up initial screen for application task.

**Globals:** None

**Calls:** PROChighlight - copyright message & title in inverse  
FNinkey(-1, ) - display time at once (no inkeys)  
PROCdraw\_box - once for selection box, once for info box.  
NB Called by PROC@HELP, so must not contain any code which is preemptable outside BASIC's control.

**Description:** Force mode 0 which clears the screen, defaults windows etc.  
Do VDU19's to give us BlueScreen.  
Invert video, and output the title string on the top line. Do a FNinkey to force time display at top right (without doing any INKEYs).  
Draw a big (display) box and a little (prompt) box.

### ***PROCdraw\_box(L%,R%,T%,B%)***

**File :** util1

**Called by:** draw\_base\_screen, fkey\_boxes

**Calls:** -

**Purpose:** Draw a box, clearing the screen inside it.

**Globals:** Assumes that top\_left\$, top\_right\$, flat\_bar\$, vert\_bar\$, bot\_left\$, bot\_right\$ have been set up by PROCdeclarations.

**Parameters:** Left, right margins, top, bottom margins. (Coordinate system is left handed from top left corner.)

**Exit conditions:** No text window.

**Description:** First create a text window internal to the box, and clear it to current background, then back to default window. Print left top corner, whole of top edge, top right corner. Print left and right edges of box one line at a time (note that we use FOR, so we cannot draw a box with zero internal height). Finally, draw bottom left corner, bottom edge and bottom right corner.

***FNEx\_key\_char(inkey\_intnl\_parm%)***

File : util1

Called by: @HELP, screen\_dump

Calls: -

Purpose: Sneak preview of keyboard, updating time display at top right of screen - unlike FNinkey, we assume that cursor isn't important, and don't update it.

Globals: No globals, no declarations required.

parameters: ..parm%(>=0) : equivalent to INKEY parameter

Exit conditions: Only exits when a character is found in buffer. If one is found within the first time ...parm%, then highlight state is preserved, otherwise set to zero.

Description: The parameter specifies how often (interval in centiseconds) we inspect the keyboard buffer. We will always wait until there is a key in the buffer, updating the time in the top right of the screen every time round the timeout loop.  
When the routine exits, the character is still in the buffer, and no event will have been precipitated through INKEY. However, if BASIC is running in a normal environment, preempts will still occur.

***PROCfkey\_boxes***

File : util1

Called by: init\_fkeys, @HELP

Calls: draw\_box, vert\_bar

Purpose: Draw boxes for fkey display.

Globals: No globals, no parameters

Exit conditions: No text window, highlight level preserved.

Description: Draws two boxes, four fkeys in each, positioned off right margin, so we avoid the dreaded BOTTOM-RIGHT-HAND-CORNER problem  
For each set of four fkeys, call PROCdraw\_box to draw an outline, then three calls of PROCvert\_bar to fill in the vertical divisions.

***PROCfkey\_caption(f\_c\_intnl\_index%,f\_c\_intnl\_on%)***

File : util1

Called by: display\_fkeys, @HELP

Calls: -

Purpose: Display fkey caption in bold or shaded out.

Description: If the caption is a null string, do nothing. Otherwise, if the second parameter is TRUE, print Fn ( n in 1 to 8 ) and the caption. If the second parameter is FALSE, then print a patch of shading in print-at-graphics-cursor mode in normal video, so only the foreground pixels are written - these are in background colour with respect to the fkey display, so the effect is to write a lot of background pixels over the existing display, thus leaving it shaded out.

## *Chapter 6*

### ***PROChighlight(level%)***

File : util1  
Called by: draw\_base\_screen, display\_fkeys, inkey, warning, info, @HELP  
Calls: -  
Purpose: Implement level of highlighting required.  
Description: NB current version is based on MODE 0, 3 or 4 display with only two colours, but recommendations on input parameter values for existing applications allow for future expansion.  
Normal video is achieved with COLOUR1: COLOUR128 while inverse has the colours interchanged, ie. COLOUR0: COLOUR129  
parameters: level% normally 0 for normal and 1 for inverse video.

### ***PROCinfo(level%,info1\$,info2\$,info3\$)***

File : util1  
Called by: warning, prompt, screen\_dump, display\_help  
Calls: highlight  
Purpose: Display three lines of text in the info box, in the specified highlight state.  
Globals: No globals assumed.  
parameters: level% specifies highlight level  
info1\$, info2\$, info3\$ contain the text to display. Max line length is 74 characters, no checking is carried out.  
Exit conditions: No text window, highlight level 0.  
Description: Define text window filling the prompt box, set the highlight state and clear the window to current background colour. Return to default window and print the specified information in the box. Finally restore the highlight state to normal.

### ***PROCinit\_fkeys***

File : util1  
Called by: display\_fkeys, @HELP  
Calls: fkey\_boxes  
Purpose: Draw boxes and display all valid fkeys.  
Description: Highlight assumed already inverse video (should be set to PROChighlight(1) by caller).  
First draw boxes for fkey display, setting current\_options to indicate that this has been done. For each fkey, check if the caption is non-null. If it is, set the appropriate bit in current\_options using a bit mask, which we shift between iterations, and display the Fkey caption.

***FNinkey(inkey\_inini\_parm%,inkey\_inini\_x%,inkey\_inini\_y%)***

File : util1  
Called by: draw\_base\_screen, warning, USER  
Calls: highlight  
Purpose: Poll keyboard, updating time display at screen top right and keeping cursor at ...x%, ...y%.  
Globals: No globals, no declarations required.  
parameters:  
...parm%=-1 : update time display ONLY  
...parm%>=0 : INKEY parameter to use  
Exit conditions: Only exits when a character is received. If one is received within the first time ...parm%, then highlight state is preserved, otherwise set to zero.  
Description: Note that this routine assumes an 80-character wide mode and default window.  
If entered with parameter -1, automatically sets the return value to -1 and doesn't do an INKEY, but merely displays the system time in a 21-character wide field at the top right of the screen. For other values of the parameter, the routine will not exit until a character is received, and will seek characters using the parameter as the INKEY time.  
NB The current keyboard driver does not support negative INKEY values, so FNinkey may not have negative parameters other than -1.

***PROCnull***

File : util1  
Called by: USER if needed  
Calls: -  
Purpose: Dummy routine very useful in debugging stages and in ON ... PROC statements.

## *Chapter 6*

### *PROCprompt*

File : util1  
Called by: USER, @HELP  
Calls: sysprompt.info  
Purpose: Display prompt information for this context in info box.  
Globals: display\_info\_context% specifies which prompt.  
info1\$ ... info3\$ also used by other info box routines.  
Exit conditions: No text window, highlight level 0.  
Description: display\_info\_context% is a global number for the context which refers into the data set up by PROCdefine\_prompt\_info, and PROCdefine\_system\_prompts. -1 is the 'undefined' value, which means a blank prompt box.  
Other negative values are used internally by the HELP system, and are implemented by a call to PROCsysprompt.  
For user-defined prompts, the DATA pointer is RESTORED to prompt\_data%(display\_info\_context%), three lines of text are read and displayed using PROCinfo.

### *PROCscreen\_dump(graphics%)*

File : util1  
USER  
Calls: (in event of failure) info, ex\_key\_char, prompt  
(normally) m/c graphics\_dump%, text\_dump%  
Purpose: Calls either SCRDM or TXTDMP modules to print from screen.  
Description: If C set on return from the relevant machine code call, display an error message in the prompt box. Note that we can't use PROCwarning for this since our error message isn't a predetermined one, but depends on the string pointed to by @BHA%, so we use PROCinfo directly.  
Wait for user to press space before we clear the error message, but be careful, since we might be in a no-preempt environment, so we don't want to use INKEY. Use ex\_key\_char first to make sure its safe. If user has pressed a task key, we want rid of it until he presses space, so use \*fx 15.  
Once user has pressed space, we want to resume where we were, so restore whatever was in the prompt box. More precisely, restore what the global variable display\_info\_context% says was in the prompt box.

***PROCsyprompt***

**File :** util1  
**Called by:** prompt  
**Calls:** info  
**Purpose:** Display prompt information for system context in info box.  
**Globals:** display\_info\_context% is a global number for the context which refers into the data set up by PROCdefine\_system\_prompts.  
-1 is an 'undefined' value for a blank box.  
info1\$ also used by other info box routines.  
**Description:** If parameter is -1, shove out a blank box, and quit. If parameter is in range -4 to -2, read the appropriate first line for use in a HELP screen. Display help prompt with PROCinfo.

***PROCutil\_assemble***

**File :** util1  
**Called by:** declarations - no other call permitted  
**Calls:** align  
**Purpose:** Assembles machine code used by the standard utilities.  
**Description:** User may declare his own code separately if required. We first give ourselves a DIMensioned area (guaranteed all in one bank) to hold the code and some data passed by indirection.  
For the machine code declared by this routine, see "Machine code entry points" below.

***PROCvert\_bar(tab% , T%,B%)***

**File :** util1  
**Called by:** fkey\_boxes  
**Calls:** -  
**Purpose:** Draw vertical bar to divide box.  
**Globals:** assumes top\_TS, vert\_bar\$, bot\_TS set up by PROCdeclarations  
**Parameters:** tab% is position of bar, T%, B% are top and bottom lines.  
**Exit conditions:** Text window and highlight level unchanged.  
**Description:** Note that it is the user's responsibility to ensure that the tabs and top and bottom line positions that he supplies refer to the current window. It is recommended that the routine be used with default window, to maintain consistent numbering.  
Print a T-shape at top of bar, then print a vertical bar for each line of the bar height (note we use FOR, so the bar must be at least 3 lines high). Finally print an inverted T-shape for the bottom of the bar.

## *Chapter 6*

### *PROCwarning(warning\_no%)*

**File :** util1  
**Called by:** USER  
**Calls:** highlight, info, clear\_buffer, inkey  
**Purpose:** Display error message in the info box.  
**Globals:** Assumes all the RESTORE labels have been declared and assigned by PROCdefine\_error\_messages called from PROCdeclarations. press\_space\$ must also have been declared before use. info1\$ ... info3\$ also used by other info box procs.  
**Parameters:** An error (non negative) no. which refers into the data set up by PROCdefine\_error\_messages.  
**Exit conditions:** No text window, highlight level 0.  
**Description:** The error message pointed to by error\_data%() indexed by the parameter is RESTORED, and the highlight state and three lines of text are read. The text is displayed, the keyboard buffer cleared, and then the routine waits for the user to press space.

### **6.7.2 Usage**

PROC@HELP and PROCdisplay\_help should not be called by the programmer, since they form part of the help system, which operates essentially "in parallel" with the main program.

PROCskey\_boxes, PROCskey\_caption, and PROCsinit\_fkeys should not be called by the programmer without great care, since they could confuse the main PROCdisplay\_fkeys routine, which should always be used for changing the fkey display.

PROCdraw\_box and PROCvert\_bar will only be found useful directly if you are doing a lot of formatting within the standard display.

PROCsysprompt exists for internal use by PROCprompt when called by the help system, and is not really useful for the programmer.

### **6.7.3 Util2 routines - must be provided by user**

#### *PROCdefine\_error\_messages*

**File :** util2  
**Called by:** Declarations - no other call permitted.  
**Calls:** -

#### *PROCdefine\_fkey\_captions*

**File :** util2  
**Called by:** Declarations - no other call permitted.  
**Calls:** -

***PROCdefine\_help***

File : util2  
Called by: Declarations - no other call permitted.  
Calls: -

***PROCdefine\_prompt\_info***

File : util2  
Called by: Declarations - no other call permitted.  
Calls: -

The user should never need to call PROCdefine\_error\_messages, PROCdefine\_skey\_captions, PROCdefine\_help, PROCdefine\_prompt\_info or PROCdefine\_system\_prompts himself as they are purely declarations and need to be called only once.

#### 6.7.4 Machine code entry points

***examine\_key\_buffer%***

A byte mode entry point calling COP (\$)OPXKC - used by FNEx\_key\_char

***text\_dump%***

A word mode entry point to do a COP (\$)OPCMD to the TXTDMP module

***graphics\_dump%***

A word mode entry point to do a COP (\$)OPCMD to the SCRDMP module

***childs\_help%***

A word mode entry point used by PROC@HELP if it finds that the variable childs\_help% is defined.

Calls (\$)CO with X=(\$)COHELP,  
Y=handle of the child coroutine (from childs\_help%)

## 6.8 Source Code Management

The routines in the standard package which should be incorporated unchanged into a new BlueScreen application (the "util1" routines) are in:

**\$S.BAS\_UTILS.GN4\_UTIL1** in source form, and

**\$S.BAS\_UTILS.I\_util1** in compressed (uncommented) form

The latter file should be \*EXECed into the user's program at the stage of conversion from source to Tokenised program (see the documentation on writing BASIC program modules). The utilities package occupies line numbers from 25000 upwards. The current version reaches line number 27300, but the user should avoid numbers below 28000.

Examples of the user-provided routines can be found in a "skeleton" form in:

**\$S.BAS\_UTILS.GN3\_UTIL2**

## 6.9 Error Messages

PROCwarning can deliver a number of error messages which are defined by the user in PROCdefine\_error\_messages. After displaying the error message, the prompt box is corrupt and the user is expected to deal with this.

The only other source of error messages is the PROCscreen\_dump routine. If carry set is received from the TXTDMP or SCRDMP routines, then \$@BHA% is assumed to point at the error message and the error :

"Print failed because"

\$@BHA%

"Press space to continue"

is displayed. When the user presses space, the prompt box is restored by reference to the current prompt, as defined by display\_info\_context%. Thus the user is warned not to display text in the prompt box using PROCinfo or PRINT statements if PROCscreen\_dump may be called.

## 6.10 OS calls required

BASIC INKEY statement is used, which interfaces to MOS through OSBYTE &81.

COP (\$)OPXKC used within PROC@HELP and PROCscreen\_dump.

COP (\$)OPCMD to TXTDMP and to SCRDMP modules. These modules also use a number of MOS interfaces connected with reading the screen.

JSL (\$)CO with X=(\$)COHELP may be used to call a child coroutine for help.

OSCLI"fx 15 1" used to clear the keyboard buffer in places where preempts are not tolerable.

BASIC TIMES() statement is used, which interfaces to MOS.

## 7. Device Drivers

### 7.1 Introduction

A Device driver presents a standard abstract driver interface to some underlying hardware, permitting the design of device-independent applications software.

By hiding the complexity of the underlying hardware, the design of applications software is greatly simplified.

In addition, device drivers can be used to restrict access to hardware, providing a form of resource control.

On Communicator, a device driver is implemented as a module that supports a set of standard device driver reason codes, (these define the abstract driver interface).

Applications usually access device drivers via streams. A stream consists of a sequence of bytes. Drivers support two types of stream:

Control Streams - used to send commands to and read status from a device driver

Data Streams - used to write and read data from to and from a device driver.

Each device driver has an associated device driver filename. When an application first wishes to access a device driver it performs an OPEN operation on the device driver filename. If the OPEN succeeds the application is returned a device driver handle which it uses for all following driver access. Once opened, the application communicates with the driver via the data and control streams. When the application has finished with the driver it performs a CLOSE operation on it, freeing the driver for use by other programs. If after it has performed a close operation on a device, the application wishes to access that driver again, another OPEN operation must be performed on it.

BASIC provides support for device driver access. Applications written in assembler have access to a wider repertoire of driver functions - where supported.

The remainder of this Chapter covers the above topics in more detail and then goes on to discuss the general procedure to be adopted in the design and implementation of driver by OEMs. A dummy device driver listing is provided at the end of the document to help illustrate some of the concepts covered.

It is recommended that, if they have not already done so, readers consult the Communicator Systems Manual on device drivers and file systems before proceeding further.

## **7.2 Device Driver Module Reason Codes**

A device driver is a special module that implements a set of standard device driver reason codes.

The following basic reason codes must be supported by all device drivers:

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

In addition to the basic reason codes, certain drivers implement some or all of the following extended reason codes:

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

Drivers implementing extended reason codes are usually filing systems.

## 7.3 Device Driver Control Streams

After opening a device driver, an application controls it via the driver's associated control stream and sends and reads data to and from it via the associated data stream.

Commands in the control stream consist of a single command character followed by an ASCII string of zero or more characters containing parameters associated with the command. Some commands allow their parameter string to be terminated by an upper-case letter or special symbol. All parameter strings may be terminated by a control character.

Command and parameter decoding is performed automatically from a driver command table. The following types of parameter are currently supported:

- NOP** - the command takes no parameters.
- OPT** - the command must be one of a number of options (e.g. on/off). Options must not include upper-case letters.
- DEC** - the parameter is a two-byte decimal number.
- HEX** - the parameter is a three-byte hexadecimal number.
- STR** - the parameter is an arbitrary string, (currently limited to thirty characters).

Parameter types HEX and STR may only be terminated by a control character.

## 7.4 Device Driver Filenames and Handles

Every device driver in the Communicator system has a unique device driver filename associated with it. The device driver filename enables the Communicator MOS to locate specific device drivers. The filename is only used when a driver is OPENED. The filename of a device driver normally corresponds to the name of the module that implements it.

When opening a driver the application passes a string of the following form:

<module name >:(<file specifier >) (<command sequence >)

where

- module name** is the name of the module implementing the driver.
- file specifier** is a qualifier whose interpretation is driver dependent.  
For example, if the driver is a file system, the file specifier is a filename. In other drivers it could be used to select between different minor devices.
- command sequence** is a series of control-stream commands to be sent to the driver immediately after opening. Any ";" characters will be substituted with carriage returns.

If an application successfully OPENS a driver it is returned a unique device driver handle. The handle returned by the OPEN action refers to the driver's data stream. The handle of the driver's control stream is the value of the handle returned plus one.

## **7.5 Writing Device Drivers**

This section attempts to provide some guidance for those wishing to develop their own device drivers.

### **7.5.1 Device Driver Reason Codes**

All device drivers must support the set of basic device driver reason codes mentioned in section 2 of this document, so as to present applications with the standard device driver interface.

In addition to the basic reason codes, the designer may wish to implement some or all of the extended reason codes where they are applicable to the hardware and it is anticipated that they will be required by, or useful to, applications software.

### **7.5.2 Returning Status**

All implemented driver operations must return status to the calling coroutine indicating success or failure in the appropriate manner. If the requested operation is successful the driver should return control to the calling coroutine with the Carry Flag RESET. If the requested operation fails the driver should return control with the Carry Flag SET and an error code in the X register. In addition, if the designer so wishes, the driver may return a pointer to an error message in registers BHA, signifying its presence by SETTING BIT0 in the X register.

### 7.5.3 Device Driver Macros

In order to simplify the creation of device drivers a number of driver macros are supplied as part of the OEM developers' package. These macros implement driver command table creation and command decoding.

The following driver macros are located in the file \$S.getdecode01:

<b>DTDTAB</b>	contains routines to implement control stream command decoding.
<b>DTDENT</b>	used to add entries to a driver module's command table.
<b>DTDEND</b>	used to terminate a driver module's command table.

Associated with the above macros is a set of manifest constants located in the file \$S.get.dtd01

#### *The DTDTAB Macro*

The DTDTAB macro contains a set of functions to decode commands received on a driver module's control stream. The DTDTAB macro should be invoked directly after the module's header information, (usually created via the START macro).

The DTDTAB macro implements the following routines which are accessible to the module via global pointer variables:

<b>DTDALD</b>	Allocate a handle and direct page memory. Accessed via the global pointer \$STDALD.
<b>DTDCHD</b>	Convert a driver handle to a Direct Page pointer. Accessed via the global pointer \$STDCHD.
<b>DTDCLS</b>	Relinquish direct page and handle. Accessed via the global pointer \$STDCLS.
<b>CON</b>	Checks to see driver has been opened in CONfigure mode.
<b>OPEN</b>	Resets the buffer pointers for a channel. Invoked via the global pointer \$STD_RST.
<b>WRCS</b>	Decodes bytes from a control stream. Accessed via the global pointer \$STD_CPT.
<b>RDCS</b>	Reads bytes from the status string. Accessed via the global pointer \$STD_CGT.
<b>SPT</b>	Adds a byte to the non-automatic status string. Accessed via the global pointer \$STD_SPT.

The macro DTDTAB is invoked thus:

```
DTBTAB $identifier, $mod_entry_addr, $global_dp, $local_dp, $global_dpinit,
$module_name
```

Where:

<b>\$identifier</b>	is a string of not more than 4 characters (not in quotes) which uniquely identifies the command table, (eg DUM for a dummy driver). This is used to generate symbols consisting of this string followed by 2 hex digits, used in building up the command table.
<b>\$mod_entry_addr</b>	is the entry address for all the module reason codes. This address will be entered in WORD mode with D pointing to your global direct page, and should be exited with an RTS.
<b>\$global_dp</b>	specifies how much global direct page the driver needs. This is added to the amount needed by the DTD system and allocated for you. The driver can use the direct page workspace from 0..(\$global_dp-1).

**\$local\_dp**

specifies how much direct page the driver needs which is local to each handle. This is added to the amount needed by the DTD system and allocated for you when you call \$DTDALD (see below). The driver should only use direct page from offset (\$)DTDDP onwards - all below that is reserved for the DTD system. The symbol or expression assigned to \$local\_dp must already be defined when the macro is called on the first pass. It is usual, therefore to declare the driver's local direct page earlier in the source (using '^ (\$)DTDDP' followed by use of the '#' command).

**\$global\_dpinit**

is an address of a routine to initialise the global direct page workspace. This will be called when this workspace is allocated, usually the first time the module is called. The routine is entered in WORD mode, with D pointing to the global direct page. If no global direct page has been claimed (\$global\_dp=0), this should point to an RTS.

**\$module\_name**

is the name of the module, (ie the same as that used in the START macro) as stored in the module header (not in quotes).

***DTDTAB Functions******The DTDALD Function***

The function DTDALD is used to allocate a local direct page and handle when a device or minor device is OPENED. The address of the direct page is stored in the workspace associated with the handle.

Called with:      The processor in WORD mode.  
                   D -> global direct page.

On exit:      C = 0 succeeded.  
                   Y = Handle.  
                   A = direct page allocated. D is preserved.  
                   C => 1 failed.  
                   A,Y undefined.  
                   D preserved.

***The DTDCHD Function***

The DTDCHD routine converts a given driver handle into a pointer to that handle's local Direct Page.

Called with:      The processor in WORD mode.  
                   Y = handle.

On exit:      D -> local direct page.  
                   A,X,Y preserved.

***The DTDCLS Function***

The DTDCLS routine de-allocates the direct page associated with the supplied handle along with the handle itself.

Called with:      The processor in WORD mode.  
                   Y = handle.  
                   D -> Direct Page to be freed.

On exit:      C = 0 succeeded.  
                   C => 1 failed.

***The CON Function***

The CON function checks to see if the driver has been opened in configure mode. That is with the "~" as the first character of the filename.

Called with:      The processor in WORD mode.

BHA -> rest of name.

On exit:      C=1, X=2 configure open  
                   C=0, X=0 normal open.

## *Chapter 7*

### *The OPEN Function*

The OPEN routine resets the buffer pointer for the specified channel.

Called with: Processor in WORD mode.  
D -> Local Direct Page.  
A = the flag returned in X by CON.

### *The WRCS Function*

This routine decodes command sequences received on a driver's control stream. If a valid command is received, the corresponding routine is invoked.

Called with: Processor in WORD mode.  
D -> Local Direct Page.  
A = character to be put to the control stream.

### *The RDGS Function*

This routine reads a byte from the current status string.

Called with: Processor in WORD mode.  
D -> Local Direct Page.  
  
On exit: C = 0 successful  
A = character read.  
  
C = 1 End of status string.  
A undefined.

### *The SPT Function*

This routine is used to build up the status string returned in response to a query "?" command on a driver's control channel. The character supplied in register A is added to the current status string. The first character of the string is created automatically and corresponds to the character following the query command.

For example, in the telecom driver the command "?d" tests for the presence of a carrier. The returned string starts with a "d" and the driver calls SPT with the characters "on" or "off" corresponding to the presence or non-presence of carrier, ie the returned status string is either "don" or "doff".

Called with Processor in WORD mode.  
D -> local Direct Page.  
A = character to be added to the string.

***The DTDENT Macro***

The DTDENT macro is used to create entries in the driver module command table. Each legitimate command received on a driver's control stream has an entry in the driver's command table. An entry in the command table specifies the type of parameters a command takes, a vector to a piece of code that implements the command, a list of option strings where applicable, and an optional information string giving a description of the the command (used by configure ).

A call to the DTDENT macro takes the following format:

```
$dpstat DTDENT $command, $param_type, $entry_address, $option_str, $info_str
```

where:

**\$command** is the single character string (in quotes) which the user sends down the control stream to initiate the command.

**\$param\_type** is an identifier which specifies the type of parameter the command takes, as follows :-

<b>NOP</b>	- he command takes no parameters.
<b>OPT</b>	- he parameter must be one of the options presented in the option string (see below).
<b>DEC</b>	- he parameter is a 2-byte decimal number
<b>HEX</b>	- he parameter is a 3-byte hex number (no #).
<b>STR</b>	- he parameter is an arbitrary string.
<b>OPC</b>	- only used for the "?" command - see below

Parameter types HEX and STR are only terminated by a control character, other types may also be terminated by the start character of the next command (ie !, @-Z).

**\$entry\_address** is the address which will be called when a complete and valid command has been issued. The routine will be entered in WORD mode with the following entry conditions :-

<b>NOP</b>	- no entry conditions
<b>OPT</b>	- Y = option used (0,2,4 ...)
<b>DEC</b>	- HA = number
<b>HEX</b>	- BHA = number
<b>STR</b>	- BHA -> string

The routine should exit with an RTS.

**\$option\_str** This string is only used when the option string parameter types (OPT or OPC) are being used. This type is used when a valid parameter to the command consists of one of a number of fixed strings (eg "on" or "off"). The option string consists of all these strings, separated by a delimiting character (normally "/"), and preceded by the character used for the delimiter eg "/on/off". When such a command has been received, the entry\_address will be called with Y in this case being 0 for on, 2 for off. For the OPC parameter type (used by "?"), the option string will be used

if the parameter string is not one of the automatic status commands. If the parameter type is not OPT or OPC, this string should be "". The option string for a given command can be read by sending the "!" command, followed by the command letter. This is used by Configure.

**\$info\_str**

This string gives a description of what the command sets. The string is read by sending the "%" command followed by the command character. The Configure program reads this string and prints it out on the left hand side of each entry. If it is not sensible to have a configured setting for this command, then \$info\_str should be set to "".

**\$dpstat**

This optional label, if present, indicates that an automatic status entry should be maintained for this command. Depending on Sparam\_type, an amount of local direct page will be automatically allocated for it (1 byte for OPT, 2 for DEC, 3 for HEX, 32 for STR) and the label \$dpstat will be assigned to the start of that workspace, for use by the driver if necessary. Whenever a command is issued which has an automatic status entry, the parameter to the command is stored in the direct page allocated for it, before calling \$entry\_address.

If any of the commands have been assigned automatic status entries, then the module writer should include an extra DTDENT call of the following form:-

**DTDENT "?",OPC,\$status\_entry,\$status\_option\_string**

This sets up a command "?" which requests status information specified by the character after the "?". If this character is a command character which has an automatic status entry (ie \$dpstat was specified) then a status string will be put into the control BGET buffer which consists of the command character, followed by a string representing the last setting of that command, in a similar form to that originally specified (ie option string types are returned as one of the options, decimal and hex numbers are returned in ASCII string format).

If the character after the "?" does not match any such command character, then the command is treated in a similar fashion to a normal OPT command (ie if the parameter to the "?" command matches anything in \$status\_option\_string, then \$status\_entry will be called, with appropriate entry conditions). The differences are:

- a) that the "?" command terminates only on a control character, and
- b) that the control stream status buffer is reset and the first character received after the "?" is put into this buffer, as the start of a non-automatic status string.

When \$status\_entry is called, the driver can repeatedly call the global SDTDSPT with A-successive characters to add to the status string. (The string is terminated at all times, so no terminator need be sent.)

The returned status string is read by successive BGETs from the control stream, and is terminated by a control character (so it can be read from BASIC using GET\$#control.)

To differentiate between the automatic status commands, and the non-automatic ones (ie those specified in \$status\_option\_string), it is a convention that the non-automatic statuses are specified by lower-case letters.

### The DTDEND Macro

The DTDEND macro is used to terminate a driver module's command table and tidy up the world, (eg all the option strings and info strings are assembled at the end of the table).

DTDEND starts by making 2 further calls to DTDENT; these define the "%" and "!" commands, which are used by CONFIGURE to obtain the command descriptions and parameter types (respectively) from each driver. The command "%%" is now used by CONFIGURE to fetch a string representing all configurable commands taken by the driver; this string is created automatically by the DTDENT calls, and requires no action by the device driver author.

DTDEND should be called immediately after the last DTDENT macro.

DTDTEND is invoked thus:

```
DTDEND $description_string
```

\$description\_string is a quoted string of not more than forty characters which describes the driver. Null parameters may be supplied either in quotes, or (for compatibility with existing drivers) without quotes. This string is recovered by CONFIGURE using the "%%" command, and will appear following the device name at the top of the relevant configure page.

### 7.5.4 Implementing a Device Driver

The SYSTEM macro should be invoked at start of the device driver source code in order to set up the MOS variables followed by two GET directives to include the files \$.S.get.d1d02 and \$.S.get.decode01.

Next should come the standard system macro START which sets up the driver's module header.

The module header should be followed by the command table and associated functions. The start of the command table is entered via the macro DTDTAB. This is followed by calls to DTDENT to create table entries for the commands supported by the driver. The DTDEND macro is used to terminate the command table.

Following the command table should come the driver module entry source. The entry point of the module must interpret and act on the reason codes passed to the driver when it is invoked. The designer should attempt to make all driver module code relocatable.

The driver module code may access the functions provided by DTDTAB when required via their associated global pointers.

The following piece of code implements a driver called "DUMMY" and illustrates the general method of driver construction:

*Chapter 7*

```
; > Dummy

OPT    2

SYSTEM
GET    $s.get.dtd01
GET    $s.get.decode01

OPT    1

MACRO
$label BRL4  $addr
$label BRL   $addr
=      0
MEND

ORG    &FF0000      ; conventional ORG for position
; independent code.

; Declare local direct page here

        ^      'DTDDP
ldpslt #      0
zfred  #      1

ldpsize *     @-ldpslt

; Module header

START  Dummy,I.,'MHCBNK+'MHCPOS+'MHCDEV

; Command table

        DTDTAB DUM,DMENTR,0,ldpsize,GINIT,Dummy
        DTDENT "X",NOP,XCOM,"",""
hstat  DTDENT "H",HEX,HCOM,"","Hex number"
tstat  DTDENT "T",OPT,TCOM,"/lunch/tea","Meal"
sstat  DTDENT "S",STR,SCOM,"","String"
dstat  DTDENT "D",DEC,DCOM,"","Decimal number"
        DTDENT "?",OPC,STAT,"",""
        DTDEND
```

**DMENTR WRDROUT**

; Check reason code

```
ASSERT 'DVRST=&10
ASSERT 'DVOPN=&12
ASSERT 'DVCLS=&14
ASSERT 'DVBGT=&16
ASSERT 'DVBPT=&18
ASSERT 'DVCGT=&1A
ASSERT 'DVCPT=&1C
```

```
CPXIM 'DVRST
BCC SECXIT
CPXIM 'DVCPT+2
BCS SECXIT
```

**DMFIX** \*     'DVRST :SHL: 1       ; Fix NEWTUR bug (hopefully!)

**PER**    SRVTAB-1-DMFIX       ; make room for address to go to

```
PHA                              ; save A
TXA
ASLA                          ; multiply by 2
                                ; (so now is mult of 4)
CLC
ADCS 3
STAS 3                      ; store back on stack
PLA                            ; restore A
RTS                            ; go to that address
```

**SRVTAB**

BRL4	DMRST
BRL4	DMOPN
BRL4	DMCLS
BRL4	DMBGТ
BRL4	DMBPT
BRL4	DMCGT
BRL4	DMCPT

**SECXIT**

SEC
RTS

```
:*****  
:  
:      Reset driver  
:      Not currently used by system  
:  
:in: WORD mode  
:      Y = handle  
:  
:out: -  
  
DMRST WRDROUT  
BSR $DTDCHD      ; convert handle in Y to local DP  
BRL $DTDRST      ; reset buffer pointers
```

```
:*****  
:  
:      Open stream to driver  
:  
:in: WORD mode  
:      D -> global DP  
:      BHA -> rest of filename, '@' implies configure open  
:      Y = open type (&40 = OPENIN, &80 = OPENOUT, &C0 = OPENUP)  
:  
:out: C=0 => successful open, Y = handle  
:      C=1 => failed to open, Y undefined
```

```
DMOPN WRDROUT  
BSR $DTDCON      ; check for '@'  
PHX             ; save 'configure' flag  
BSR $DTDALD      ; allocate local DP and handle  
PLX             ; restore 'configure' flag  
BCS SECXIT      ; couldn't allocate local direct page  
TAD             ; D:= local DP  
TXA             ; transfer configure flag to A  
PHY             ; save handle  
BSR $DTDRST      ; initialise buffer pointers  
PLY             ; and restore  
CLC             ; indicate successful open  
RTS
```

```
;*****  
;  
; Close stream to driver  
;  
;in: WORD mode  
; Y = handle  
;  
; out: -  
  
DMCLS WRDROUT  
    BSR $DTDCHD      ; get local direct page  
    BSR $DTDCLS       ; free direct page  
    RTS  
  
;*****  
;  
; Get a byte from data stream  
;  
;in: WORD mode  
; Y = handle  
;  
; out: C=0 => byte successfully read, A = byte read  
;      C=1 => failed to read a byte  
  
DMBGT WRDROUT  
    BSR $DTDCHD      ; get local direct page  
    OPSYS 'OPWRS  
    = "BGET from handle ",0  
    TYA  
    BSR DMWRHX       ; write handle  
    OPSYS 'OPNLJ  
    LDAIM 42          ; return 42 from BGET !  
    CLC  
    RTS
```

```
; *****
; Put a byte to data stream
;
; in: WORD mode
;      Y = handle
;      A = byte to be written
;
; out: C=0 => byte successfully written
;       C=1 => byte failed to be written

DMBPT WRDROUT
    BSR $DSTDCHD      ; get local DP
    OPSYS 'OPWRS
    = "BPUT character ",0
    BSR DMWRHX
    OPSYS 'OPWRS
    = " to handle ",0
    TYA
    BSR DMWRHX
    OPSYS 'OPNLI
    CLC           ; indicate success
    RTS

; *****
;
; Get byte from control stream
;
; in: WORD mode
;      Y = handle (of data stream)
;
; out: C=0 => A = byte read from status buffer
;       C=1 => no bytes in status buffer, A undefined

DMCGT WRDROUT
    BSR $DSTDCHD
    BRL $STDTCGT
```

```
;-----  
;  
; Put byte to control stream  
;  
; in: WORD mode  
; Y = handle (of data stream)  
; A = byte to be written  
;  
; out: C=0 always
```

```
DMCPT WRDROUT  
    BSR $DTDCHD  
    BRL $DTDCPT
```

```
;-----  
;  
; Write hex routine
```

```
DMWRHX WRDROUT  
    PHA  
    SWA  
    BSR DMWRH2  
    PLA  
    PHA  
    BSR DMWRH2  
    PLA  
    RTS
```

```
DMWRH2  
    ANDIM 255  
    PHA  
    LSRA  
    LSRA  
    LSRA  
    LSRA  
    BSR DMWRH3  
    PLA
```

```
DMWRH3  
    ANDIM 15  
    SED  
    CMPIM 10  
    ADCIM "0"  
    CLD  
    OPSYS 'OPWRC  
    RTS
```

```
; Code to handle commands

XCOM  WRDROUT
      OPSYS 'OPWRS
      = "This is the X command",10,13,0
GDINIT
STAT
RTS

TCOM  WRDROUT
      CPYIM 0
      BEQ  LUNCH
      CPYIM 2
      BEQ  TEA
      OPSYS 'OPWRS
      = "Unknown T command",10,13,0
RTS

HCOM  WRDROUT
      PHP
      BYTE
      OPSYS 'OPWRS
      = "Hex number ",0
      OPSYS 'OPWRC
      SWA
      OPSYS 'OPWRC
      PHB
      PLA
      OPSYS 'OPWRC
      OPSYS 'OPNLI
      PLP
RTS

DCOM  WRDROUT
      OPSYS 'OPWRS
      = "Decimal number &",0
      BSR  DMWRHX
      OPSYS 'OPNLI
RTS

SCOM  WRDROUT
      LDIXIM 2      ; terminate on any control char
      OPSYS 'OPWRA      ; write string pointed at by BHA
      OPSYS 'OPNLI
RTS

LUNCH OPSYS 'OPWRS
      = "It's lunch time!",10,13,0
RTS
```

```

TEA OPSYS 'OPWRS
= "Not tea-time already?",10,13,0
RTS

FINISH

END

```

Looking at the DTDENT macro calls it can be seen that the driver handles six commands (X, H, T, S, D and ?).

Following the command table is the entry point to the driver module labelled DMENTR. Right at the start of the module entry points are seven ASSERT directives which ensure that the expected reason codes take the expected values. If they do not the assembly will fail. The driver first checks the validity of its supplied reason code. If the value falls outside of the reason code range the module branches to the error routine SECXIT which sets the Carry flag and exits back to the invoker.

If the reason code is in range, it is used as an offset into the table SRVTAB which vectors the module to the specified reason code handler.

The DMRST routine handles the driver RESET reason code. It is called with a handle in Y. DMRST calls on the DTDTAB function DTDCHD to convert the handle into local Direct Page and then calls on the function OPEN to reset its associated buffer pointers.

The DMOPN routine handles the driver OPEN reason code. It is called with pointer to global Direct Page in D, BHA pointing to the rest of the filename and Y containing the OPEN TYPE. DMOPN first checks to see if it is being opened by configure via a call to the function CON. It then allocates a local handle and direct page via the function DTDALD. Failure to allocate either results in the driver exiting via SECXIT. Having successfully allocated both, DMOPN initialises the handle's associated buffer pointers via OPEN and exits with the new handle in Y and the configure flag in A.

The DMCLS routine handles the driver CLOSE reason code. It is called with a handle in Y. DMCLS invokes the function DTDCHD to convert the supplied handle into a Direct Page pointer and then de-allocates it via DTDCLS.

The DMBGT routine handles the driver BGET reason code. It is called with a handle in Y. DMBGT converts the handle into local Direct Page via DTDCHD, outputs a message to the screen and returns a witty value.

The DMBPT routine handles the driver BPUT reason code. It is called with a handle in Y and the data byte in A. DMBPT converts the handle into local Direct Page via DTDCHD and outputs a message to the screen (including the character supplied in A).

The DMCGT function handles the driver CGET reason code. It is called with a handle in Y. DMCGT converts the handle into local Direct Page via DTDCHD and returns a byte from the control buffer via the function RDCS.

The DMCPT function handles the driver reason code CPUT. It is called with a handle in Y and a control byte in A. DMCPT converts the handle into local Direct Page via DTDCHD and processes the control byte via the WRCS function.

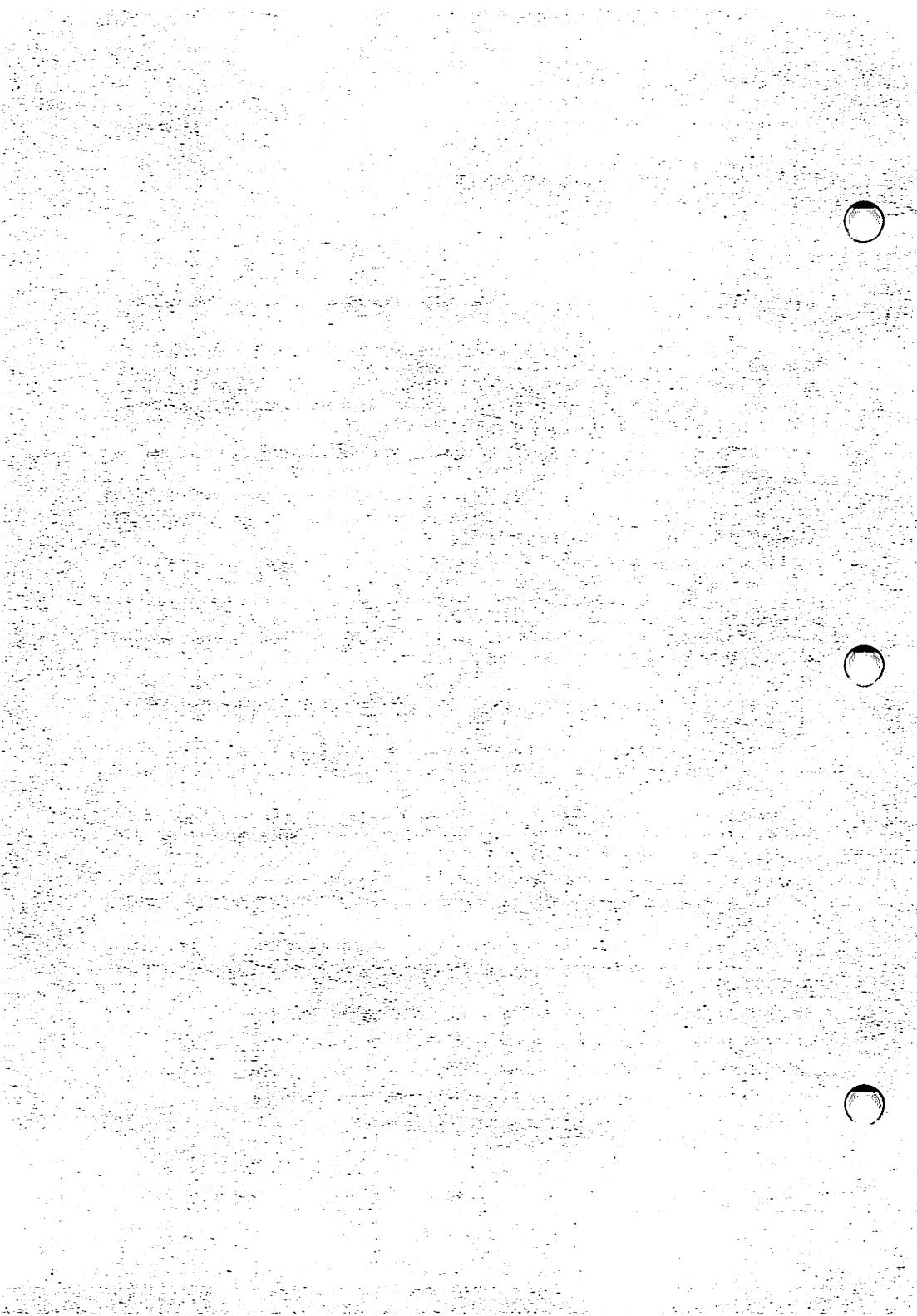
The routine XCOM is invoked by the command handler on decoding the X command.

The routine TCOM is invoked by the command handler on decoding the T command.

The routine HCOM is invoked by the command handler on decoding the H command.

The routine DCOM is invoked by the command handler on decoding the D command.

The routine SCOM is invoked by the command handler on decoding the S command.



## 8. Communicator emulation modules

### 8.1 Overview

This Chapter gives programming information for Communicator emulation modules version 1.00.

Emulation modules are part of the PHONE section of the Communicator software. There is only one Communicator terminal program, which consists of two modules - the terminal shell (TSHELL) and the terminal kernel (KEYPAGE). This is customised for varying terminal emulations such as VT100 and Videotex through emulation modules. Each emulation module provides an abstract emulation of a particular actual terminal type. The emulation is abstract in the sense that the emulation module does no I/O itself, but simply translates character streams which are then interpreted by the PHONE software.

This Chapter discusses emulation modules in general and documents the routines which are common to all emulation modules. There are three separate sections which cover the individual emulation modules:

- Teletype/BBC emulations
- Videotex emulation
- VT100 emulation

The Communicator Systems Manual documents a considerable part of the interface to emulation modules. That information is not repeated here.

The terminal shell searches for emulation modules at run time and does not need to be written with a knowledge of what emulation modules will be available. This means that new emulations can be written without disturbing the terminal shell code.

The emulation to be used for a particular PHONE service name is taken from the directory entry for the service. Within Communicator, the module name of an emulation module is the characters E. followed by the emulation name. The four Communicator emulation modules are therefore named:

- E.TELETYPE
- E.VT100
- E.VIDEOTEX
- E.BBC

No other modules in Communicator should have names starting with E..

An emulation module is activated (opened) when a call is made to a directory entry that requires that module. The module is de-activated (closed) when that call is cleared.

Emulation modules are written as Communicator device drivers and share many calls with normal device drivers, in particular the control character stream calls.

The PHONE directory entry may contain just the emulation type, e.g. TELETYPE, or may have the type followed by a string of option characters, eg TELETYPE:N+. 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. They generally set up optional operating modes for the emulation.

## **8.2 Software Interface**

The software interface is fully described in the Communicator Systems Manual. The precise behaviour of each module is given in the specific emulation section.

## **8.3 Environment**

Emulation modules behave as device drivers within the Communicator operating system, though they do not control any hardware. They are only useful within the context of the PHONE application, as they do not handle standard I/O calls (BPUT and BGET) but only their own specialised calls. It does not make sense for example to open an emulation module and then copy a file to it. You can do it, but nothing at all will happen.

Emulation modules are "multi-user", in the sense that any emulation module can be opened multiple times simultaneously. Each instance is completely independent.

When an emulation module is opened it allocates itself a certain amount of "local direct page" memory. For the four existing emulation modules this amount is small (< 256 bytes). However, future emulation modules might need significant amounts of memory to hold one or multiple screen images with attributes.

Emulation modules assume a particular mapping of special keyboard keys into "top bit set" characters. This mapping is:

Key	Hex code
HOME	&80
F1..F8	&81..&88
INSERT	&89
COPY	&8B
Left arrow	&8C
Right arrow	&8D
Down arrow	&8E
Up arrow	&8F
Shift F1..F8	&91..&98

## 8.4 Structure

This section describes the major flow of control through an emulation module. Emulation modules are written with heavy use of the "DTD" series of Communicator device driver macros. These are documented separately.

As with all Communicator drivers, there is a single entry point, defined in the macro EMSTRT (file xxx02, defined in emlmacro). EMSTRT sets up the module header and jumps to the label ENTRY. ENTRY is defined within the DTD macros, and handles common entry and exit code for drivers. It carries out a BSR to the routine RCODES (in file RCODES) to process the driver call reason codes.

RCODES is common to all emulation modules. It checks that the reason code is valid and then branches through a table to specific code to handle it. These reason code handling routines are:

EMNAME	Common routines, implemented in file RCODES
EMINIT	Common routines, implemented in file RCODES
EMKILL	Common routines, implemented in file RCODES
EMHELP	Common routines, implemented in file RCODES
EMCGET	Common routines, implemented in file RCODES
EMCPUT	Common routines, implemented in file RCODES
EMOPEN	Common routines, implemented in file RCODES
EMCLOSE	Common routines, implemented in file RCODES
EMRESET	Emulation specific, implemented in xxx03 etc.
EMSERIN	Emulation specific, implemented in xxx03 etc.
EMKEYIN	Emulation specific, implemented in xxx03 etc.

Note that the normal data stream reason codes BGET and BPUT are not used by emulation modules and return an error.

EMNAME, EMINIT, EMKILL and EMHELP are very simple routines directly coded within RCODES. EMCGET and EMCPUT, the control stream routines, are implemented by the standard DTD macros \$DTDCGET and \$DTDCPUT. EMOPEN and EMCLOSE use the standard DTD macros for allocating and freeing local direct page and handles. In addition they are directly responsible for allocating and freeing the emulation buffers.

EMRESET, EMSERIN and EMKEYIN are specific to each emulation and are described in the individual emulation sections.

## 8.5 Data structures

This section describes the major data structures used by the emulation modules. Emulation modules do not use any files.

There are three major data areas used. These are:

- Local direct page
- The KEYIN response block
- The SERIN response block

The local direct page contains data used by the DTD macro package, common emulation variables and emulation specific variables.

The DTD data is documented elsewhere. The emulation module variables are placed above the DTD data in the local direct page. The DTD variable (\$)DTDDP is used to identify the start of these emulation module variables.

For historical reasons, the emulation local direct page variables are allocated using a macro AllocalDP. They are all prefixed by this macro with "l". The comments in the code mention variables prefixed with tp. This no longer applies.

The following local direct page variables are used in all the emulation modules:

GDPsave	Backward pointer from local to global direct page used as a check on the validity of handles.
EMflag	Holds various emulation mode values: Bit 7 Echo mode on if 1 Bit 14 Line feed mode on if 1 Bit 15 Newline mode on if 1
bushandle	Handle to the buffer space allocated for response blocks.
buspointer	Pointer to the buffer space allocated for response blocks.
serinblock	Pointer to the EMSERIN result block (actually equal to lbufpointer).
keyinblock	Pointer to the EMKEYIN result block
scrptr	Temporary working pointer used by the EMKEYIN and EMSERIN routines as the next place to put a screen output character.
lineptr	Temporary working pointer used by the EMKEYIN and EMSERIN routines as the next place to put a line output character.

Other emulation specific variables are described in the individual module sections.

The EMKEYIN and EMSERIN response blocks memory is allocated when the emulation module is opened and freed when the emulation module is closed. The internal structure of these is described in the Communicator Systems manual.

## 8.6 Procedures

This section includes a description of each routine in the common parts of the emulation module code. All of these routines are in the file RCODES.

### 8.6.1 RCODES

RCODES is called from the DTD driver entry code. It checks the driver reason code and if there is no error, branches to the appropriate handler routine. See "Structure" above for more information.

RCODES branches to the handlers listed in the table SRVTAB.

Note that on error, emulation modules always return X = 0, ie they do not return specific error codes.

RCODES leaves the registers set as follows for the reason code handlers:

D	global direct page
A	as set on the driver call
Y	the handle for this instance of the emulation module

### 8.6.2 EMNAME

This handles the NAME reason code. The BHA value returned points to the label EMLname. This is an area of the code initialised to the module name (set by the variable \$Module) terminated with a zero. EMNAME never fails.

### 8.6.3 EMINIT

This handles the INIT reason code. It does nothing at all and always succeeds. The allocation of global direct page, which might be expected to take place here is handled within the DTD code.

### 8.6.4 EMKILL

This handles the KILL reason code. This fails if there are any instances of the driver still open. KILL is not properly supported by the Communicator OS at present, so the EMKILL code has not been tested and will almost certainly not work when/if the OS supports KILL.

### 8.6.5 EMHELP

This handles the HELP reason code. This always returns an error, as the emulation modules do not handle help themselves.

### 8.6.6 EMCGET

This handles the control get (CGET) reason code. This calls checkhandle to set up local direct page. It then branches to the \$DTDCGT code which handles all the commands.

### 8.6.7 EMCPUT

This handles the control put (CPUT) reason code. This calls checkhandle to set up local direct page. It then branches to the \$DTDCPT code which handles all the commands.

### 8.6.8 checkhandle

This is a subroutine used by many of the reason code handlers. It performs two functions. Firstly, it verifies that the handle passed is valid by checking for equality of the global direct page of the driver and the copy of this stored in the local direct page for this instance of the driver. Secondly, it sets D to point to the driver's local direct page.

The DTD routine \$DTDCHD is used to derive the local direct page from the handle.

### **8.6.9 EMCFLAGS**

This subroutine is used by commands that set operating modes of the emulation module. It sets or clears bits in IEMflag. On entry A contains a mask of the bit(s) to modify. Y = 0 means set bits, Y > 0 means clear bits.

### **8.6.10 EMOPEN**

EMOPEN opens a new instance of the emulation module.

EMOPEN first calls \$DTDCON which checks for "configure type" opens. The real activity starts with the call to \$DTDALD. This attempts to allocate a handle and the local direct page. The next call is to \$STDTRST which resets internal DTD variables concerned with the command stream parser.

EMOPEN then saves the global direct page in IGDPsave for use later as a check in checkhandle. EMLALA (q.v.) is then called to allocate the response block memory. LocalReset is called to initialise the emulation module's own variables. This routine is defined in file xxx03 as it is specific to each emulation.

Various of these operations may fail, in which case EMOPEN tidies up and returns an error.

On a normal exit, the handle that was allocated is returned in Y.

### **8.6.11 EMLALA**

This routine allocates memory for the emulation module response blocks. It also initialises the related local direct page pointers and initialises the contents of the response blocks.

Both response blocks are allocated as a single area of memory. The memory space needed for the response blocks is defined by the macro calls DefRespBlock in file xxx01. These calls define the size of the screen and keyboard output buffers for each of the two response blocks. In the same file, bufsize is defined as the total amount of memory required.

### **8.6.12 EMCLOSE**

This closes an instance of the driver. Checkhandle is used to check that we have a valid handle. EMCLOSE then proceeds to free the response block memory. It then zeros IGDPsave in the local direct page so that any future references to the handle can be identified as an error by checkhandle. Finally the \$STDCLS routine is used to free the local direct page and the handle.

### **8.6.13 gdzero**

This routine initialises the global direct page for the emulation module. It is called from the DTD ENTRY routine. It simply sets the current number of open channels to zero.

### **8.6.14 nullroutine**

This is an RTS that is used as a do nothing call by parts of the code.

## 8.7 OS calls required

Emulation modules do few calls to the OS because they are not responsible for carrying out any direct I/O.

- (\$)OPWRS      This is used for printing debugging information under the control of the \$Test assembler variable.
- (\$)OPFZB      This is used to release the module's global direct page in an EMKILL call. Note that in the current Communicator system EMKILL is never called on an emulation module.
- Emulation modules use memory management routines in EMOPEN and EMCLOSE to allocate space for the response buffers. These are called through "JSL (\$)MM". The X values used are:
  - (\$)MMALA      Used (indirectly) in EMOPEN to allocate response buffers for a particular instance of the emulation module.
  - (\$)MMFP      Used in EMCLOSE to throw away the response buffers for a particular instance of the emulation module.
- Note that emulation modules use many standard "DTD" Communicator device driver calls. These are responsible for allocating and disposing of local direct page memory and use OS calls to do so. See the DTD section for further details.

## 8.8 Communicator BBC and Teletype emulation modules

### 8.8.1 Overview

This section discusses the Acorn Communicator Teletype and BBC emulation modules version 1.00. It only contains information that is specific to these emulations. For general information on emulation modules, refer to the Emulation Modules section. Teletype and BBC are documented together because they are assembled from the same source files using conditional assembly.

### 8.8.2 Teletype emulation Specification

The Communicator Teletype emulation is a literal emulation of the teletype specification.

Only a restricted set of characters is accepted from the serial line for display on the screen. This set is:

BEL	&07	beep
BS	&08	move cursor backward
LF	&0A	move cursor downward
CR	&0D	return cursor to left edge of the screen
printable characters		
&20..&7E		

Note that the DEL character &7F is not displayed on the screen. This is because many communications services which expect to be communicating with teletype devices use DEL as padding at the beginning of lines.

The Teletype emulation implements echo, newline and linefeed modes. It has no responses.

As with all the current emulation modules, the only error code ever returned in the X register is zero, the MOS default code.

All ASCII characters (ie in the ASCII range &00 to &7F) typed at the keyboard are transmitted onward to the serial line with no modification or filtering. The only exception to this is that in newline mode, CR is expanded to CR LF. Certain Communicator special keys also transmit codes in Teletype mode. These are as follows:

Key	Code transmitted
Tab	&09
Home	&1E
Left arrow	&08
Right arrow	&09
Down arrow	&0A
Up arrow	&0B
DEL	&7F

### 8.8.3 BBC emulation Specification

The BBC emulation module allows the remote system access to the full range of BBC style screen facilities that are implemented by the Communicator VDU drivers and accessible through the VDU character stream.

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. See VDU driver documentation (or in default of that, a BBC manual) for the functions of VDU control codes. VDU 17 and 19 cannot be used if the communications link is configured for Xon/Xoff.

The keyboard is treated in exactly the same way as for the Teletype emulation.

The BBC emulation implements echo, newline and linefeed modes. It has no responses.

### 8.8.4 Data structures

The BBC and Teletype emulations use some local direct page space and separately allocated response block buffers. See the Emulation Modules section for a discussion of common variables. The BBC and Teletype emulations do not use any other variables.

The size of the response blocks in these modules is 2 bytes as the longest response required is the conversion of CR to CR LF.

### 8.8.5 Procedures

This section includes a description of each routine in the specific Teletype and BBC emulation code. See the Emulation Modules section for common parts of the emulation module code.

#### *EMRESET, LocalReset*

This is the main routine that handles the RESET module call reason code. EMRESET checks that it has been passed a good handle and resets the DTD interpretation system.

At this point within EMRESET is the label LocalReset. LocalReset is called from the EMOPEN code to initialise variables. For the BBC and Teletype modules the only variable to be initialised is IEMflag, where echo, newline and linefeed modes are turned off.

The BBC and Teletype emulations do not explicitly set a screen mode, unlike the VT100 and Videotex emulations. Within the Communicator PHONE software, the screen is left in mode 0 on entry to the terminal, so normally BBC and Teletype will run in this mode. However, they do not depend on a particular mode for their operation.

#### *EMSERIN*

This is the main routine that handles the EMSERIN module call reason code. EMSERIN checks that it has been passed a good handle and sets up the local direct page using checkhandle. It moves the serial input response block pointers into lscrptr and llineptr so that sscrput will operate automatically.

In the Teletype module, EMSERIN strips the top bit of the incoming character and then calls checkchar to filter out undesirable characters. In the BBC module this step is simply omitted. If the incoming character is a CR, the code checks whether linefeed mode is on and if so sends the CR followed by LF to the screen driver. Otherwise the character is passed on unchanged to the screen via sscrput.

Finally, EMSERIN calculates the number of characters transmitted to the screen and line and places the results in the response block. BHA is loaded with the address of the serial input response block for return.

EMSERIN uses code that is very similar to the VT100 and Videotex versions. For Teletype and BBC this is an "overkill" because no characters are ever returned to the serial line and the number of characters sent to the screen is always either 1 or 2 and could be calculated more simply. Code simplifications could be made at the cost of loss of similarity with the other emulation modules.

#### *sscrput*

This routine stores the character in A in the serial input response block at the offset lscrptr and increments lscrptr. This handles standard treatment of received characters to be sent to the screen.

#### *checkchar*

checkchar is used by the Teletype emulation module to filter incoming characters. It is passed the character in A. It returns carry clear if the character should be sent to the screen. Acceptable characters are BEL, BS, LF, CR and '..CHR(126).

checkchar is only assembled in the Teletype module.

**EMKEYIN**

This is the routine that handles the EMKEYIN module call reason code. EMKEYIN checks that it has been passed a good handle and sets up the local direct page using checkhandle. It moves the key input response block pointers into lscrptr and llineptr so that keyput will operate correctly.

EMKEYIN calls funccheck to translate special top bit set characters. If the character is valid EMKEYIN checks whether it is a CR and newline mode is on. In this case the CR is expanded to CR LF. Otherwise the character is passed to keyput to be sent to the serial line and optionally echoed to the screen.

EMKEYIN calculates the number of characters transmitted to the screen and line and places the results in the response block. Finally BHA is loaded with the address of the keyboard input response block for return.

**keyput**

keyput deals with a known valid, translated keyboard character. If places the character in A in the keyboard response block line output buffer. If echo mode is on it also places it in the screen output buffer.

**funccheck**

This routine translates keyboard characters to the form in which they will be transmitted to the serial line. The input character is in A and the output is in A. Carry is returned set if the character is invalid. This can only happen if it is an unrecognised top bit set character.

Only top bit set characters are altered; other characters are passed straight through. Top bit set characters are translated through the table FuncTran (see the table in the specification section above).

**EMLnewl, EMLinf, EMLecho**

These routines set or clear bits in the IEMflags word. They are called as a result of DTD commands being processed by the emulation module.

## 8.9 Communicator Videotex emulation module

### 8.9.1 Overview

This section discusses the Acorn Communicator Videotex emulation module version 1.00. It only contains information that is specific to this emulation. For general information on emulation modules, refer to the Emulation Modules section.

### 8.9.2 Videotex emulation Specification

The Videotex emulation module implements a UK Prestel standard Videotex terminal emulation.

This emulation places the Communicator screen in mode 7 on initialisation. 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 accepted characters are:

ENQ	&05	(sends the emulations response string)
APB	&08	active position backward
APF	&09	active position forward
APD	&0A	active position downward
APU	&0B	active position upward
CLS	&0C	clear screen
APR	&0D	active position return
CON	&11	cursor on
COFF	&14	cursor off
APH	&1E	active position home
printable characters		
&20..&7F display control sequences		
ESC &40 through to ESC &5F		

The Videotex emulation implements newline and linefeed modes. However, it is expected that these will be used very rarely as standard Videotex terminals do not have these modes. It has one response, which it transmits in response to ENQ (&05).

As with all the current emulation modules, the only error code ever returned in the X register is zero, the MOS default code.

Keyboard characters (in the ASCII range &00 to &7F) are forwarded to the serial line unaltered, with the following exceptions. The # and pound keys are recoded to their UK Videotex equivalents (&5F and &23 respectively). In newline mode, CR is expanded to CR LF. Certain Communicator special keys also transmit codes in the Videotex emulation. These are as follows:

Key	Code transmitted
Tab	&09
Home	&1E
Left arrow	&08
Right arrow	&09
Down arrow	&0A
Up arrow	&0B
DEL	&7F

The specification for the Videotex emulation module states that it should provide special handling for the \* and # keys on the Communicator numeric keypad, translating them to \* and # rather than the . and = codes that the keyboard driver returns. However the current keyboard driver does not distinguish between these keys and . and = elsewhere on the keyboard. This means that currently these keys are treated specially elsewhere in the Communicator software. Future releases of the software should move the responsibility back to the Videotex emulation module if possible.

### **8.9.3 Data structures**

The Videotex emulation uses some local direct page space and separately allocated response block buffers. See the Chapter 1 for a discussion of common variables.

The other variables in the local direct page are as follows.

#### *IEMflag*

This word is used to hold emulation command flags as follows:

bit 14	linefeed mode if 1
bit 15	newline mode if 1

#### *Isetmode7*

This is a boolean variable that tracks whether the Communicator screen has been set to mode 7 yet. On the very first call to EMKEYIN or EMSERIN after opening the module, the module sends a "set mode 7" sequence to the vdu and sets Isetmode7.

#### *IhadESC*

This is a boolean variable which keeps track of whether the previous character received was an ESC character. This is the only state information required in the videotex emulation. Note that the sequence of characters ESC ESC is invalid. If it is received it is ignored and IhadESC will be left false (i.e. zero).

#### *ICharCount*

This word contains the current cursor position on the Videotex screen. The emulation module needs to track this itself in order to handle the situation where the cursor is moved over the top or bottom of the screen and to prevent the cursor reaching the 25th line. ICharCount counts character positions starting at zero in the top left corner to 959 in the bottom right. For example on line 5 character 23 ICharCount will contain 223.

The size of the response blocks in these modules is 32 bytes. The worst case is that the first character to arrive is an ENQ. This will generate the two character sequence to set mode 7 followed by the response string which is limited to 30 characters by the DTD module command software.

### **8.9.4 Procedures**

This section includes a description of each routine in the specific Videotex code. See the general emulation module documentation for common parts of the emulation module code.

#### *setmode7*

This is a macro which is included in EMSERIN and EMKEYIN to check whether the Communicator screen has been placed in mode 7 and to do it if it has not.

***EMRESET, LocalReset***

This is the main routine that handles the RESET module call reason code. EMRESET checks that it has been passed a good handle and resets the DTD interpretation system.

At this point within EMRESET is the label LocalReset. LocalReset is called from the EMOOPEN code to initialise variables. lhadESC is zeroed (i.e. set false) to show that we have not yet received an ESC character. lsemode7 is set to false.

lCharCount is set to zero, as if the cursor were in the top left of the screen. This will be the case as soon as the "set mode 7" sequence has been sent to the screen driver.

lEMflag is initialised to zero to turn newline and linefeed modes off.

The carry flag status returned by EMRESET is that returned by DTDRST.

***EMSERIN***

This is the main routine that handles the EMSERIN module call reason code. EMSERIN checks that it has been passed a good handle and sets up the local direct page using checkhandle. It initialises the serial input response block pointers in lscrptr and llneptr.

EMSERIN strips the top bit of the incoming character because the UK Prestel Videotex standard uses 7 bit characters. EMSERIN checks for setting mode 7 using the macro setmode7.

Treatment of the received character then depends on whether the previous character was ESC. This state is checked in lhadESC. If there has been no ESC, TransSer is called to carry out some translations (q.v.). If the character is an ESC itself, the lhadESC flag is set and no further action is taken. If the character is ENQ, SendResponse is called to transmit the modules response string. Carriage return needs to be translated into CR LF if linefeed mode is on. After all this treatment, the remaining character (or possibly characters) is sent to the screen by calling sscrput (q.v.) which handles control characters specially.

If the previous character was ESC, only characters in the range &40..&5F are valid; all other characters are thrown away. If the character is valid, it is translated directly up into the range &80..&9F which is how the VDU driver accepts display controls. The character is then sent to the screen.

Finally, EMSERIN calculates the number of characters transmitted to the screen and line and places the results in the response block. BHA is loaded with the address of the serial input response block for return.

***SendResponse***

This routine transmits the response string to the serial line. This may have been set up by the module's R command. The label Rstring has been set to point to this string by the DTDENT macro that defined the R command. Characters are transferred from here to the serial line output block until a control character is found, or until 256 characters have been moved. (Note this should be improved to limit the characters sent to the size of the response block.)

***sscrput***

This routine treats the character in A as a (translated) character to be sent to the mode 7 screen. sscrput deals with both control characters and displayable characters. The principle complication in its implementation is ensuring that cursor movements are carried out as expected by the UK Prestel specification, not as implemented by the mode 7 screen driver.

Control characters are passed to IntCtrl (q.v.) for interpretation.

## *Chapter 8*

Displayable characters are generally simply placed in the screen output buffer, using the macro SPUT. &7F has to be translated to &FF so that it is displayed as a character rather than treated as a delete. As the character is printed, ICharCount is incremented to track the cursor position. If the character is printed in the bottom right hand corner of the 24 line screen, the mode 7 drivers will move the cursor to the first position on the 25th line. Special action therefore has to be taken to move the cursor to the home position.

CdHome, Cls2 and Right are defined within sscrput and are used by the control character interpretation routines.

### *IntCtrl*

This routine handles control characters to be printed on the screen. The code jumps to various handlers through the table CtrlTab. Most controls are invalid and are simply ignored by jumping to Cdnull. The individual routines are described separately below.

### *Goleft*

Goleft sends a backspace character to the screen driver, unless we are currently at the top left of the screen. In this case it moves the cursor to the bottom right of the screen using SETXY.

### *Goright*

This sends an HT character to the screen and then uses the code in sscrput at label Right to check whether the cursor has moved past the bottom right hand corner of the screen.

### *Godown*

Godown adds 40 (VtxLine) to the character count to update ICharCount and then normally just sends a LF character to move the cursor down. If however the cursor would move past the bottom of the screen, a new position on the top line is calculated and SETXY is used to move the cursor there.

### *Goup*

This is similar to Godown. It subtracts 40 (VtxLine) from the character count to update ICharCount and then normally just sends a VT character to move the cursor up. If however the cursor would move past the top of the screen, a new position on the bottom line is calculated and SETXY is used to move the cursor there.

### *ClearScreen*

The UK Prestel standard says that clear screen has the side effect of turning the cursor off, so this is done first. The code then sends a FF to the screen driver and zeros ICharCount via label Cls2 (within sscrput).

### *CRetn*

This handles incoming CR characters. The difficult bit is working out the new value of ICharCount. It does this by calling FindXY to generate the current X and Y values. It then subtracts the X value from ICharCount to create the new ICharCount value. Finally it sends a CR to the screen to move the cursor.

### *CsrOn, CsrOff*

These send the cursor on (ConString) or off (CoffString) strings to the screen driver. They push the offset of these strings and then go to the label ConOff to send it.

ConOff makes the string addressable and then simply copies 10 bytes from the string into the screen output buffer.

***FindXY***

This routine takes a cursor position in A (lCharCount format) and returns the XY position in the X and Y registers. It implements:

```
X := A MOD 40;
Y := A DIV 40;
```

In default of a divide instruction it does this by repeated subtraction!

***SETXY***

This moves the screen cursor to the character position in A (lCharCount format). First, FindXY is called to generate the XY form of the position. Then the &IF gotoxy sequence is sent to the screen driver.

***EMKEYIN***

This is the routine that handles the EMKEYIN module call reason code. EMKEYIN checks that it has been passed a good handle and sets up the local direct page using checkhandle. It initialises the key input response block pointers in lscrptr and llineptr. It checks whether mode 7 has been set yet and sets it if not.

EMKEYIN then calls TransKey to translate some special characters and function keys to their transmitted form. TransKey returns carry set if the key has no translation. If the character is valid EMKEYIN checks whether it is a CR and newline mode is on. In this case the CR is expanded to CR LF. Otherwise the character is passed to keyput to be sent to the serial line.

EMKEYIN calculates the number of characters transmitted to the screen and line and places the results in the response block. Finally BHA is loaded with the address of the keyboard input response block for return.

***keyput***

keyput deals with a known valid, translated keyboard character. It places the character in A in the keyboard response block line output buffer.

***TransKey***

This routine translates keyboard characters to the form in which they will be transmitted to the serial line. The input character is in A and the output is in A. Carry is returned set if the character is invalid. This can only happen if it is an unrecognised top bit set character.

Top bit set characters are translated through the table FuncTran.

Normal displayable characters which need translation are as follows:

Key	Transmitted	character
&23	&5F	hash
&60	&23	pound
&5F	&60	long dash

***TransSer***

This routine translates certain displayable characters to match the UK Prestel character set to the screen driver character set. The translations are:

Incoming	Screen	character
&23	&60	pound
&5F	&23	hash
&60	&5F	long dash

## *Chapter 8*

### ***EMLnewl, EMLInfd***

These routines set or clear bits in the IEMflags word. They are called as a result of DTD commands being processed by the emulation module.

### **8.9.5 OS calls required**

The Videotex code makes no OS calls other than those described in the general emulation module documentation.

## **8.10 Communicator VT100 emulation module**

### **8.10.1 Overview**

This section discusses the VT100 emulation module version 1.00.

It only contains information that is specific to the VT100 emulation. For general information on emulation modules, refer to the Emulation Modules section.

### **8.10.2 Specification**

The overall specification of the VT100 emulation module is as follows.

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 and US character sets
- Graphics character set (partial)
- Set scrolling region
- Set/clear tab stops
- Line feed/Newline mode
- Application cursor mode
- Relative origin mode
- Wraparound on/off
- Report cursor position
- Report status
- What are you?
- Reset terminal
- Local echo mode
- Erase character

## *Chapter 8*

Refer to the DEC VT100 manual for a description of what these features do. 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 one response, which it transmits in response to ENQ (&05). 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

The Communicator Systems manual gives the keyboard mappings for the VT100 emulation module.

Version 1.00 of the emulation module only partly implements "keypad application mode", because the current keyboard drivers are not capable of distinguishing between the numeric keypad and the same character typed on the main keypad. However, in a future implementation of this, the keys should be mapped as follows:

Communicator key	Normal mode	Keypad Application mode
Numeric 1	1	ESC O q
Numeric 2	2	ESC O r
Numeric 3	3	ESC O s
Numeric 4	4	ESC O t
Numeric 5	5	ESC O u
Numeric 6	6	ESC O v
Numeric 7	7	ESC O w
Numeric 8	8	ESC O x
Numeric 9	9	ESC O y
Numeric *./	.	ESC O n
Numeric 0	0	ESC O p
Numeric #/=	=	ESC O M
Copy	:	ESC O l (implemented)
Home	:	ESC O m (implemented)

The remainder of this section gives additional information on how the Communicator VT100 emulation relates to a real VT100.

Communicator has no NO SCROLL key. However, CTRL S and CTRL Q may be used instead if the line is running the Xon/Xoff protocol (very probable).

Communicator has no spare LED's to emulate the LED's on the VT100 keyboard. This means that there is no ONLINE, keyboard locked, L1, L2, L3, or L4 keyboard LED's. If the VT100 emulation is being used through the telephone line, the Communicator LINE IN USE light is equivalent to the ONLINE light.

Communicator VT100 has no SETUP mode. The PHONE directory program has similar capabilities with the following exceptions:

- Tabs must be set from the host computer, there is no equivalent to SET-UP A to set them locally.
- Smooth scroll is not implemented, so cannot be set.
- Autorepeat is a fixed feature of the Communicator keyboard. It cannot be varied within the VT100 emulation.
- Margin bell is not implemented.
- Keypress is not implemented.
- VT52 mode is not implemented.
- The display cannot be set to interlace mode.
- 50Hz/60Hz power selection is not required for Communicator.
- White background screen is not implemented.
- The cursor is always an underline.
- Answerback is not implemented.
- Screen brightness is not controlled from Communicator and so cannot be set from the keyboard.

The following are set up by including options after the emulation module name:

- UK pound (#-) or US # (#+) sign
- wrap at end of line (W+ or W-)
- newline mode (N+, N-)
- local echo (E+, E-)

The following are set up by other options within the directory entry:

- Xon/Xoff protocol
- data bits and parity
- transmission speed

The Communicator VT100 emulation only implements the 80 column display mode. 132 column mode is not available.

A VT100 has various self-test facilities. Communicator has its own self-test which occurs on power up and is not directly related to the VT100 emulation. For this reason, none of the specific VT100 self-test features have been implemented.

Communicator VT100 has no local mode. However, the DISPLAY program provides more advanced capabilities.

ESC # 8 screen alignment is not implemented. This feature does not seem useful in the Communicator environment.

ESC # 3, 4, 5, 6 double height and width are not implemented. These features could be implemented by writing to the screen "graphically" rather than as characters. This would however be slow.

CSI ... x "report parameters" provides a fixed report not reflecting the actual state. The parameter information is not readily available to the emulation module. The values reported are all reasonable and it seems unlikely that any host system software will take any actions on them that will cause problems.

## *Chapter 8*

CSI ... y "invoke confidence test" is not implemented (see self test above).

ECH is implemented even though it is a VT220 function.

DOSU is implemented (dubiously) even though it is an ANSI, not a DEC VTxxx function.

The special graphics character set is not implemented perfectly. At present it uses the best available character from the standard Communicator character set. This could be improved by redefining some of the Communicator character set. This would need either:

a consistent allocation of "spare" characters to modules,

or the character set being saved as part of the screen/keyboard context.

### **8.10.3 Data structures**

The VT100 emulation uses some local direct page space and separately allocated response block buffers. See the Emulation Module section for a discussion of common variables.

The other variables in the local direct page are as follows.

#### *IEMflag*

This word is used to hold emulation command flags as follows:

bit 5	use US character set (display &23 as hash not pound) if set to 1
bit 6	wrap mode on if 1
bit 7	echo mode if 1
bit 14	linefeed mode if 1
bit 15	newline mode if 1

#### *ICurKeyMode*

Bit 7 of this byte is used to determine whether the emulation is in "cursor key application mode". This determines the codes transmitted by the cursor keys. See the Communicator Systems manual.

#### *IOriginMode*

Bit 7 of this byte determines whether incoming cursor positioning commands are interpreted relative to the whole screen (= 0) or to the current scrolling window (= 1).

#### *IKeyAppMode*

Bit 7 of this byte should determine the characters transmitted when keys on the Communicator numeric keypad are pressed. The options are normal numeric values (= 0) or the VT100 keypad application mode sequences. Currently this mode only affects the Copy and Home keys because the Communicator keyboard drivers do not distinguish numeric keypad characters.

#### *IRxState*

This is the major finite state machine state variable for the incoming character stream.

#### *IXpos, IYpos*

These bytes contain the X, Y position of the cursor on the screen. This is maintained by the VT100 emulation itself, and is never read back from the VDU drivers. The top left corner of the screen is 0, 0. The emulation module assumes that the cursor is at the position where it last left it whenever it is entered - it does not explicitly move the cursor on each entry.

***ICSet***

This byte determines whether we are currently printing characters from the G0 or G1 character set (0 = G0, 1 = G1). Which character sets G0 and G1 are defined by IG0set and IG1set.

***IG0Set, IG1set***

These bytes determine which character sets have been mapped into the G0 and G1 sets respectively. There are 12 possible character sets, 0 to 11. These bytes are coded as  $2 * \text{character set number}$ , i.e. 0 to 22 for ease in indexing into a word table. The 12 character sets are:

0	special graphics
1	"alternate ROM" in VT100 terms. This is a special option on a VT100 to provide a special character set. The VT100 emulation module maps this back to the UK set.
2	"alternate ROM" with special graphics. Mapped to set 0.
3	invalid (mapped to UK)
4	invalid (mapped to UK)
5	invalid (mapped to UK)
6	mapped to 0 (Dave Pick convention)
7	mapped to 1 (Dave Pick convention)
8	invalid (mapped to UK)
9	invalid (mapped to UK)
A	UK set
B	US ASCII set (differs only in &23 hash/pound)

***IGxRend***

This byte holds the current "graphics rendition" of characters to be written to the screen, i.e. their attributes. It is bit mapped as follows:

bit 1	bold if 1 (not actioned)
bit 4	underscore if 1 (see note below)
bit 5	blink if 1 (not actioned)
bit 7	reverse if 1 (implemented)

The underscore attribute is OR'ed with the reverse attribute to decide whether characters are displayed normal or inverse on the screen. This is like the behaviour of a VT100 terminal without the advanced video option.

***IMINIX, IMAXIX***

These bytes are set to 0 and 79 respectively on startup and reset and define the (inclusive) left and right edges of the useable screen area. They are in fact never altered anywhere in the code, but are implemented as variables for future flexibility (e.g. in implementing 132 columns).

***IMINIY, IMAXIY***

These bytes determine the vertical extent of the screen which is in use. On startup or reset they are set to 0 and 23 respectively, giving the standard 24 line screen. They may be altered by the "set scrolling region" command sequence.

## *Chapter 8*

### *Itemp*

This is a general purpose temporary variable byte, used in particular by the routine RXCHKK.

### *ICntParams*

A VT100 command sequence typically has the form ESC [ <parameter1> ; <parameter2>... <terminator>. As the emulation module it receives the parameters in such a sequence it counts them in ICnParams. ICnParams is initialised to 0 and during a sequence contains the number of parameters completely received, multiplied by 3. The reason for the times 3 is to index simply into the IParams table.

### *IParams*

(See above for a description of command parameters). IParams is an array of 3-byte parameter entries. As a command sequence is received each parameter is placed in IParams. The format of each entry is two bytes of numeric parameter followed by one byte specifying the type. The type byte has the value 0 for normal ANSI parameters. For private parameters, the type byte contains the private parameter indicator character that preceded the parameter value, typically "?".

### *IRPICNT*

This is a temporary variable used during execution of certain VT100 commands. It is used as a loop count which is preset by the value of the first command parameter. The reason for not using the first element of IParams for this is that in many of the commands, a value of zero (or blank parameter) has to be mapped to 1 before carrying out the loop.

### *ISaveBuf*

When an ESC 7 command is received, the variables IXpos, IYpos, ICSet, IG0Set, IG1Set and IGxRend are saved in this area. When ESC 8 is received the values stored here are copied back to their original positions and the screen is altered to match.

### *ITabSet*

This variable contains 80 bits (i.e. 10 bytes), 1 bit for each column on the screen. If a bit is set to 1 then there is a tab set at that position. Within each byte the columns are numbered from bit 0 upwards. ITabSet is initialised to all zeros, i.e. no tabs set.

### *Isetmode3*

This is a boolean variable that tracks whether the Communicator screen has been set to mode 3 yet. On the very first call to EMKEYIN or EMSERIN after opening the module, the module sends a "set mode 3" sequence to the vdu and resets Isetmode3.

The response blocks for the VT100 module are each 64 bytes long. The longest output sequence is to the screen when it is necessary to clear an area in the centre of the screen. See comment on DOSU below for a potential problem.

### 8.10.4 Procedures

This section includes a description of each routine in the specific VT100 code. See the Emulation Modules section for common parts of the emulation module code.

Note that the VT100 emulation module is assembled in 65816 BYTE mode almost everywhere. Procedure entries are BYTE mode unless explicitly stated otherwise.

#### *EMRESET, LocalReset*

This is the main routine that handles the RESET module call reason code. EMRESET checks that it has been passed a good handle and resets the DTD interpretation system.

At this point within EMRESET is the label LocalReset. LocalReset is called from the EMOPEN code to initialise variables. First, all the local direct page variables from lsetmode3 through to lTabSet are zeroed. Then the routine DORIS q.v. (DO Reset to Initial State) is called to set up correct initial values for various VT100 variables.

#### *EMSERIN*

This is the main routine that handles the EMSERIN module call reason code. EMSERIN checks that it has been passed a good handle and sets up the local direct page using checkhandle. It moves the serial input response block pointers into lscrptr, llineptr and lblock so that the routines VTScrOut and VTSerOut will operate automatically.

On the very first call to EMSERIN after an open, it may be necessary to set up mode 3 on the Communicator screen. SetMode3 is called to check for this. EMSERIN then calls VTSERIN in file VT10004. VTSERIN carries out all the interpretation of the received character.

On return from VTSERIN, EMSERIN calculates the number of characters transmitted to the screen and line and places the results in the response block. Finally BHA is loaded with the address of the serial input response block for return.

#### *EMKEYIN*

This is the main routine that handles the EMKEYIN module call reason code. EMKEYIN checks that it has been passed a good handle and sets up the local direct page using checkhandle. It moves the key input response block pointers into lscrptr, llineptr and lblock so that the routines VTScrOut and VTSerOut will operate automatically.

On the very first call to EMKEYIN after an open, it may be necessary to set up mode 3 on the Communicator screen. SetMode3 is called to check for this. EMKEYIN then calls VTKEYIN in file VT10007. VTKEYIN carries out all the translation of keyboard characters.

On return from VTKEYIN, EMKEYIN calculates the number of characters transmitted to the screen and line and places the results in the response block. Finally BHA is loaded with the address of the keyboard input response block for return.

#### *VTSerOut*

This is used throughout the emulation module as the equivalent of "print a character to the screen". In fact it simply places the character in A in the response block pointed to by lscrptr.

#### *VTSerOut*

This is used throughout the emulation module as the equivalent of "send a character to the serial line". In fact it simply places the character in A in the response block pointed to by llineptr.

**EMLnewl, EMLInfd, EMLEcho, EMLhash, EMLwrap**

These routines set or clear bits in the IEMflags word. They are called as a result of DTD commands being processed by the emulation module.

In addition to simply setting its bit, EMLhash calls DORIS in order to carry out the setup for character set pointers. DORIS has many other effects as well. It would be more satisfactory if EMLhash called a subroutine shared with DORIS that simply altered the character set pointers.

**GETXY, SETXY**

GETXY is a macro used to load the X and Y registers with the X and Y positions of the cursor on the screen. SETXY is a macro which moves the real cursor to the position indicated by the variables IXpos and IYpos.

**REPPI, EATPI**

These macros are used to form a loop on the first parameter of a command sequence. REPPI loads the value of the first parameter into the temporary variable IRPICNT and defines the label \$reppi1 as the top of the loop. EATPI is used at the end of the loop. It decrements IRPICNT and loops if this is non-zero. These macros are used in various of the following command implementations.

**VTSERIN**

This routine processes a character received from the serial line. It strips the top bit of the character because the VT100 emulation operates in a seven bit environment. This saves additional checks later. It then filters out DEL and NUL characters which a VT100 totally ignores. This means that they do not interfere with any command sequence which is being received.

VTSERIN and in fact the whole interpretation of received characters is done through a finite state machine. The state of the machine is in lRxState, which takes only even values for convenience. VTSERIN branches out to different processing routines depending on the current state, through the table RxTable. Note that CASE is a macro defined in emlmacros.

The values of lRxState and their meanings are:

0	initially or after any normal printable or control character has been received
2	ESC received from state 0
4	ESC ] received from state 0 Note that this is the state while parameters are being accumulated.
6	ESC % (SCI) received from state 0
8	ESC ( (SCS) received from state 0
10	unrecognised ESC <1 class character> sequence, wait for <F class character> before reverting to state 0
12	ESC ] (OSC) received from state 0
14	ESC ^ (PM) received from state 0
16	ESC _ (APC) received from state 0
18	ESC P (DCS) received from state 0
20	received ESC as the first character of an ST (ESC \) sequence while in a string state (OSC, PM, APC or DCS)
22	ESC # received from state 0
24	ESC ) (BSCS) received from state 0

Further explanation of each state is provided by the documentation for the routines that process them.

**NormChar**

This routine is used by VTSERIN in state 0 to process a printable character or normal (C0) control character. Printable characters in the range &20 to &7E are passed to the routine DISPGC for display.

Control characters are processed through the case table CtrlTab.

Invalid control characters are just passed to an RTS and thereby ignored.

BEL (&07) results in a BEL character being passed on to the screen driver. Backspace (&08) is implemented by setting parameter 1 to 1 and then calling the code that processes "cursor backward" (DOCUB).

LF (&0A), VT (&0B) and FF (&0C) are all handled the same way by a VT100 - as a line feed. First DOIND is called to move the cursor down one line. If linefeed mode is on the cursor is then moved to the left edge of the screen using the CR processing code.

CR (&0D) moves the cursor to the left edge of the window, both in IXpos and on the real screen.

ESC (&1B) simply changes the internal state to 2 ready for further sequence interpretation.

SO (&0E) and SI (&0F) select the G1 and G0 character set respectively. This simply involves setting the variable ICSet appropriately - this takes effect when a character is printed through DISPGC.

HT (&09) moves the cursor to the next tab setting or to the right margin if there are no more tabs on the line. Note that by default there are no tabs set at all. Before HT becomes useful, tabs must have been set by a host sequence. ICOHT pushes a 3-byte pointer to the table "Binary". This translates an index from 0 to 7 into a bit mask.

ICOHT mainly consists of a loop that increments the X position and then checks whether there is a tab there. The loop finishes if a tab is found or if the right margin is reached. When the loop finishes it pops off the Binary address from the stack and leaves via SETXY which sets up the new cursor position.

#### *HadEsc*

This routine is used by VTSERIN in state 2 to process the next character received after ESC. First, the character is classified by RXCHKK q.v. If it is less than &20 it is thrown away and the state reverts to 0. Otherwise the code jumps to various routines through ICIFTTable according to the character value.

Many of the possible characters in the range &20 to &2F ("I" characters) are not recognised. These go to the label ICIIIBad which sets the state to 10. State 10 is then used to wait for the end of the unrecognised sequence before reverting to normal character interpretation. Other characters simply cause a change of state as they introduce particular command sequence types. See state table values above for these values.

ESC E and ESC D do newline and index respectively using routines ICOOCR and DOIND. ESC H sets a tab at the current cursor position using routine TabSC q.v. ESC M is reverse index. Normally this simply moves the cursor upwards. However, if the cursor is at the top of the screen the whole screen is scrolled down, regardless of the current scrolling window setting.

ESC c reinitialises the emulation module using DORIS.

ESC = and ESC > set and reset keypad application mode respectively by setting the variable IKeyAppmode. This takes effect within the keyboard processing code.

ESC [ is a CSI sequence which introduces a series of parameters. The state is changed to 4, the whole parameter table is cleared to zeros and the parameter count is zeroed.

ESC 7 saves the cursor position, graphic rendition and character set information. All of these variables are positioned contiguously within the local direct page area. Saving them is therefore a matter of copying them to ISaveBuf. ESC 8 restores the values of these variables. This involves copying them back from ISaveBuf, using GOTOXY to actually move the cursor and using SetGR to set up the graphics rendition again.

#### *TabSC*

This routine is called from various places to set or clear a tab setting. It is entered with A = tab position in the range 0 to 79 and carry set to set a tab, clear to clear a tab.

## *Chapter 8*

### *HadHash*

This routine is used by VTSERIN to process the next character received after ESC#. In fact no ESC# commands are implemented, so the code simply reverts to state 0.

### *HadCSI*

This routine is used by VTSERIN to process characters received after the CSI sequence ESC [. The characters may be part of a parameter value, a parameter separator, a private parameter marker or the sequence terminator.

First the incoming character is classified by RXCHKK q.v. Any character >= &40 is a valid terminator, so in this case the code branches to EndCSI. EndCSI sets the state back to zero then does a case jump through the table CSITAB. The routines in CSITAB actually implement the CSI command sequences. Many of the terminators do not constitute a recognised command. These are pointed at DONOWT which simply executes an RTS.

If the incoming character is < &30, it is invalid. In this case the CSI sequence is thrown away and the state is set back to zero immediately.

Characters in the range &30 to &3F are part of parameters. Digits (&30 to &39) are added into the current parameter value using routine DECDIG q.v. ":" or ";" are parameter separators. They cause the code to move on to the next parameter and add one to the parameter count. The other characters in the range &30 to &3F, <, =, > and ? are private parameter markers. If one is received it is stored in the third byte of the parameter area.

### *DECDIG*

This routine is called from with HadCSI to add a digit to the current parameter value. It performs the operation:

```
current parameter = current parameter * 10 + A
```

Note that current parameter is a word value. On entry X already points to the current parameter.

### *HadSCI*

This routine is used by VTSERIN after ESC % has been received. The Communicator emulation module does not implement any Special Character Interpretation. HadSCI classifies the incoming character and resets the state to zero once a terminating character has been received.

### *HadSCS, HadBSCS*

These routines are used by VTSERIN after ESC (or ESC) respectively have been received. They map a particular character set into the G0 or G1 set respectively. If another "I" character is received, the code does not reset the state but remains waiting for a valid terminator.

When a valid terminator is received it is passed to CHKSCS for checking q.v. If it is indeed valid the character set reference is stored in IG0set or IG1set.

### *CHKSCS*

CHKSCS verifies that the final character of an SCS sequence is valid, ie is in the set "0123456789AB". The character is passed in A. If the character is valid carry is returned clear, otherwise carry is set. A valid character is converted into a "character set reference" in A which is an even value in the range 0 to 22 (2 \* position of the character in the set).

***HadBadI***

This routine is used by VTSERIN after ESC x where ESC x is not a recognisable sequence. If another "I" character is received, the code does not reset the state but remains waiting for a valid terminator. Once a terminator is received the state is reset to zero.

***HadOSC, HadPM, HadAPC, HadDCS***

These routines are used by VTSERIN after ESC ], ESC ^, ESC \_ or ESC P respectively have been received. None of these commands have any action on Communicator. They are all string commands which are terminated by the String Terminator ST (ESC \). The code therefore simply waits to receive ESC and then moves to state 20 to wait for the \.

***WaitST***

This routine is used by VTSERIN after the ESC which should introduce an ST sequence to terminate OSC, PM, APC or DCS. The character received should be \, but there's not much the code can do if it isn't, so it always reverts to state 0.

***DOCUB***

This routine moves the cursor backwards. It is branched to from HadCSI and also used to implement backspace (BS). It loads the current cursor position and then decrements the X position until the left margin is reached or the repeat count is exhausted.

***DOCUD, DOCUF, DOCUU***

These routines move the cursor down, forward and up. They are branched to from HadCSI. They operate in a very similar way to DOCUB, with the appropriate margin limit/ X/Y position substituted.

***DOCUP, DOHVP***

This is a single routine which repositions the cursor. It is branched to from HadCSI. DOCUP and DOHVP are ESC [ ... H and ESC [ ... f respectively, but their effects are identical. Note that the parameters in the commands number rows and columns on the screen from 1, whereas the module internal variables number them from 0.

First the code loads the old cursor position. If either of the X/Y positions is invalid, the old value is preserved. Processing for each of the X and Y positions is very similar. The parameter is loaded and normalised to be based at zero. The code checks whether we are using absolute positioning mode or "origin mode" to determine whether the parameters should be interpreted as relative to the scrolling margins. The code checks that the new position is within the screen area and rejects it if not.

Finally, the new X/Y position is set on the actual screen.

***DOSGR***

This routine is branched to from HadCSI and sets the "graphics rendition", i.e. whether subsequent characters are displayed bold, underscored, blinking or reversed. The routine tracks the state of all of these even though the Communicator screen is not able to implement them all.

A single SGR command may set and/or clear various attributes. The main loop in DOSGR works through each parameter that has been sent. Private parameters and parameters with a value greater than 7 are rejected.

The character attributes are stored within a single byte IGxRend. A parameter value of zero clears all the attributes. The bit positions within IGxRend are arranged such that the parameter value used to set the attribute is the same as the bit number where the attribute is stored. This allows the use of the mask array Binary to calculate the bit to set.

## *Chapter 8*

Once all the parameters have been processed the code exits through SetGR to action any changes which have been made.

### ***SetGR***

This routine is entered from DOSGR and DORIS. It takes the current character attributes in IGxRend and sets up the Communicator screen driver to display subsequent characters appropriately. The only Communicator screen attribute that the emulation module uses is inverse, i.e. black on white. This is set if either the reverse or underscore attributes are on. This is like the behaviour of a VT100 terminal without the advanced video option.

### ***DOSTBM***

This routine is branched to from HadCSI and sets the scrolling margins. This is a DEC specific command ESC [ ... r.

The parameters are the top and bottom of the scrolling region. The scrolling region always extends over the whole width of the screen. The parameters are normalised to be based on 0 not 1. They are then checked that the bottom is within the screen and that the top is above it. If all is well the new scrolling region is set.

### ***DODA, DODAX***

These routines return a device attributes report to the host system. DODA is branched to from HadCSI when the sequence ESC [ ... c is received. DODAX is branched to from HadESC when ESC Z is received. DODA checks that the parameter value is zero and is not a private parameter before taking action.

The code then returns the fixed response string to the communications line:

ESC [ ? 1 ; 0 c

This is the string returned by a base VT100 terminal.

### ***DODSR***

This routine returns a device status report to the host system. DODSR is branched to from HadCSI when the sequence ESC [ ... n is received.

The valid values for parameter 1 are:

5	report status
6	report cursor position

Private parameter values are not valid and no action is taken on them.

The status report is a fixed string indicating that no errors have occurred:

ESC [ 0 n

The cursor position report is:

ESC [ <Y position> ; <X position> R

Note that the Y position as transmitted here runs from 1 to 24 and the X position runs from 1 to 80.

**DOREQTP**

This routine returns a report of the terminal parameters to the host system. DOREQTP is branched to from HadCSI when the sequence ESC [ ... x is received. This is a DEC private function.

The emulation module returns a fixed string to the communications line which says "this is a report and the terminal is only reporting on request, the terminal parameters are no parity, 8 bits, speeds are 9600 bits/second, clock multiplier is 16, flags 0". The string is:

**ESC [ 3 ; 1 ; 1 ; 112 ; 112 ; 1 ; 0 ; x**

This may well not actually be true of course. However, there is no mechanism for the emulation module to find out such things as the current bit rate. In future implementations of the Communicator software, using X.25 communications, many of these parameters are not even meaningful. The values have been chosen as being typical for a VT100 terminal and unlikely to cause difficulties for the user if the host takes any action on them.

**TXPS**

This subroutine takes a parameter value in A, converts it into decimal ASCII form, transmits it to the communications line and then transmits a ";" separator character.

NOTE TXCSI, TXSEP and TXPAR are documented here even though they are actually in file VT10007

**TXCSI**

This subroutine simply transmits the CSI sequence ESC [ to the communications line.

**TXSEP**

This subroutine transmits ";" as a parameter separator to the communications line.

**TXPAR**

This subroutine accepts a value between 0 and 255 in A. It converts it to decimal ASCII form and transmits the ASCII characters to the communications line. It is used for transmitting parameter values.

The implementation carries out the division by repeated subtraction and explicitly checks for values less than 10. It calls itself recursively to calculate each digits. This is fairly crude, but adequate for the current VT100 emulation.

**DOECH**

This routine is branched to from HadCSI on receipt of ESC [ ... X. It erases a number of characters on one line of the screen to blank. This is actually a VT220 facility rather than VT100.

It prints spaces to the real screen until the repeat count in parameter 1 is exhausted or the right margin is reached. It then moves the cursor on the real screen back to its original position.

**DOED**

This routine handles "erase in display" and is branched to from HadCSI on receipt of ESC [ ... J. The first parameter value determines the precise action:

- |   |                            |
|---|----------------------------|
| 0 | erase to end of screen     |
| 1 | erase from start of screen |
| 2 | erase the whole screen     |

## **Chapter 8**

Private parameter values and parameter values greater than 2 are rejected.

All of the erase routines are implemented by setting a window in the Communicator's screen driver and then sending a FF character to clear the whole of the window. In the case of "erase whole screen" this is straightforward; just set the window to the full screen and send FF. For "erase from start" and "erase to end" it is done in two stages - first erase the part of the current line and then erase the rectangular area above/below the current line. Erasing within the line uses the routines DOEL0 and DOEL1. Erasing large areas uses EraseWindow.

At the end of any of these operations the cursor is restored to its original position.

### **DOEL**

This routine handles "erase in line" and is branched to from HadCSI on receipt of ESC [ ... K. The first parameter value determines the precise action:

- |   |                          |
|---|--------------------------|
| 0 | erase to end of line     |
| 1 | erase from start of line |
| 2 | erase the whole line     |

Private parameter values and parameter values greater than 2 are rejected.

These are implemented by setting a Communicator screen window on the area of the current line to be erased and then sending FF to clear it. The subroutine SetLWindow is used to set the window. The label EraseX sends the form feed, restores the cursor to its original position and then sets the window back to full screen.

Note that the label OldWindow is used elsewhere as an entry point for restoring the full screen window. Throughout the operation of the VT100 emulation module, the Communicator screen window is only temporarily reduced; normally it is set to full screen. The emulation module assumes that the window is at full screen whenever it is entered.

DOEL0 and DOEL2 are used as subroutines by DOED (see above).

### **SetWindow**

This routine is used to set up a Communicator screen window that extends all the way across the screen but is limited vertically. It is entered with A equal to the top line number (inclusive) and X equal to the bottom line number (inclusive) of the window.

It is implemented using the VDU 28 character sequence.

SetWindow is used by the "erase in screen" routines and by various parts of the screen that carry out scrolling.

### **SetLWindow**

SetLWindow sets a Communicator screen window within the current line. On entry A contains the left column position (inclusive) and X contains the right column position (inclusive) of the window. It is implemented using the VDU 28 character sequence.

SetLWindow is used by the "erase in line" routines.

### **DOIND**

This routine handles the VT100 index (move cursor down and scroll if necessary) command. It is branched to from HadESC on receipt of ESC D. It is also called from the code that handles incoming VT, FF and LF, from DOSU and from DISPGC.

If the cursor is on the 24th line, DOIND sets up a 24 line window on the screen and then sends a LF, causing a scroll. It restores the full screen window before exiting. If the cursor is not on the 24th line, DOIND simply sends a LF character to the screen driver and increments the Y position.

### **DORIS**

This routine re-initialises the VT100 emulation module. It is called from EMRESET, LocalReset (and therefore from EMOPEN) and is branched to from HadESC on receipt of ESC c.

DORIS sets up a screen size of 24 by 80 by setting:

```
IMINIX = 0
IMINIY = 0
IMAXIX = 79
IMAXIY = 23
```

IMINIX and IMAXIX then remain set to 0 and 79 throughout the operation of the module. IMINIY and IMAXIY may be altered to reflect the current scrolling region size.

Both the G0 and G1 sets are set to be the standard UK or US set. The choice between UK and US is made on the basis of the "hash" flag in IEMflags.

Finally all the tab settings are cleared.

### **CirTabs**

CirTabs removes all tab settings by setting all the tab array bytes to zero. It is called from DORIS and from DOTABS on receipt of ESC { 3 g.

### **DORM, DOSM**

These routines reset and set VT100 operation modes respectively. DORM is branched to from HadCSI on receipt of ESC [ ... l. DOSM is branched to from HadCSI on receipt of ESC [ ... h. DORM and DOSM are largely handled by the same code, except that DORM pushes 0 on the stack and DOSM pushes &FF on the stack before starting the main loop.

A DORM/DOSM command can reset or set many modes. There is therefore a main loop which looks at the received parameters one by one. The parameter is first checked. If its value is greater than &FF it is rejected. Any "Acorn" private parameters (marked by >) are ignored. The only standard parameter value which is recognised is 20 which sets or resets both newline and linefeed mode. This is implemented in the subroutine SRNLF.

There are three DEC private modes, introduced by ?. These are handled by the subroutine PrivPars. This subroutine simply branches to the various processing routines through the case table ParTable. The branch is made with A = 0 for reset and A = &FF for set.

### **SRAWM**

This routine handles setting or resetting automatic wrap mode. This mode determines whether the cursor wraps to the next line when the 80th character on a line is printed. SRAWM is branched to from PrivPars. SRAWM simply alters the flag bit in IEMflags.

### **SRCKMode**

This routine handles setting or resetting cursor key mode. This mode determines the character sequence transmitted by the cursor keys. SRCKMode is branched to from PrivPars. SRCKMode stores A in ICurKeyMode for later reference.

### **SROrigin**

This routine handles setting or resetting origin mode. This mode determines whether cursor positioning commands are interpreted relative to the whole screen or to the current scrolling margins. SROrigin is branched to from PrivPars. SRCKMode stores A in lOriginMode for later reference. If we are setting "relative mode", SROrigin also moves the cursor to the top left hand corner of the current scrolling region.

### **DOSU**

DOSU scrolls the screen upwards the number of times specified in the first parameter. DOSU is branched to from HadCSI on receipt of ESC [ ... S. DOSU saves the current cursor position, moves the cursor to the bottom of the screen and then calls DOIND repeatedly to carry out the scroll.

SU is an ANSI standard command which is not implemented by any of the VTxxx terminals up to VT220. It is retained in the emulator because it was present in the original Master terminal code, but is not a part of the Communicator product specification. It is currently implemented rather inefficiently. It is also dangerous because the character sequence generated could be longer than the space reserved for the emulation module response block. YOU HAVE BEEN WARNED!

### **DOTABS**

DOTABS clears the tab at the current cursor position or all the tabs. It is branched to from HadCSI on receipt of ESC [ ... g. The parameter can be 0 which means clear the tab at the cursor position or 3 to mean clear all tabs. Private parameter values result in no action.

The subroutine ClrTabs is used to clear all the tabs. The subroutine TabSC is used to clear a single tab position.

### **DISPGC**

This routine displays the character in A on the screen. A is guaranteed to be in the range &20 to &7E. DISPGC is called from NormChar. DISPGC is responsible for displaying the character with regard to:

whether the G0 or G1 character set is active

which set G0 or G1 is moving the cursor after displaying the character, particularly being concerned about line wrap mode

DISPGC first calculates the translated form of the character. The actual character set to use is calculated by using ICSet as an index into IGOset/IG1set to load the character set identifier (an even number in the range 0 to 22). This character set identifier is used as an index into the table TTabs to retrieve the relative address of one of the character translation tables UKSet, USSet or SGSet. Some further stack arithmetic adds the character itself to this address, allowing the character's translation to be loaded into A.

At this point the character is normally simply printed and the X position is incremented. However, we have to behave specially at the end of a line. If wrap mode is off (the default), the character is printed and then the cursor is moved back where it was. This means that repeated characters at the end of the line overwrite each other.

If wrap mode is on, the character is printed, the screen cursor is explicitly moved to the beginning of the current line and then DOIND is used to move down to the next line. DOIND handles the possible requirement to scroll.

**UKSet, USSet, SGSet**

These are the character set translation tables.

USSet translates the incoming US ASCII characters to Communicator's character set. The only translation needed to translate &60 to &BB to display &60 as a backward quote rather than pound. UKSet differs from USSet only in that &23 is displayed as a pound rather than a hash character.

SGSet is an approximation to the VT100 special graphics set. It is not perfect - see emulation limitations section above for details. In particular the following characters are incorrect:

Hex	VT100 graphic	Communicator graphic
&62	HT	right arrow
&63	FF	paragraph sign
&64	CR	left arrow
&65	LF	down arrow
&68	NL	n tilde
&69	VT	up arrow
&6F	line scan 1	upper right corner
&70	line scan 3	upper right corner
&72	line scan 7	underline

**GOTOXY**

This routine moves the actual screen cursor to the position defined by the local variables IXpos and IYpos. It falls through into SETXY.

**SETXY**

This moves the actual screen cursor to the position given by the X and Y registers. It also sets the local variables IXpos and IYpos. It transmits a VDU 31 X Y sequence to the screen driver.

**RXCHKK**

This takes a character in A and returns a classification of it in the flags C, N and V. No registers are affected, though RXCHKK uses the local direct page temporary ltemp. The classification results are:

	C	N	V	
&00 <= A <= &1F	0	0	0	for a (C0) control character
&20 <= A <= &2F	1	0	0	for an I (intermediate) character
&30 <= A <= &3F	1	1	0	for a Fp or P character
&40 <= A <= &FF	1	1	1	for a Fc or Fs or F character

RXCHKK is used by many of the incoming character processing routines.

**PISET1**

This sets the first parameter to 1 and defines it as a standard parameter. This is used to set up for a single repeat before calling certain general routines that repeat on the first parameter.

**GETPI1, GETP2**

These routines load parameter 1 or 2 respectively into A. Carry is set if the parameter is a private one. If the parameter value is greater than 255, the value returned is 255 to indicate an overflow. These two routines are implemented using the macro GETPX.

## *Chapter 8*

### ***REPPI***

This routine sets up the variable IRPICNT ready for code that repeats on the value of the first parameter. Parameter values of zero and private parameters are translated to a value of 1. Typically a parameter value of zero results from a missing parameter.

### ***VTKEYIN***

This routine processes a character received from the keyboard. Keyboard characters with their top bit set are special characters. See the Emulation Modules section for the mapping of special keyboard characters to codes.

Standard characters are passed on the communications line through the routine EchoOut q.v. The CR character may be translate to CR LF under control of the newline mode flag.

Top bit set keys are translated twice through the tables KBATAB and KBJTAB. KBATAB translates the character into an action code. KBJTAB is a series of branches to processing routines. (NOTE this could be recoded using the CASE macro).

### ***CommaKey***

This routine is called when the COPY key is pressed. This is regarded as the VT100 keypad comma key. In keypad numeric mode this simply sends comma. In keypad application mode it transmits ESC O 1.

### ***DashKey***

This routine is called when the HOME key is pressed. This is regarded as the VT100 keypad dash key. In keypad numeric mode this simply sends dash. In keypad application mode it transmits ESC O m.

### ***PF1, PF2, PF3, PF4***

These routines are called when SHIFT F1, SHIFT F2, SHIFT F3, SHIFT F4 respectively are pressed. They are regarded as the VT100 PF1, PF2, PF3 and PF4 keys. They simply transmit ESC O followed by P, Q, R and S respectively.

### ***ESCOplus***

This routine transmits ESC O followed by the byte in A to the communications lin.

### ***UpArrow, DownArrow, RlArrow, LfArrow***

These routines are called when the associated cursor keys are pressed. The character sequence transmitted depends on "cursor key application mode" stored in the ICurKeyMode flag.

### ***EchoOut***

EchoOut sends the character in A to the serial line. If echo mode is on, it also transmits the character back to the screen.

## **8.11 OS calls required**

The VT100 code makes no OS calls other than those described in the general Emulation Modules section.

## **9. Application Escape**

### **9.1 Introduction**

Application Escape provides access for user developed applications to the built-in system phone and terminal facilities. This enables OEMs to incorporate their own applications into on-line terminal sessions.

Section 2 gives an overview of the facilities provided by Application Escape.

Section 3 describes the interface to Application Escape and the exit conditions.

Section 4 covers the responsibilities of the OEM application.

Section 5 describes TSHELL's validation actions on return from the OEM application.

Section 6 Covers close down of TSHELL and error codes returned by TSHELL to the user.

For more information on Application Escape and how it relates to the rest of the terminal facilities refer to the Communicator Systems Manual.

### **9.2 Overview of facilities**

The application escape is triggered either by a function key being hit (F8 'Leave' from the terminal module), or on answering a call when the modem is in auto answer mode. When this happens, TShell searches for a Communicator module called PhoneEsc. This module will typically have been loaded into RAM memory at boot time as a user module.

If PHONEESC is not found, TSHELL does absolutely nothing, just as if an invalid F key had been pressed. If PHONEESC is found, it is entered as a module and passed the following information:

the handle for the communications interface (ie Modem or RS423),

the handle for the emulation module,

the handle for the printer driver.

The PHONEESC module can then completely take over operation of the call if it so wishes. When PHONEESC returns, TSHELL may continue the call just as though nothing had happened, or close down if required. PHONEESC is responsible for leaving the screen in a suitable state on exit.

## 9.3 Interface description

### 9.3.1 Entry conditions

If keypage is sending a file, PHONEESC is ignored.

If keypage has session spool on, further capture of data will cease upon entering PHONEESC and be resumed on return to TSHELL.

BHA	==>	[status] 1 byte [no. of handles] 1 byte [pointer to text] 3 bytes [line handle] 2 bytes [eml. handle] 2 bytes [printer handle] 2 bytes
status byte:		0 = invoked by user 1 = invoked by autoanswer call
number of handles		Will have value of 3.
pointer to text		On entry will point to a return terminated null message. If required, PHONEESC can return a pointer to a message, which will be printed when TSHELL closes down PHONEESC AND the call.

### 9.3.2 Exit conditions

BHA	==>	[status] 1 byte [no. of handles] 1 byte [pointer to text] 3 bytes [line handle] 2 bytes [eml. handle] 2 bytes [printer handle] 2 bytes
status byte		0 = kill PHONEESC coroutine and continue 1 = don't kill PHONEESC, just continue 2 = kill PHONEESC and close down TSHELL/KEYPAGE
no. handles		Should equal no. handles on entry.
pointer to text		May have been modified by PHONEESC if required.
line, emulation, printer		These may have been changed by PHONEESC. If the line or emulation handles are zero, TSHELL will close the call down. If the handles are valid, they will be used, instead of the handles passed to PHONEESC on entry.

Assuming the handles are valid, TSHELL and KEYPAGE will adopt these new handles. The printer handle can be 0 either on entry or exit, (or on entry and exit). KEYPAGE will respond as required:- opening a new channel if required, or continuing with the old handle, or closing down the print.

## 9.4 Responsibilities of the PHONEESC module.

PHONEESC must ensure that it returns to TSHELL in a clean state. To achieve this, consideration must be given to:

- (a) Screen and Context information restored suitably for TSHELL. (This might require saving and restoring screen and context information, or just selectively ensuring that TSHELL is returned to in the state in which it was left, eg screen mode, function key base values, escape disable/enable).
- (b) Any memory allocated must be de-allocated.
- (c) Any device drivers or files opened must be closed.
- (d) If required it should provide its own HELP information.

With the exception of (d) all of the above must be encapsulated in a routine, called when the coroutine is entered with reason code in X = 'ENKILL'. This code will be called by the MOS as a SUBROUTINE, and the routine should therefore finish with an RTL instruction.

## 9.5 Validation of PHONEESC return state by TSHELL

TSHELL always reserves the right to close down the call if it receives invalid information from PHONEESC. The validation performed is as follows:

If the line or emulation handles are 0 then PHONEESC is killed and TSHELL closed down.

If any of the handles are odd then they are treated as invalid. The relevant odd handles will be used, PHONEESC is killed, and TSHELL closed down.

If TSHELL does not receive the expected number of handles then PHONEESC is killed and TSHELL closed down.

If the handles are valid then the following operations are performed:

Assign the line handle returned by PHONEESC to TSHELL's copy of the handle, updating the control handle as well. KeyPage is then informed of the new handle.

Assign the emulation handle returned by PHONEESC to TSHELL's copy of the handle, updating the control handle as well. KeyPage is then informed of the new handle.

Assign Printer handle to TSHELL's copy of the handle. KeyPage is then informed of the new handle.

### 9.5.1 Killing PHONEESC

PHONEESC may be killed,

EITHER: because it requested to be killed,

OR: because TSHELL received invalid data and decides to close down.

OR: because PHONEESC returned with carry set. If this occurs, BHA is taken to point to a return terminated error message (and NOT the control block). TSHELL will closedown reporting error code &00 and displaying the returned message. PHONEESC should only return carry set if it cannot startup, ie couldn't get direct page, or workspace etc. In this situation the coroutine will be deleted only. ie the coroutine will not be entered with reason code fENKILL.

## **9.6 Closedown**

### **9.6.1 Normal closedown of TSHELL with PHONEESC coroutine active**

If the user presses F4 (End Call), or any other TSHELL error condition occurs (eg carrier lost) then any active PHONEESC coroutine will first be killed.

### **9.6.2 Reporting what has happened**

If TSHELL closes down for any reason whilst handling an Application Escape, the user will be informed as follows:

- a) PHONEESC requests to be killed and to close TSHELL:  
"Data call terminated due to the Application Escape closing down."  
<message returned by PHONEESC>"
- b) TSHELL receives invalid data from PHONEESC, and decides to close down:  
"Data call terminated due to an error in the Application Escape."  
"(Error code: &xx)"  
<message returned by PHONEESC>"

An error code will be returned to allow OEM developers to track down problems with their code. A more detailed error message is not given as it would be meaningless to the intended end users if ever it occurred. The error code is a bit pattern. Setting a particular bit indicates a certain type of error has occurred. It is possible for more than one error to have occurred.

#### ***ERROR CODES returned by TSHELL to the user***

<b>bit0</b>	An invalid status value was returned. (i.e. a number other than (0,1,2).
<b>bit1</b>	The parameter reporting the number of handles had an incorrect value. For the current implementation, this value should be the same on exit as on entry, and should be three.
<b>bit2</b>	An invalid line handle was returned. The handle was either 0, or it was an odd number.
<b>bit3</b>	An invalid emulation handle was returned. The handle was either 0, or it was an odd number.
<b>bit4</b>	An invalid Printer handle was returned. i.e. it was an odd value ( a 0 Printer handle is quite valid)
<b>bits 5,6,7</b>	Reserved for future expansion

# 10. Macros and definition files

## 10.1 Introduction

This Chapter describes the purpose and use of the various "GET" files supplied as part of the OEM developer's package.

GET files contain information which a programmer may find useful to include in his own source. This information consists of macros and/or manifest constants.

A macro is a set of group of instructions, (assembler mnemonic or directive), that can be inserted into a piece of source code by citing its name as an instruction - known as "invoking" the macro.

A manifest constant is a fixed value referenced by name.

Macros serve a number of useful functions. They can be used to implement small but frequently encountered sets of assembler mnemonics which do not or must not require the overheads associated with a subroutine call.

Useful but non-existent processor instructions can be implemented as a set of standard instructions invoked as a macro call.

Macros can be used to provide access to complex data structures without the user needing to know the details of their actual construction. By "hiding" underlying structures in this way, changes and upgrades can be effected without users needing to alter their source code. Simply re-assembling with the new macro definition suffices. An example of this type of macro is the system macro START used to build module headers.

In a similar manner, manifest constants are used to represent absolute values symbolically. By referencing constants via a symbolic definition, changes to absolute values do not imply changes to the content of the user's source code. Again re-assembly with the new definitions is all that is required.

GET files are included in source files via the GET assembler directive which takes the form:

GET      \$filename

where \$filename is a string specifying the location of the required file. GET directives should appear at the beginning of the source code. This is due to problems with the current release of MASM.

Alternatively, a GET file can be included at assembly time from the assembler command line.

## 10.2 The SYSTEM Macros

### 10.2.1 Introduction

The SYSTEM macros are probably of the greatest importance to the OEM developer.

The SYSTEM macros comprise a set of common system operations and global constant definitions.

The SYSTEM macros enable the OEM developer to interface applications written in assembler to the Communicator MOS and many of the built-in functions and device drivers.

The operation macros can be grouped as follows:

- Pseudo-opcodes -** These implement useful but non-existent 65816 instructions, or in some cases existent but incorrectly implemented by the assembler.
- Mode Directives -** These place the processor and/or the assembler into different combinations of 8- or 16-byte mode operation.
- Module Macros -** These support the creation of Communicator Module Headers.
- OS Macros -** These simplify access to operating system functions.

The global definitions, or manifests, declare various system constants such as module entry reason codes, operating system COP function values, system data structures etc.

The system macros are distributed amongst the following files:

SYSTEM  
SYS2  
SYS2A  
SYS3  
SYS4

All of these file are linked, so the developer's source need only GET the file \$S.get.system.

### 10.2.2 The SYSTEM Operational Macros

The SYSTEM file contains all of the system operational macros along with the SYSTEM macro which implements all of the manifest macros.

#### *Pseudo-Opcodes*

The SYSTEM pseudo-opcodes fall into two categories; those that replace non-working mnemonics within the MASM assembler and those that simulate desired operations not found in the 65816 instruction set.

#### *Replacement Macros*

The following macros replace existing MASM mnemonics:

(\$label)	(\$)STYZ	\$value	replaces STYZ
(\$label)	(\$)LDYZ	\$value	replaces LDYZ
(\$label)	(\$)BITZ	\$value	replaces BITZ

The operand \$value must be an 8-bit direct page offset.

#### *New Mnemonic Macros*

These pseudo-opcodes are intended to augment the standard 65816 instruction set.

The BSR macro provides a branch long relative to subroutine mnemonic.

(\$label) BSR \$address

The following macros extend the addressing range of the branch-on-condition relative instructions which are normally limited to a signed 8-bits offset:

(\$label)	BEQUAL	\$address	Branch if EQUAL
(\$label)	BNEQ	\$address	Branch if NOT EQUAL
(\$label)	BCCL	\$address	Branch if CARRY CLEAR
(\$label)	BCSL	\$address	Branch if CARRY SET
(\$label)	BVCL	\$address	Branch if OVERFLOW CLEAR
(\$label)	BVSL	\$address	Branch if OVERFLOW SET
(\$label)	BPLL	\$address	Branch if POSITIVE
(\$label)	BMIL	\$address	Branch if NEGATIVE

The SEV macro satisfies the 65816 processor's lack of an instruction to set the status register V-flag:

(\$label) SEV

**Mode Directive Macros**

The 65816 processor and assembler can together operate in a large number of 16- and 8-bit mode combinations. To simplify the job of the programmer and thereby (hopefully) reduce errors, a set of macros are provided to support mode selection:

{\$label} BYTE	{\$type}	set 8-bit mode
{\$label} WORD	{\$type}	set 16-bit mode
{\$label} WRDROUT		Start scope of local labels in 16-bit mode
{\$label} BYTROUT		Start scope of local labels in 8-bit mode
{\$label} WXYBAMROUT		Start scope of local labels with index registers in 16-bit mode and accumulator and memory access in 8-bit mode.
{\$label} ENDROUT		End of scope of local labels

These macros set the correct mode for both assembler and processor. The \$type string will normally consist of either "(\$)M" specifying the accumulator or "(\$)X" specifying the index registers. If \$type is not present the default is "(\$)M+(\$)X".

The following two macros set assembly mode only:

{\$label} BYT	{\$type}	set 8-bit mode
{\$label} WRD	{\$type}	set 16-bit mode

### *Module Macros*

Two macros are provided for the creation of the standard Communicator Module Header at the start of a module.

The START macro is placed immediately before the entry point to the module code:

**(\$label) START \$module, \$ver, \$flag, \$code, \$arg, \$sif, \$prog**

where:

<b>\$module</b>	is a string representing the module name. The string should be of less than 10 characters and can be followed by an optional list of slash ("/") separated star ("*") commands supported by the module.
<b>\$ver</b>	is a 4 nibble BCD version number.
<b>\$flag</b>	is a flag byte - no longer used.
<b>\$code</b>	is the code field. Contains flags indicating module type and attributes. For example for a device driver written in BASIC that can appear on the main menu \$code might be "&MHCDEV+&MHCBAS+&MHCMEN".
<b>\$arg</b>	is the argument string. For each command in the \$module string, this contains a string to be printed out by *HELP, detailing the arguments to that command. The commands' argument strings are slash ("/") separated. The entire argument string must be terminated with a NULL (&00).
<b>\$sif</b>	is a pointer to a special information field. This allows unlimited expansion of the module header.
<b>\$prog</b>	is the label associated with the module's reason code handler. If \$code is not supplied the entry point is assumed to be directly after the module header.

The following example illustrates the use of the START macro:

**START WOMBAT,&0200,,&MHCBNK+&MHCPoS+&MHCDEV,,WOMSTART**

The name of the module is "WOMBAT", it is version "2.00", the module is bank and position independent, it is a device driver and its reason code handler resides at WOMSTART.

The other macro associated with module header support is FINISH:

**(\$label) FINISH**

FINISH should be placed at the end of the module source. It is used by the START macro to calculate the length of the module during the second assembly pass.

### *Operating System Macros*

These two macros are provided to simplify access to the Communicator MOS:

**(\$label) TESTESC**

this macro is designed for use by older software. It checks to see whether the ESCAPE key has been pressed. Returns with CARRY SET if the ESCAPE key has been pressed or RESET if not. DBXY are preserved, HA corrupted.

**(\$label) OPSYS \$arg**

this macro provides an easier interface to the COP opcode. \$arg is stored after the COP.

### **10.2.3 The SYSTEM Manifest Macros**

The SYSTEM manifest macros describe all of the important system data structures and constants. These are necessary for the OEM developer to be able to graft application onto the standard Communicator software.

The only macro that actually needs to be called from within an OEM's module definition is SYSTEM. SYSTEM calls all the relevant macros that comprise the manifest constants.

OEMs may define their own manifests with the DEF macro:

DEF \$value,\$label

The label defined by DEF will be give a pound ("£") prefix.

eg

DEF &00,A

would define £A as &00.

Definitions should never have comments on the same line.

The following naming conventions have been adopted:

Labels of 2 or 3 characters (excluding the £prefix) represent addresses;

Labels of 4 or 5 characters (excluding the £prefix) represent constants;

Labels of 1 character only (excluding the £prefix) are reserved and are used to define processor status register flags.

#### *The SYSTEM File Manifest Macros*

The SYSTEM file contains the three macros SYSTEM, SYS0 and SYS1.

#### **SYSTEM**

The SYSTEM macro is used to declare all of the other system manifests within an OEM module source. It is normally placed at the beginning of the source code.

#### **SYS0**

The SYS0 macro contains manifest definitions for various hardware locations:

£HWVER	this is the address of a location that defines whether the Communicator hardware is of the OLD or NEW type. If the location contains a null value the hardware is NEW, otherwise OLD.
£HWSTN	if the hardware is of type NEW then this location will contain the Econet station ID.
£HWVIA	this is the address of the 6522 VIA.

#### **SYS1**

The SYS1 macro determines whether a name table of system manifests will be built:

SYS1 \$nampar

The \$nampar parameter is the name of the table to be built.

**The SYS2 File Manifest Macros**

The SYS2 file contains manifest constants associated with central MOS functions. These include basic module and coroutine reason and status codes; preempt key values, COP call reason codes, system workspace definitions, task control block definitions, net filing system and Econet constants etc.

**Module Entry Reason Codes**

<b>£ENCOM</b>	-	Execute module command.
<b>£ENINIT</b>	-	Initialise module.
<b>£ENKILL</b>	-	Kill module and remove dependents.
<b>£ENHELP</b>	-	Help request.
<b>£ENRES1</b>	-	Reserved.

**Module Status Return Codes**

<b>ERCOK</b>	-	Return OK.
<b>ERCER</b>	-	Return Bad.
<b>ERCOKS</b>	-	Return OK request save screen.
<b>ERCERS</b>	-	Return Bad request save screen.
<b>ERCKEY</b>	-	Return due to exception key.
<b>ERCBAD</b>	-	Miscellaneous BAD return.
<b>ERCKILL</b>	-	Module not willing to continue.
<b>£ERFNP</b>	-	File not found.
<b>£EREEOF</b>	-	End of file.

**Special Characters Returned by RDCH and INKEY When C=1**

<b>£SCPRE</b>	-	Call preempted.
<b>£SCESC</b>	-	ESCAPE key pressed.
<b>£SCTIME</b>	-	Timeout.

**Preempt Key Values**

<b>£ECBASE</b>	-	Base preempt key value.
<b>£ECPHON</b>	-	PHONE key value.
<b>£ECCOMP</b>	-	COMP key value.
<b>£ECCALC</b>	-	CALC key value.
<b>£ECSTOP</b>	-	STOP key value.
<b>£ECHHELP</b>	-	HELP key value.

**Minimum Recommended Stack Size**

**£SMSTK**

**COP Signatures**

<b>£OPWRC</b>	-	Write character to the VDU drivers.
<b>£OPWRS</b>	-	Write string to the VDU drivers.
<b>£OPWRA</b>	-	Write string to given address.
<b>£OPNLI</b>	-	Send LF-CR to the VDU drivers.
<b>£OPRDC</b>	-	Read character.
<b>£OPCLI</b>	-	Command line interpret string.
<b>£OPOSB</b>	-	BBC OSBYTE call.
<b>£OPOSW</b>	-	BBC OSWORD call.
<b>£OPBGT</b>	-	BBC OSBGET call.
<b>£OPBPT</b>	-	BBC OSBPUT call.
<b>£OPCOM</b>	-	Command line interpret string.

<b>£OPERR</b>	-	System error.
<b>£OPADD</b>	-	Allocate Direct Page.
<b>£OPADF</b>	-	Allocate Fast Direct Page.
<b>£OPFD</b>	-	Free Direct Page. !!Use £OPFZB instead!!
<b>£OPAST</b>	-	Allocate stack.
<b>£OPFST</b>	-	Free stack.
<b>£OPASC</b>	-	BBC OSACI.
<b>£OPAEX</b>	-	Acknowledge event.
<b>£OPBYX</b>	-	Convert BBC BYX address.
<b>£OPRLH</b>	-	Read hex number from BHA line.
<b>£OPRLS</b>	-	Read string from BHA line.
<b>£OPFZB</b>	-	Free zero bank pool.
<b>£OPARM</b>	-	Call arithmetic module.
<b>£OPSCX</b>	-	Save context of VDU and MOS.
<b>£OPRCX</b>	-	Restore context of VDU and MOS.
<b>£OPSSC</b>	-	Save screen.
<b>£OPRSC</b>	-	Restore screen.
<b>£OPSEV</b>	-	Signal event(s).
<b>£OPPRE</b>	-	Allow preemption.
<b>£OPRLN</b>	-	Read line.
<b>£OPCVS</b>	-	Convert stack index.
<b>£OPCVD</b>	-	Convert Direct Page index.
<b>£OPCRC</b>	-	Calculate CRC.
<b>£OPBHA</b>	-	Make BHA point to in-line string.
<b>£OPREF</b>	-	Refer to module, (Reserved).
<b>£OPCMD</b>	-	Call module.
<b>£OPRFR</b>	-	Refer to module.
<b>£OPURF</b>	-	Unreference module.
<b>£OPDIS</b>	-	Write to status line.
<b>£OPFMA</b>	-	Find module.
<b>£OPWRM</b>	-	Write module name.
<b>£OPFPO</b>	-	Find pool owner.
<b>£OPIIQ</b>	-	Intercept interrupt.
<b>£OPRIQ</b>	-	Release interrupt intercept.
<b>£OPMIQ</b>	-	Modify interrupt vector.
<b>£OPSUM</b>	-	Compute end-around-carry sum.
<b>£OPSLM</b>	-	Scan list of modules.
<b>£OPRMI</b>	-	Read module info.
<b>£OPAM</b>	-	Add module.
<b>£OPNLU</b>	-	Name lookup.
<b>£OPEOF</b>	-	Test for End of File, (use £OPEND instead).
<b>£OPAH</b>	-	Allocate 16-bit handle.
<b>£OPFH</b>	-	Free handle.
<b>£OPCUH</b>	-	Call (device driver) using handle.
<b>£OPDTD</b>	-	Device driver command table decode.
<b>£OPGSR</b>	-	General system read routine.
<b>£OPADY</b>	-	Add Y to BHA.
<b>£OPATR</b>	-	Allocate transient.
<b>£OPFTR</b>	-	Free transient.
<b>£OPEC0</b>	-	Call low-level Econet routines.
<b>£OPRHL</b>	-	Read hardware links.
<b>£OPXKC</b>	-	Examine keyboard character.
<b>£OPOPN</b>	-	Open file.
<b>£OPCLS</b>	-	Close file.

<b>\$OPEND</b>	-	Check for EOF.
<b>\$OPMM</b>	-	Call memory manager.
<b>\$OPVER</b>	-	Return OS version number.
<b>\$OPFSC</b>	-	Free screen.
<b>\$OPNET</b>	-	Call high-level NET routines.
<b>\$OPVH</b>	-	Validate (16-bit) handle.
<b>\$OPBGB</b>	-	Read bytes from device/file.
<b>\$OPPB</b>	-	Write bytes to device/file.
<b>\$OPLOD</b>	-	Load file into memory.
<b>\$OPSAV</b>	-	Save memory into file.
<b>\$OPRLE</b>	-	Read file's LOAD and EXECUTE addresses plus length.
<b>\$OPWLE</b>	-	Update file's LOAD and EXECUTE addresses plus length.
<b>\$OPRAT</b>	-	Read file attributes.
<b>\$OPWAT</b>	-	Write file attributes.
<b>\$OPRSP</b>	-	Read file's sequential pointer.
<b>\$OPWSP</b>	-	Write file's sequential pointer.
<b>\$OPRPL</b>	-	Read file's physical length.
<b>\$OPRLL</b>	-	Read file's logical length.
<b>\$OPWLL</b>	-	Write file's logical length.
<b>\$OPRCH</b>	-	Read catalog header.
<b>\$OPRFN</b>	-	Read file/object names from directory.
<b>\$OPDEL</b>	-	Delete a named object.
<b>\$OPREN</b>	-	Rename an object.
<b>\$OPSTAR</b>	-	Command line interpret string at given address.
<b>\$OPFCX</b>	-	Free MOS/VDU context including fonts.
<b>\$OPOSX</b>	-	OSWORD-like call.
<b>\$OPDFR</b>	-	Defer call.
<b>\$OPERC</b>	-	Convert error return.

**System Workspace**

<b>\$OSM</b>	-	Start of system memory.
<b>\$CBA</b>	-	Start of control blocks.
<b>\$CBZ</b>	-	End of control blocks.
<b>\$CBF</b>	-	Pointer to first free control block.
<b>\$CBL</b>	-	Pointer to last free control block.
<b>\$EMMM</b>	-	Memory control block representing the machine.
<b>\$IQ</b>	-	Interrupt queue daisy chain.
<b>\$EMMC</b>	-	Memory control block representing CMOS RAM.
<b>\$EMMW</b>	-	Memory control block representing whole machine.
<b>\$EMMV</b>	-	Memory control block representing VDU RAM.
<b>\$MD</b>	-	Module list pointer.
<b>\$IQS</b>	-	IRQ stack.
<b>\$DFR</b>	-	Defer list.
<b>\$TV</b>	-	Vector of task pointers.
<b>\$TVSIZ</b>	-	Size of task vector.
<b>\$TB</b>	-	Copy of task control block.
<b>\$TBSIZ</b>	-	Size of task control block.

**Task Control Block Structure**

<b>\$TBLEN</b>	-	TCB length
<b>\$TBRES</b>	-	TCB resource vector.
<b>\$TBWRK</b>	-	Task work locations.
<b>\$TBSZ</b>	-	Number of work locations.

*Chapter 10*

<b>£TBBRK</b>	-	Break signature pointer.
<b>£TBCO</b>	-	Current coroutine pointer.
<b>£TBEV</b>	-	

*Reason Codes for £OPNET*

<b>£NETOP</b>	-	Do a fileserver operation.
<b>£NETXH</b>	-	Convert internal handle to external handle.
<b>£NETRF</b>	-	Read current fileserver station number.
<b>£NETWFW</b>	-	Write FileStore station number.
<b>£NETRR</b>	-	Read retry settings.
<b>£NETWR</b>	-	Write retry settings.
<b>£NETRC</b>	-	Read context handles.
<b>£NETWC</b>	-	Write context handles.

*Reason Codes for £OPECO*

<b>£ECOTX</b>	-	Transmit a packet.
<b>£ECORO</b>	-	Open control block for reception.
<b>£ECOPR</b>	-	Open control block to receive remote procedure call.
<b>£ECOOP</b>	-	Open control block to receive Operating System calls.
<b>£ECORP</b>	-	Poll for reception.
<b>£ECORR</b>	-	Read receive block.
<b>£ECORD</b>	-	Delete receive block.
<b>£ECOXB</b>	-	Allocate RAM for extra control block.
<b>£ECONB</b>	-	Return number of receive control blocks.

*Error Codes Returned by £OPECO*

<b>£ECOBD</b>	-	Bad control block.
<b>£ECOIH</b>	-	Invalid handle.
<b>£ECOIM</b>	-	Insufficient memory/no free handles.
<b>£ECONL</b>	-	Not listening.
<b>£ECONE</b>	-	Net error.
<b>£ECOCO</b>	-	Collision.
<b>£ECONC</b>	-	No clock.
<b>£ECOLJ</b>	-	Line jammed.

**The SYS2A File Manifest Macros****Device Driver Module Entry Codes**

<b>EDVRST</b>	-	Reset device.
<b>EDVOPN</b>	-	Open device.
<b>EDVCLS</b>	-	Close device.
<b>EDVBGT</b>	-	Read byte from device.
<b>EDVBPT</b>	-	Write byte to device.
<b>EDVCGT</b>	-	Read control byte.
<b>EDVCPT</b>	-	Write control byte.
<b>EDVEOF</b>	-	Test for End Of File.
<b>EDVBGB</b>	-	Get block of bytes.
<b>EDVBPB</b>	-	Write block of bytes.
<b>EDVLOD</b>	-	Load a file into memory.
<b>EDVSAV</b>	-	Save a block of memory to a file.
<b>EDVRLE</b>	-	Read LOAD and EXEC addresses for a file.
<b>EDVWLE</b>	-	Write LOAD and EXEC addresses for a file.
<b>EDVRAT</b>	-	Read file attributes.
<b>EDVWAT</b>	-	Write file attributes.
<b>EDVRSP</b>	-	Read sequential file pointer.
<b>EDVWSP</b>	-	Write sequential file pointer.
<b>EDVRPL</b>	-	Read a file's physical length.
<b>EDVRLL</b>	-	Read a file's logical length.
<b>EDVWLL</b>	-	Write a file's logical length.
<b>EDVRCH</b>	-	Read a catalog header.
<b>EDVRFN</b>	-	Read file/object names from a directory.
<b>EDVDEL</b>	-	Delete file/object.
<b>EDVREN</b>	-	Rename file/object.
<b>EDVUNK</b>	-	Execute/transient-load an unknown command.

**EVENT Flags**

<b>EEVESC</b>	-	ESCAPE key event.
<b>EEVPRE</b>	-	Preempt key event.
<b>EEVALL</b>	-	ESCAPE AND Preempt key events.

**Resource Control Block Definitions**

<b>ERSLEN</b>	-	Length of resource vector.
<b>ERSBNK</b>	-	Reserved bank(s).

**Standard Control Block Structure Definition**

<b>ENXT</b>	-	Next offset.
<b>ECBSIZE</b>	-	Size of control block.
<b>ECBTYPE</b>	-	Type of control block.
<b>ECBNXT</b>	-	Pointer to next control block.

**Control Block Types Stored in ECBTYPE**

<b>ECBFREE</b>	-	Free block.
<b>ECBMEM</b>	-	Memory descriptor.
<b>ECBSIND</b>	-	Static indirection.
<b>ECBMIND</b>	-	Module indirection.
<b>ECBCO</b>	-	Coroutine descriptor.
<b>ECBCO2</b>	-	Coroutine descriptor extension.

## *Chapter 10*

<b>£CBLCB</b>	-	LCB command.
<b>£CBMOD</b>	-	Module descriptor.
<b>£CBIQ</b>	-	Interrupt descriptor.
<b>£CBHN</b>	-	Handle descriptor.
<b>£CBDFR</b>	-	Defer block.

### *Interrupt Queue-1 Structure Definition*

<b>£IQNXT</b>	-	Pointer to next interrupt queue.
<b>£IQHW</b>	-	Hardware address.
<b>£IQAND</b>	-	AND mask.
<b>£IQEOR</b>	-	EOR mask.
<b>£IQCB2</b>	-	Second control block.
<b>£IQEN</b>	-	Enabled mask.

### *Interrupt Queue-2 Structure Definition*

<b>£IQPRI</b>	-	Interrupt priority
<b>£IQDIR</b>	-	Direct Page required.
<b>£IQPREG</b>	-	IRQ routine mode.
<b>£IQRTN</b>	-	IRQ routine entry address.
<b>£IQCNT</b>	-	Interrupt count.

### *Memory Pool Descriptor Structure Definition*

<b>£MMNXT</b>	-	Next descriptor in this pool.
<b>£MMSPL</b>	-	Memory sub-pool list.
<b>£MMBD</b>	-	Memory base (page).
<b>£MMEP</b>	-	Memory end (page).
<b>£MMMD</b>	-	Creator module.

### *Handle Descriptor Structure Definition*

<b>£HNNXT</b>	-	Next handle descriptor.
<b>£HNHD</b>	-	Associated 8-bit handle.
<b>£HNRTN</b>	-	Associated (indirection) routine to call.
<b>£HNORG</b>	-	User slots.

### *Module Descriptor Structure Definition*

<b>£MDNXT</b>	-	Next module descriptor.
<b>£MDDIR</b>	-	Module sub-list.
<b>£MDPTR</b>	-	Module pointer.
<b>£MDDHD</b>	-	Module handle.

### *Module Header Structure Definition*

<b>£MHBRL</b>	-	Branch to start of module code.
<b>£MHLEN</b>	-	Length of module.
<b>£MHVER</b>	-	Module version number.
<b>£MHCODE</b>	-	Module type etc.
<b>£MHSIF</b>	-	Pointer to special information field.
<b>£MHNAME</b>	-	Module name.

***Module Flags Definitions Located at £MHCODE***

<b>£MHCPoS</b>	-	Module is position indepent.
<b>£MHCBNK</b>	-	Module is bank indepent.
<b>£MHCFS</b>	-	Module supports the file system interface.
<b>£MHCDEV</b>	-	Module supports the device interface.
<b>£MHCBAS</b>	-	Module is a BASIC program supporting the BASREASON call.
<b>£MHCMEN</b>	-	Module may appear on the main menu.
<b>£MHCPD</b>	-	Module is a VIEW printer driver.
<b>£MHCONe</b>	-	Module may appear only once on the main menu.
<b>£MHCMKL</b>	-	Module may not be killed once started from the main menu.

***Croutine Descriptor Structure Definition***

<b>£COSTKL</b>	-	Coroutine stack pointer.
<b>£COPAR</b>	-	Pointer to parent coroutine.
<b>£COFLG</b>	-	Coroutine option flags.
<b>£COBNK</b>	-	Coroutine bank register.
<b>£CODIR</b>	-	Coroutine Direct Page register.

***Standard Zero-Page Descriptor Structure Definition***

<b>£ZPCO</b>	-	Current coroutine.
<b>£ZPARG</b>	-	Coroutine argument.
<b>£ZPADR</b>	-	Coroutine address.
<b>£ZPUSER</b>	-	First free user location.

***Standard Handles***

<b>£HDMMMM</b>	-	Whole of memory.
<b>£HDMMMT</b>	-	Task memory.
<b>£HDMMC</b>	-	CMOS RAM.
<b>£HDMMMW</b>	-	Whole machine.
<b>£HDMMMV</b>	-	VDU RAM.

***Indirections***

<b>£IVIB, £INIB</b>	-	Hardware JMPL IRQ/BRK routine.
<b>£IVCOP, £INCOP</b>	-	JMPL COP routine.
<b>£IVBRK, £INBRK</b>	-	JMPL BRK routine.
<b>£IVABT, £INABT</b>	-	JMPL ABT routine.
<b>£IVNMI, £INNMI</b>	-	JMPL NMI routine.
<b>£IVRST, £INRST</b>	-	JMPL RST routine.
<b>£IVIRQ, £INIRQ</b>	-	JMPL IRQ routine.

## *Chapter 10*

### *The SYS3 File Manifest Macros*

#### *Memory Management Reason Codes*

<b>EMMFP</b>	-	Free pool.
<b>EMMAX</b>	-	Allocate explicit region within pool.
<b>EMMASD</b>	-	Allocate small area descending.
<b>EMMALD</b>	-	Allocate large area descending.
<b>EMMASA</b>	-	Allocate small area ascending.
<b>EMMALA</b>	-	Allocate large area ascending.
<b>EMMBAS</b>	-	Return pool base.
<b>EMMLEN</b>	-	Return pool length.
<b>EMMTOP</b>	-	Return pool end address.
<b>EMMFND</b>	-	Find pool.
<b>EMMCCHK</b>	-	Check consistency of memory management structures.
<b>EMMAZB</b>	-	Allocate Zero-Page Descending.
<b>EMMMRG</b>	-	Merge two pools.

#### *fCO Reason Codes*

<b>ECOCRE</b>	-	Create coroutine.
<b>ECOENV</b>	-	Set coroutine environment.
<b>ECODEL</b>	-	Delete coroutine.
<b>ECODES</b>	-	Destroy coroutine.
<b>ECOREN</b>	-	Restore coroutine environment.
<b>ECOBRK</b>	-	Set coroutine break handler.
<b>ECOCBH</b>	-	Cancel coroutine break handler.
<b>ECORTB</b>	-	Return from coroutine break handler.
<b>ECOIAM</b>	-	Return handle of current coroutine.
<b>ECONAME</b>	-	Return name of current coroutine.
<b>ECOINIT</b>	-	Initialise coroutine.
<b>ECOKILL</b>	-	Kill coroutine.
<b>ECOHELP</b>	-	Invoke coroutine help.
<b>ECOMOD</b>	-	Create coroutine from module.

#### *Coroutine COP Calls*

<b>ECCO</b>	-	Call coroutine.
<b>ECWT</b>	-	Suspend a coroutine.
<b>ECRS</b>	-	Resume a coroutine.

#### *Clock Driver Entries*

<b>ECKCMD</b>	-	Clock driver command handler.
<b>ECKINIT</b>	-	Claim memory and initialise clock driver.
<b>ECKDAY</b>	-	Find day of week.
<b>ECKREAD</b>	-	Read the clock.
<b>ECKSET</b>	-	Set the clock.
<b>ECKPAGE</b>	-	Find Direct-Page address.
<b>ECKMNTH</b>	-	Find the number of days in a given month.

***The SYS4 File Manifest Macros***

***Processor Status Register Flag Definitions***

<b><i>EN</i></b>	-	N flag.
<b><i>EV</i></b>	-	V flag.
<b><i>EM</i></b>	-	M flag.
<b><i>EX</i></b>	-	X flag.
<b><i>ED</i></b>	-	D flag.
<b><i>EI</i></b>	-	I flag.
<b><i>EZ</i></b>	-	Z flag.
<b><i>EC</i></b>	-	C flag.

## 10.3 The DRIVER macros

### 10.3.1 Introduction

The DRIVER macros are provided to simplify the task of creating device drivers. The DRIVER macros provide support for device driver command stream decoding and tasks such as driver handle to Direct Page conversion.

The DRIVER macros are comprised of a set of driver associated functions and a set of driver associated manifest constants.

The operational macros reside in the file `$S.getDecode01`. The manifest constants are to be found in `$S.getDid01`.

OEM developers wishing to make use of the DRIVER macros should GET both files at the beginning of their driver module source code.

For further detail on the implementation of device drivers and the use of DRIVER macros see the Device Driver section.

### 10.3.2 The DRIVER Operational Macros

The DRIVER operational macros reside in `$S.getDecode01`.

Decode01 contains the following macros:

<b>OPDTD</b>	This macro implements the control stream command decoding mechanism.
<b>DTDCHD</b>	This macro converts supplied driver handle to a pointer to that driver's Direct Page.
<b>DTDTAB</b>	This macro implements a set of general functions to decode commands; allocate and release driver resources and environment; and maintain and return status information. Signifies the start of the DRIVER command table.
<b>DTDFUJ</b>	This macro is used to create and assign values to global strings.
<b>DTDENT</b>	This macro is used to add a command to the DRIVER command table.
<b>DTDFJ2</b>	This macro creates an in-line string.
<b>DTDEND</b>	This macro is used to terminate the DRIVER command table.

***The OPDTD Macro***

The OPDTD macro invokes the MOS COP call £OPDTD which decodes bytes received on a driver's command stream:

(\$label)	OPDTD
Called with:	The processor in WORD mode. D → Local Direct Page. A = control byte.

***The DTDCHD Macro***

The DTDCHD macro is called whenever a driver is invoked with its handle. DTDCHD converts the supplied handle to its associated Direct Page address.

(\$label)	DTDCHD
Called with:	The processor in WORD mode. Y = handle.
Returns:	D → lcoal Direct Page. A,X,Y preserved.

***The DTDTAB Macro***

The DTDTAB macro is the first macro called by a device driver after the standard module header macro START. DTDTAB implements a number of driver related functions which are accessed from within the DRIVER code via global pointers:

ENTRY	is the actual module entry point.
DTDALD	Allocates a handle and associated local Direct Page.
DTDCLS	Releases Handle and associated local Direct Page.
DTDCHD	Converts supplied handle to local Direct Page pointer. (Uses the macro DTDCHD).
OPEN	Resets the buffer pointers for a channel.
WRCS	Decodes bytes received on a channel's control stream.
RDCS	Reads a byte from the current status string.
SPT	Status BPUT.
CON	Checks to see if the driver has been opened in CONfigure mode.

***Invoking DTDTAB Macro***

The DTDTAB macro is invoked thus:

```
$label DTDTAB $rout,$sentry,$gdpage,$ldpage,$gdinit,$modname
```

Where:

<b>\$rout</b>	An identifier (<=4 chars) to uniquely identify caller.
<b>\$sentry</b>	Address to go to when module is called (RTS to come back).
<b>\$gdpage</b>	Amount of global direct page needed by DRIVER.
<b>\$ldpage</b>	Optional amount of handle-local direct page needed by DRIVER.
<b>\$gdinit</b>	Address of routine to initialise global direct page.
<b>\$modname</b>	Name of module (not in quotes).

***The ENTRY Function***

ENTRY is the actual DRIVER module start routine. When the DRIVER is invoked, ENTRY checks to see if a global DRIVER Direct Page has been allocated. If not one is obtained. Assuming everything is OK, ENTRY then invokes the code specified by the Sentry parameter in DTDTAB as a subroutine.

***The DTDALD Function***

DTDALD is used to allocate a handle and associated local Direct Page for a driver channel. The address of the Direct Page is stored in the handle's workspace.

Called with:      The processor in WORD mode.  
                  D -> global direct page.

On exit:      C = 0 succeeded.  
                  Y = Handle.  
                  A = direct page allocated. D is preserved.  
  
                  C => 1 failed.  
                  A,Y undefined.  
                  D preserved.

***The DTDCLS Function***

The DTDCLS function releases a handle and its associated local Direct Page.

Called with:      The processor in WORD mode.  
                  Y = handle.  
                  D -> Direct Page to be freed.

On exit:      C = 0 succeeded.  
                  C => 1 failed.

***The DTDCHD Function***

The DTDCHD function converts a supplied driver channel handle into a pointer to its associated local Direct Page.

Called with:      The processor in WORD mode.  
                  Y = handle.

On exit:          D -> local direct page.  
                  A,X,Y preserved.

***The OPEN Function***

The OPEN function resets the buffer pointers for the selected channel.

Called with:      Processor in WORD mode.  
                  D -> Local Direct Page.  
                  A = the flag returned in X by CON.

***The WRCS Function***

This function is used to decode commands received on a driver's control stream. Commands are received a byte at a time and sequentially analysed until either a complete command has been decoded or an error has been deemed to have occurred. Once a complete command has been successfully received, the appropriate command function is invoked.

Called with:      Processor in WORD mode.  
                  D -> Local Direct Page.  
                  A = character to be put to the control stream.

***The RDCS Function***

This routine reads a byte from the current status string.

Called with:      Processor in WORD mode.  
                  D -> Local Direct Page.

On exit:          C = 0 successfull  
                  A = character read.  
  
                  C = 1 End of status string.  
                  A undefined.

***The SPT Function***

This routine is used to build up the status string returned in response to a query "?" command on a driver's control channel. The character supplied in register A is added to the current status string. The first character of the string is created automatically and corresponds to the character following the query command.

Called with:      Processor in WORD mode.  
                  D -> local Direct Page.  
                  A = character to be added to the string.

## *Chapter 10*

### ***The CON Function***

The CON function checks to see if the driver has been opened in CONfigure mode. That is with "-" as the first character of the filename.

Called with:      The processor in WORD mode.  
BHA -> rest of name.

On exit:            C=1, X=2 configure open  
C=0, X=0 normal open.

### ***The DTDFUJ Macro***

The DTDFUJ macro creates a global string variable and assigns a value to it:

{\$label}            DTDFUJ \$stname,\$\$value

It is used by the DTDENT macro.

**The DTDENT Macro**

The DTDENT macro is used to create individual device driver command table entries. DTDENT is invoked thus:

**\$label DTDENT \$command,\$ptype,\$address,\$string, \$infstr**

Where:

**\$command** is the single character string (in quotes) which the user sends down the control stream to initiate the command.

**\$ptype** is an identifier which specifies the type of parameter the command takes, as follows :-

<b>NOP</b>	- the command takes no parameters.
<b>OPT</b>	- the parameter must be one of the options presented in the option string (see below).
<b>DEC</b>	- the parameter is a 2-byte decimal number
<b>HEX</b>	- the parameter is a 3-byte hex number (no &).
<b>STR</b>	- the parameter is an arbitrary string.
<b>OPC</b>	- only used for the "?" command - see below

Parameter types HEX and STR are only terminated by a control character, other types may also be terminated by the start character of the next command (ie !, @-Z).

**\$address** is the address which will be called when a complete and valid command has been issued. The routine will be entered in WORD mode with the following entry conditions :-

<b>NOP</b>	- no entry conditions
<b>OPT</b>	- Y = option used (0,2,4 ...)
<b>DEC</b>	- HA = number
<b>HEX</b>	- BHA = number
<b>STR</b>	- BHA -> string

The routine should exit with an RTS.

\$string	This string is only used when the option string parameter types (OPT or OPC) are being used. This type is used when a valid parameter to the command consists of one of a number of fixed strings (eg "on" or "off"). The option string consists of all these strings, separated by a delimiting character (normally "/"), and preceded by the character used for the delimiter eg "/on/off". When such a command has been received, the address will be called with Y in this case being 0 for on, 2 for off. For the OPC parameter type (used by "?"), the option string will be used if the parameter string is not one of the automatic status commands. If the parameter type is not OPT or OPC, this string should be "". The option string for a given command can be read by sending the "!" command, followed by the command letter. This is used by Configure.
\$infstr	This string gives a description of what the command sets. The string is read by sending the "%" command followed by the command character. The Configure program reads this string and prints it out on the left hand side of each entry. If it is not sensible to have a configured setting for this command, then \$infstr should be set to "".
\$label	This optional label, if present, indicates that an automatic status entry should be maintained for this command. Depending on \$type, an amount of local direct page will be automatically allocated for it (1 byte for OPT, 2 for DEC, 3 for HEX, 32 for STR) and the label \$label will be assigned to the start of that workspace, for use by the driver if necessary. Whenever a command is issued which has an automatic status entry, the parameter to the command is stored in the direct page allocated for it, before calling \$address.

#### *The DTDFJ2 Macro*

The macro DTDFJ2 creates an in-line string within the code. DTDFJ2 is invoked thus:

DTDFJ2 \$string

DTDFJ2 is used by the DTDEND macro.

#### *The DTDEND Macro*

The DTDEND macro is used to terminate the device driver command table. DTDEND is invoked thus:

DTDEND \$description\_string

\$description\_string is a quoted string of not more than forty characters which describes the driver. Null parameters may be supplied either in quotes, or (for compatibility with existing drivers) without quotes. This string is recovered by CONFIGURE using the "%%" command, and will appear following the device name at the top of the relevant configure page.

### 10.3.3 The Driver Manifest Macros

The driver manifest macros reside in the file \$S.get.dd01.

The manifest macros define a set of constants used by the driver operational macros. They can be divided into those that define offsets into a local handle's zero page corresponding to a standard set of driver variables and those that define a set of driver macro parameter types.

#### *Direct Page Workspace Definitions*

In fact most of these variables are not accessed directly by the driver macros. A lot of the driver macros functionality was moved to the operating system. The variables are used by the OS function invoked via the FOPDTD COP call.

DTTBP	-	BPUT buffer write-pointer.
DTRBP	-	BGET buffer write-pointer.
DTRBP2	-	BGET buffer read-pointer.
DTTABLE	-	Long pointer to device driver command table.
DTOFST	-	Offset to driver description string.
DTPARMS	-	Contains up to six bytes of command parameters.
DTFLAG	-	Flag byte for decoding.
DTADDR	-	Two-byte offset from command table to start of code.
DTOPAD	-	Two-byte offset from command table to options string.
DTINAD	-	Two-byte offset from command table to command description string.
DTTDP	-	Offset into Direct Page of status information.
DTTB	-	Buffer for command stream BPUT.
DTRB	-	Buffer for command stream BGET.
DTNLU	-	Control block for OPNLU.
DTCNFL	-	Configure-Open flag. TRUE when DTCNFL $\neq$ 0.
EDTDDP	-	Number of bytes of direct page needed.

#### *Parameter Type Definitions*

ENOP	-	No parameters.
EOPT	-	Option string parameter.
EDEC	-	Three-byte decimal number.
EHEX	-	Three-byte hexadecimal number.
ESTR	-	Arbitrary string.
EOPC	-	Option string.
ECN1	-	Read command description.
ECN2	-	Read option set.
ECN1CH	-	Character of command to read command description, ("%").
ECN2CH	-	Character of command to read option set, ("!").

## **10.4 Miscellaneous GET Files**

### **10.4.1 Introduction**

This section covers the remainder of the GET files supplied with the OEM Developer's Package. It is unlikely that a developer will need to use them but they are mentioned here for the sake of completeness.

The file **\$S.get.FILESYSTEM** contains manifest constants associated with the CMOS RAM filing system.

The file **\$S.hardware** contains manifest constants and macros associated with some of the Communicator hardware.

The file **\$S.get.OS** contains a few reason codes for invoking the MOS.

The file **\$S.get.fx** contains some MOS FX codes.

## 10.5 An Example Module

The following example module makes use of some of the SYSTEM macros along with some of its own.

The source is distributed amongst three files:

- DummyMacro contains local macro definitions.
- Dummy00 contains module manifest constant definitions and assembler directives.
- Dummy01 contains a few more manifest declarations along with the module header definition and the module code proper.

In addition there is an exec file Make which invokes the assembly and creates the executable module.

### 10.5.1 DummyMacro

The DummyMacro file contains local module macro definitions.

```
TTL Some example MACROS... > DummyMacro
;
MACRO ; Make BHA point to the given address.
$label BHApoin $address
; The code assumes WORD mode
$label
PHK ; Store program BANK.
PER $address ; and calculate the relative address.
PLA
PLB ; make data BANK = program BANK.

MEND
;
MACRO
$label BRL4 $address
$label
BRL $address ; 3 byte relative branch
= &00 ; plus 1 makes it an even 4.

MEND
;
END ; end of the MACRO file.
```

The macro BHApoin makes the BHA registers point to the supplied address.

The macro BRL4 makes a three-byte relative-branch instruction four bytes in length by tacking an additional zero onto the end of it. This macro is used to build vector tables where each branch entry must map onto a four-byte entry.

### 10.5.2 Dummy00

Dummy00 is the first source file for the dummy module. It contains assembler directives, the system manifest constants along with some module-specific declarations but no actual 65816 source code.

```
TTL    An example module.... 'DUMMY' > Dummy00

;-----;
; Created 7th July 1987 James G Smith
;-----;

; Source file print options when assembling

noprint *      &02          ; printing OFF and NEWPAGE
doprint *      &05          ; printing ON and NEWPAGE

defopt *      doprint      ; Default print state.

Dumy00 *      defopt      ; Default print state for Dummy00 file.
Dumy05 *      defopt      ; Default print state for Dummy05 file.

;-----;

OPT    noprint      ; Don't print out all of the SYSTEM
; definitions.

SYSTEM           ; Invoke the SYSTEM macro.

GET    DummyMacro   ; Include our own macro definitions.

OPT    Dumy00      ; Back to default print state.

;-----;

; Global Manifests

byte   *      &01          ; byte occupies 1 memory location.
word   *      &02          ; word occupies 2 memory locations.
laddr  *      &03          ; long address is 3 memory locations.
dword  *      &04          ; double-word is 4 memory locations.

null   *      &00          ; NULL value.
lf     *      &0A          ; line-feed character.
ff     *      &0C          ; form-feed character.
cr     *      &0D          ; Carriage-return character.
space  *      " "          ; Fairly self-evident.
comma  *      ":"          ;
fstop  *      ";"          ;

;-----;
```

```
GBLS $vers  
$vers SETS "&0100" ; our version number.
```

```
GBLS $module  
$module SETS "Dummy" ; thats us!
```

```
;
```

; Our Direct Page Allocations

```
^ &00 ; Start of Direct Page.
```

```
COMIn # laddr ; Command line pointer.
```

```
bufptr # laddr ; Pointer to data buffer.
```

```
char # byte ; Last character read.
```

```
DPend * @ ; After our last Direct Page entry.
```

```
;
```

LNK Dummy05

The macro SYSTEM is invoked to drag in the system manifest constants. The actual system GET files are included at assembly time.

Following the SYSTEM macro, the local macro file is included via the "GET DummyMacro" directive.

The module-specific manifest constants are then defined. These include two strings \$vers and \$module which hold the modules version number and name respectively.

Finally the Direct Page allocations are defined. These consist of offsets to variable within the module's Direct Page.

### 10.5.3 Dummy05

Dummy05 is the second module source file. Dummy05 contains a few more module-specific constant declarations, the module header and the module's 65826 source code.

```
TTL      An example module... the 2nd file    > Dummy05
;
ORG     &FF0000      ; Where our code is assembled for.
;
;
; Position independent.
; Bank independent.

mystate *          (LMPHPOS + LMPHBNK)

START  ,$vers,,mystate,,entry
      = "$Module/Message/Text",null
      = "/",null
;
;
```

```

;      BHA -> command line
;      X  -> =&0000 then command
;      Y  -> command number * 2

; code doesn't use up any Direct Page.

WRD
entry          ; Module entered here
               ; X=0 command.
               ; Chosen command in Y.
CPXIM &00      ; Check range of X and Y.
BNE exclr      ; Do nothing.

PHA

TYA
RORA          ; Check if Y is even.

PLA
BCS exclr     ; Is Y even?

CPYIM (jmpend-jmpstab-1)
BCS exclr     ; and in range?

TYA
ASLA          ; Multiply by 2 to get into 4 byte steps
               ; A now holds index into table

PER alldone - 1 ; Where we want to come back to.

PER (jmpstab - 1) ; Jump into table.
CLC             ; and out of the other side
ADCS 1
STAS 1
RTS            ; via RTS.

alldone
LDAIM &7E      ; Clear 'possible' ESCAPE condition.
OPSYS &OPOSB

errout
LDXIM &00      ; Notify receiver we dont have an error.

exclr
CLC
RTL           ; Exit.

:-----;

```

*Chapter 10*

```
jmpab
    BRL4  dummy      ; Some information
    BRL4  message    ; Some more information
    BRL4  text       ; Even more information
jmpend
;-----;

WRD

dummy
    PHK
    PLB
    PER  text1

    BRA  textout

message
    PHK
    PLB
    PER  text2

    BRA  textout

text
    PHK
    PLB
    PER  text3

textout           ; Print text until "null" found.
; in: S+1 -> address of text.
;      B -> BAK of text.
; out: Stack restored.

LDYIM  &0000      ; our index.

cloop
    TESTSC          ; Has ESCAPE key been pressed?
    BCS   done       ; Yes.

    LDASIY 1        ; Get the character.
    ANDIM  &00FF      ; NULL?
    BEQ   done       ; Yes.
    CMPIM  cr        ; Carriage return?
    BNE   pit        ; No.
    CMPIM  cr        ; Yes.

    OPSYS           EOPNLI      ; Output a newline.
```

```
INY  
BRA    cloop

pit          ; Print character.  
OPSYS      £OPWRC

INY  
BRA    cloop

done  
OPSYS      £OPNLI  
PLA        ; Dump the address.

RTS

;-----;

text1  
=  "This command does absolutely nothing"  
=  null

text2  
=  "This is a slightly longer message",cr  
=  "Instead of just printing some text,"cr  
=  "this command could be doing something",cr  
=  "useful"  
=  null

text3  
=  "More nothing happens here"  
=  null

;-----;

FINISH

END          ; No more files.
```

## *Chapter 10*

The ORG directive at the beginning of the file defines where the code should be assembled for.

The START macro creates the module header. Examining START's parameters we can see that the module version number is 01.00 and that it is bank and position independent. The final parameter specifies that the access point to the module is at the point labelled "entry".

Following the START macro the lines

```
= "$Module/Message/Text",null  
= "/",null
```

specify that the module supports the two star ("\*") commands Message and Text; and that neither of them takes parameters.

The module code proper starts at "entry". It performs the usual checking on reason codes and if it is satisfied with the one supplied by the calling routine, vectors to the appropriate function via the vector-table jmptab.

jmptab uses the BRL4 macro, defined in DummyMacro, to implement a set of branch vectors.

The routine eloop makes use of a number of system macros and manifest constant definitions, amongst them:

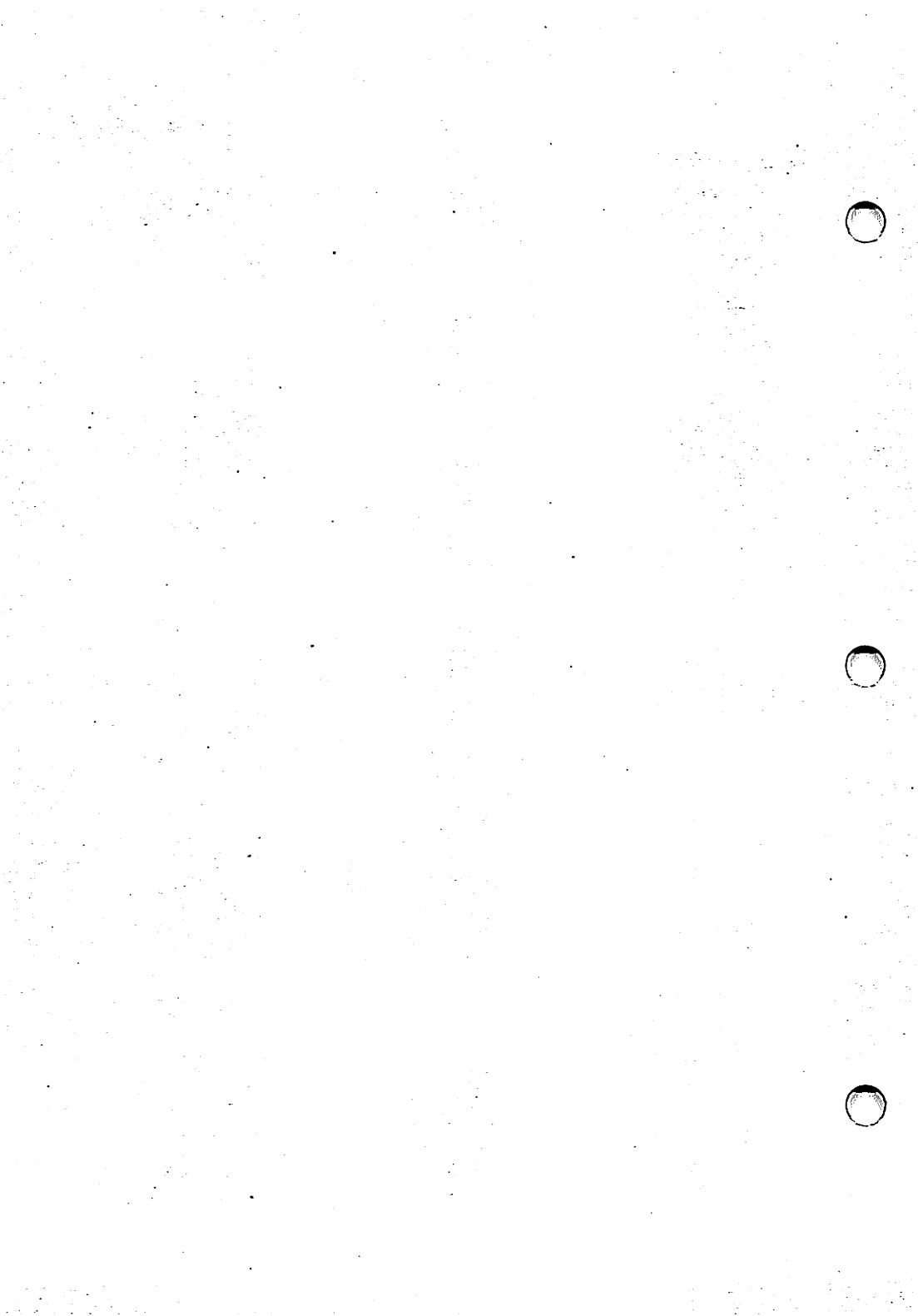
```
TESTESC  
OPSYS £OPNLI  
OPSYS £OPWRC
```

Note the FINISH macro prior to the END directive. FINISH is used by the START macro to calculate the size of the module.

#### 10.5.4 The MAKE File

The MAKE file, listed below, is an exec file that assembles the module sources and creates the executable object image. Note the inclusion of the SYSTEM GET files containing the SYSTEM macros and manifest constants from the assembler command line.

```
*!l Make
*opt 1,1
*NEWTUR
*****
STOP OFF
WIDTH 79
PRINT OFF
MLEVEL OFF
*****
ASM Dummy00
$.S.GET.SYSTEM
*****
*BASIC
CHAIN "loader"
*!b Done
```



# **11. Modem driver and BABT requirements**

## **11.1 Introduction**

This Chapter describes the modem driver resident in Communicator systems software.

The modem driver conforms to the standard driver interface and provides access to the facilities of Communicator's on-board modem.

The Communicator Modem Driver provides:

- Selectable baud rates
- Tone and Pulse dialling
- Selectable dial pausing
- Multiple V-series interfaces
- Auto-answer capability
- XON/XOFF flow-control
- Idle line watchdog timeout

Section 2 covers the modem driver commands, section 3 makes some suggestions on how to access the driver, section 4 describes the finite state model that the driver software implements and finally section 5 covers BABT requirements.

## 11.2 Modem Driver Commands

### 11.2.1 Introduction

The modem driver conforms to the usual driver standards.

Driver commands are sent to the driver a byte at a time on its control stream.

The modem driver supports the following commands:

A	- Auto-answer
B	- Send break
C	- Call a number
D	- Set data rate
E	- External line access
F	- Flush buffers
G	- Set watchdog timeout
H	- Set handshake protocol
I	- Set size of receive buffer
J	- Set carrier detect time
K	- Hook
L	- Loudspeaker
M	- Modem mode
N	- Set dialler type
O	- Set transmit buffer size
P	- Set parity
Q	- Set quit mode
S	- Set stop bits
T	- Send string of tone data
W	- Word length

In addition to the above a set of query commands is also supported. These usually correspond to one of the above commands.

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 bits setting
W	- Last word length setting
d	- Read incoming carrier
e	- Read error status
g	- Return status of watchdog
s	- Return modem state

### **11.2.2 Baud rate and modem mode**

The modem driver supports a variety of transmit/receive baud rate combinations and V-series interface modes.

Baud rates are selected using the "D" command which takes one of the following parameters:

1200/75	-	Receive = 1200, transmit = 75
75/1200	-	Receive = 75, transmit = 1200
300/300	-	Receive = 300, transmit = 300

The currently selected baud rates can be determined via the "?D" command.

The V-series mode of operation is selected with the "M" command which takes one of the following parameters:

v23c	-	V23 tones, full-duplex operation, DCE
v23ce	-	V23 tones, full-duplex equalised operation, DCE
v23t	-	V23 tones, full-duplex operation, DTE
v23te	-	V23 tones, full-duplex equalised operation, DTE
v21o	-	V21 tones, full-duplex, originate
v21a	-	V21 tones, full-duplex, answer

The "?M" command will return the last mode successfully set.

Only the following baud rate/mode combinations are legal:

75/1200	v23c
75/1200	v23ce
1200/75	v23t
1200/75	v23te
300/300	v21o
300/300	v21a

Attempts to set illegal combinations will be ignored.

### 11.2.3 Dialling

The modem driver provides for both tone and pulse dialling. In addition the application may insert different length pauses between digits dialled.

The modem driver is instructed to dial via the "C" command. The "C" command takes a dial string as a parameter.

The dial string is build up from the following characters:

0	-	Dial zero
1	-	Dial one
2	-	Dial two
3	-	Dial three
4	-	Dial four
5	-	Dial five
6	-	Dial six
7	-	Dial seven
8	-	Dial eight
9	-	Dial nine
A	-	Dial A (tone dialling only)
B	-	Dial B (tone dialling only)
C	-	Dial C (tone dialling only)
D	-	Dial D (tone dialling only)
*	-	Dial star (tone dialling only)
	-	Dial hash (tone dialling only)
E	-	Send the external line access string
T	-	Select tone dialling
P	-	Select pulse dialling
-	-	Pause for 850 mS
.	-	Pause for 2S
/	-	Pause for 4S

If none of "E", "T" or "P" is specified, the configured dialling method is used.

The last number dialled can be obtained via the "?C" command.

The external line access string is used to obtain an outside line when dialling on an internal extension. The external line access string is set via the "E" command. The string may contain any of the above characters except for "E". The default external access string is "9".

The following command dials the number "905338081" in tone mode with a two second gap between the "9" and the "0":

CTE.05338081

#### **11.2.4 Ring Detection and Auto-Answer Facilities**

The modem hardware is capable of detecting a "ringing" voltage on the phone-line. The modem driver can monitor this line for patterns of ringing voltage representing a genuine ringing signal and flag this to the user, (via an audible ringing signal), and/or to the application, (via the "?r" command). Thus applications may implement either manually answered or true auto-answered calls.

The following modem functions are used for:

The "A" command;

The "?r" command;

The "Koff" command to instruct the modem to answer a call once ringing has been detected;

Ring detection software consisting of a monitor routine called under a timed interrupt. This monitors the state of the RING bit in the modem hardware.

##### ***The "A" Command***

The "A" command is used to set the ring-detection monitor. The command may be issued at any time but only takes effect when the modem (re-)enters the s-idle state.

The "A" command takes the following parameters:

- |       |   |  |
|-------|---|--|
| on    | - | This enables the ring-detector which sets a flag whenever the ringing is detected and also generates an audible signal. The flag may be polled via "?r". |
| quiet | - | This turns off the audible signal.   |
| off   | - | This disables the ring-detector.   |

##### ***The "?r" Command***

The "?r" command returns the state of the ring detector flag. If the phone is ringing, "?r" returns rtrue, otherwise rfalse.

##### ***The "K" Command***

The "K" command is used to put the phone on or off the hook. The phone is automatically taken off the hook for a call command ("C"), and put on the hook when the channel is closed.

The "K" command takes the following parameters:

- |     |   |                              |
|-----|---|------------------------------|
| on  | - | Put the phone on the hook    |
| off | - | Take the phone off the hook. |

The "?K" command returns the current state of the hook.

### **11.2.5 Handshake Protocol**

The modem driver provides flow control capability via the standard XON/XOFF handshake protocol. When flow control is in operation, if the modem driver's receive buffer has fewer than thirty bytes free when a character is received, an XOFF character is transmitted. If subsequently a character is removed from the receive buffer, leaving more than sixty bytes free, an XON character is transmitted.

Flow control is selected via the "H" command which takes the parameters:

none	-	Disables flow control
xon/xoff	-	Enables flow control

The "?H" command returns the current handshake protocol being implemented.

### **11.2.6 The Idle Line Watchdog**

The watchdog allows timing out after a user-configurable period, on lines which are idle, or which are receiving only noise.

The timeout period is set by modem driver command "G" which takes one of the following parameters:

off	-	Turn the watchdog off
1min	-	Set idle line timeout period to one minute
5min	-	Set idle line timeout period to five minutes
15min	-	Set idle line timeout period to fifteen minutes
60min	-	Set idle line timeout period to sixty minutes

This command is made available to the user via Configure. Note that the 'off' setting is equivalent to an infinite timeout. The watchdog is active, with the user specified timeout whenever the modem is connected. In addition the watchdog is enforced, with a 50 second timeout regardless of the user's Configured setting, whenever the modem is in answer state. Thus, if a call is answered, and no carrier is detected for 50 seconds, the watchdog will drop the line.

If the watchdog times out, it returns the modem to idle state. This drops the line, re-initialises the modem for possible further calls, and sets various flags allowing the application to 'see' that the line has been dropped, and to discover why (loss of carrier or watchdog).

Once a call is established, the application should poll the modem state using the "?s" command, described below. If the state returns to idle before the call is terminated by the application, this shows that either the watchdog or the carrier monitor has 'fired', causing the line to be dropped. The application may determine which of these events has occurred using the "?d" and "?g" commands.

The watchdog runs entirely within timed interrupt routines, and therefore continues to function even if the user has pre-empted out of the phone task, or other application running the modem.

For auto-answer calls only, the watchdog is enforced with a timeout of 50 seconds when the call is answered; if in-coming 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.

***The "?s" Command***

The "?s" command returns the current state of the modem as defined in the finite state model described below. Possible responses are:

idle	-	Modem driver idle
sdial	-	Modem driver dialling
sstart	-	Starting a call
sconnect	-	Connected
sanswer	-	Answering an incoming call

Note that the 'dial' state currently exists only for the duration of the dial command, and is not normally visible to the application.

***The "?d" Command***

The "?d" command returns the state of the carrier flag. If the carrier is present it returns on otherwise off.

***The "?g" Command***

The "?g" command informs the application whether the watchdog has fired or not. If the watchdog has fired it returns gok, otherwise gfail.

### **11.2.7 Buffer Functions**

The modem driver provides several commands for buffer handling.

The size of the input buffer is set using the "I" command which take a decimal size argument. The function attempts to allocate a buffer of the requested size. If it fails, it attempts to allocate a buffer of the previous size. If this fails, it attempts to allocate a standard size buffer. If it cannot allocate a standard size buffer, a fatal system error is generated.

The size of the current input buffer is obtained via the "?I" command.

The size of the output buffer is set using the "O" command. This shuts down the current output buffer and attempts to allocate a new one in the same manner as the "I" command. If it fails to allocate a buffer, a fatal system error is generated.

The "O" command is should only be invoked when the modem is in the idle state.

The size of the output buffer is returned by the "?O" command.

Transmit and receive buffer sizes should be established before the "C" or "Koff" commands are issued, or after a "Kon".

If there are characters in the buffer when a set command is issued, the command is ignored.

To flush buffers use the "F" command. This takes one of the following as a parameter:

rx	-	Flushes the receive buffer
tx	-	Flushes the transmit buffer
both	-	Flushes both buffers

### **11.2.8 Carrier Detection**

The driver command "J" sets the minimum time that carrier must be constantly detected by the modem hardware before the software accepts it is present.

The "J" command takes a parameter which must be one of the following:

15ms	-	Minimum carrier duration of fifteen milliseconds
75ms	-	Minimum carrier duration of seventy five milliseconds
250ms	-	Minimum carrier duration of two hundred and fifty milliseconds
750ms	-	Minimum carrier duration of seven hundred and fifty milliseconds
2500ms	-	Minimum carrier duration of two thousand five hundred milliseconds
voice	-	no carrier detection

The voice setting must be used for all voice calls.

### **11.2.9 Data Format Commands**

The modem driver provides a set of functions to select the format of the transmitted and received data.

The "P" command is used to select the data's parity. "P" takes one of the following as a parameter:

odd	-	selects odd parity
even	-	selects even parity
none	-	selects no parity

The current parity setting is returned by the "?P" command.

The "S" command selects the number of stop bits per character and takes one of the following as a parameter:

1	-	selects 1 stop bit
2	-	selects 2 stop bits

The current stop bit setting is returned by the "?S" command.

The "W" command selects word length and takes one of the following as a parameter:

7	-	selects 7 data bits in a word
8	-	selects 8 data bits in a word

The current word length setting is returned by the "?W" command.

Only the following combinations of these commands are permitted:

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

### **11.2.10 Miscellaneous Commands**

#### ***The "B" command***

The "B" command forces the modem driver to purge its transmit buffer and send a break of a specified duration. The command takes a decimal value as a parameter specifying the duration of the break in units of one hundred milli-seconds.

#### ***The "L" Command***

The "L" command turns the feedback to the internal loadspeaker on or off. It takes one of the following as a parameter:

on	-	Loudspeaker ON
off	-	Loudspeaker OFF

The "?L" command returns the state of the loadspeaker.

#### ***The "N" Command***

The "N" command sets the dialler type and takes one of the following as a parameter:

pulse	-	selects pulse dialling
tone	-	selects tone dialling

The "?N" command returns the current type of dialling selected.

#### ***The "Q" Command***

This command controls what happens when BGET is called with an empty receive buffer or BPUT called with a full transmit buffer.

The "Q" command is called with one of the following as a parameter:

on	-	quit on
off	-	quit off

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.

The "?Q" command returns the current Quit setting.

***The "T" Command***

The "T" command sends a string of DTMF data down the line. Characters may be selected from amongst the following:

0	-	Send tone for zero
1	-	Send tone for one
2	-	Send tone for two
3	-	Send tone for three
4	-	Send tone for four
5	-	Send tone for five
6	-	Send tone for six
7	-	Send tone for seven
8	-	Send tone for eight
9	-	Send tone for nine
A	-	Send tone for A
B	-	Send tone for B
C	-	Send tone for C
D	-	Send tone for D
*	-	Send tone for star
#	-	Send tone for hash
E	-	Send the external line access string
.	-	Pause for 850 mS
:	-	Pause for 2S
/	-	Pause for 4S

***The "?e" Command***

The "?e" command returns a string indicating which errors have occurred since the last "?e" command. The returned string is made up from none or more of the following characters:

b	-	Receive buffer overrun
p	-	Parity error
r	-	Receiver overrun
f	-	Framing error

Errors p, r and f only refer to bytes which have actually been read by the user since the last request for error status.

Error b will be set after the last character before the overrun has been read.

## **11.3 Accessing the Driver, Typical Call Sequences**

### **11.3.1 Opening the Modem**

Before a call can be made, the modem must be opened, using OPENUP from BASIC or £OPOPN from assembler. The application should check that the OPEN has succeeded before proceeding; from BASIC, failed opens will return a zero handle.

The most likely causes of the modem failing to OPEN are:

1. The phone line is disconnected.
2. The modem is already OPEN.
3. Lack of memory in a busy machine.

A single OPENing of the modem may be used to make or answer one or more calls. If multiple calls are required, each call must be terminated, either explicitly by a Kon command, or automatically by the watchdog timing out or by the carrier monitor detecting lack of carrier.

### **11.3.2 Starting Data Calls**

Once a channel is open to the modem, calls may be started by issuing a "C" command with a suitable dial string. The "C" command will only be accepted if the modem is in idle state; a newly-opened modem will automatically be in idle state. If in doubt, "?s" may be used to check the state, or Kon may be issued to ensure the modem is returned to idle state.

The characteristics of the data call (data rate and format, buffer sizes, watchdog, carrier detection time, loudspeaker feedback etc) may either be set explicitly by the application, or left for the user to select via configure. Note that buffer sizes cannot be changed once a call has been initiated. For data calls, the carrier detection setting "voice" must not be used, otherwise carrier will never be found.

After dialling is complete, the application should poll the modem's state, using "?s", until connect state is reached. The modem will automatically assert carrier when it enters connect state. The application may abort the call using "Kon" before a connection is established.

### **11.3.3 Starting Voice Calls**

For voice calls, carrier detection time should be set to "voice"; this prevents the line being dropped prematurely due to spurious carrier detection.

Dialling is as for data calls. Once dialling is complete, the application should instruct the user to pick up the handset, and then press a key to transfer the call to the handset. On receipt of the key press, the application should issue a "Kon" to cause the modem to drop the line, to transfer the call.

### **11.3.4 Answering calls**

The modem may be used to answer incoming calls. The application should issue either Aon or Aquiet to initiate ring-detection, as described above. It should then poll the modem using "?r" until ringing is detected.

When ringing occurs, it may be answered either automatically (by issuing "Koff" immediately), or 'manually' (by prompting the user for permission to answer the call before issuing "Koff"). For auto-answer, BABT approval requires that the application guarantees to close down the line within 50 seconds if meaningful data is lost, or is never established.

"Koff" always initiates the V25 answer sequence.

Once a data call has been answered, the application should poll the modem for connect state, as for voice calls.

### 11.3.5 Transmitting and Receiving data

Once 'connect' state has been reached, for either an originated or an answered data call, data transmission and receipt may commence.

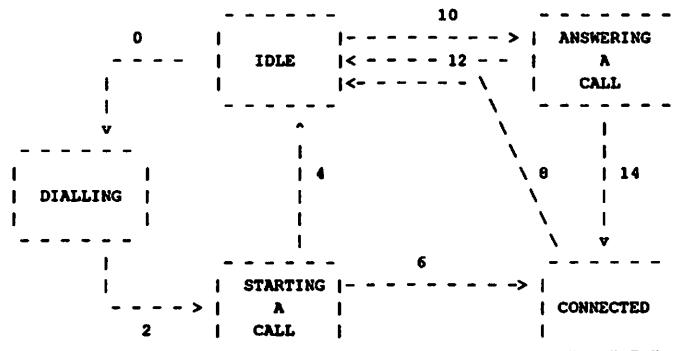
The application may send bytes for transmission using BPUT, and receive bytes using BGET. Note that use of BPUT before 'connect' state is reached will cause bytes to be buffered until they can be transmitted.

It is usual for terminal-like applications to set the modem to "Qon" mode; this causes BGET to return immediately if the receive buffer is empty and BPUT to return immediately if the transmit buffer is full. The application should check the error flag (@ERC%) for these conditions.

The application should poll the modem state occasionally using "?s"; if the state returns to idle before the application explicitly terminates the call, this must be because either the watchdog has 'fired', or because carrier has been lost. The application may discover which of these conditions has occurred by using "?g" and "?d".

## 11.4 Finite State model of modem behaviour

This section describes a finite-state model for the behaviour of the modem driver.



### 11.4.1 Transition 0

Occurs on receipt of a "C" (Connect) command from the application. During transition the line is grabbed. This transition is not allowed if the line is in use or disconnected.

#### **11.4.2 Transition 2**

Occurs on completion of the "C" command, after the specified string has been dialled. (NB The dial string may be null, if required to connect to dedicated phone lines).

Note that the dialling state only exists for the duration of the "C" command. It is therefore invisible to the user/application, but is seen by the monitor and hardware interrupts, and is necessary to maintain correct function of these during dialling.

#### **11.4.3 Transition 4**

Occurs on receipt of a "Kon" command from the application, if the application decides to abort the call, or wishes to route the call through to the handset. The Tx buffer should be flushed on this transition.

#### **11.4.4 Transition 6**

Occurs when the carrier detection monitor is satisfied that an incoming carrier is present.

#### **11.4.5 Transition 8**

Occurs when the user issues a "Kon" command, or when the watchdog or carrier detect monitor fires. The tx buffer should be flushed on this transition.

#### **11.4.6 Transition 10**

Occurs when the application instructs the modem to answer the call, using the "Koff" command. The line is grabbed, and the carrier monitor and watchdog activated.

This transition is not allowed if the line is in use or disconnected.

#### **11.4.7 Transition 12**

This transition guards against cases where the caller has hung up before we answer, or where the caller fails to assert carrier within a reasonable time. It also guards against faulty applications which might instruct the modem to answer before a ringing signal has been detected.

This transition occurs if the watchdog or noise detector should fire before carrier is detected. Their use is enforced with a timeout of 50 seconds during the waiting for carrier state; this is different from the corresponding state for originate calls.

#### **11.4.8 Transition 14**

Occurs when the carrier detector finds genuine carrier. Watch-dog and noise-detect are reset to their user-selected states.

## **11.5 Guidelines to BABT Requirements**

### **11.5.1 introduction**

Applications programmers developing software for use on Communicator must ensure that their programs do not cause Communicator to contravene BABT requirements for modem operation. Your attention is drawn to the Safety and Approvals section 1.6 at the front of this manual. The following guidelines are provided to assist in the design process.

### **11.5.2 Dialling**

It is recommended that for pulse dialling, inter-digit pauses be of no more than 850ms.

The delay between issuing the "Koff" command and transmitting dial data should be no greater than 5 seconds.

An application should not attempt to dial another number within 3 seconds of a call being cleared.

If the first call to a particular number is unsuccessful, then up to three further automatic call attempts to the number be made within a minimum of one minute between the end of one attempt and the start of the next. If all call attempts to the same number are unsuccessful, further automatic attempts should be prevented.

British Telecom recommends that intelligent devices should record or store data on failed call attempts in order to assist in investigating the cause of the failure.

### **11.5.3 Connection**

An application should clear down a call if the state sconnect is not achieved within 40 seconds of initiating a call.

### **11.5.4 Answering**

Where an application implements auto-answering, it should establish the state sconnect in not less than 300ms and not more than 3 seconds of receiving incoming call indication.

### **11.5.5 Disconnection**

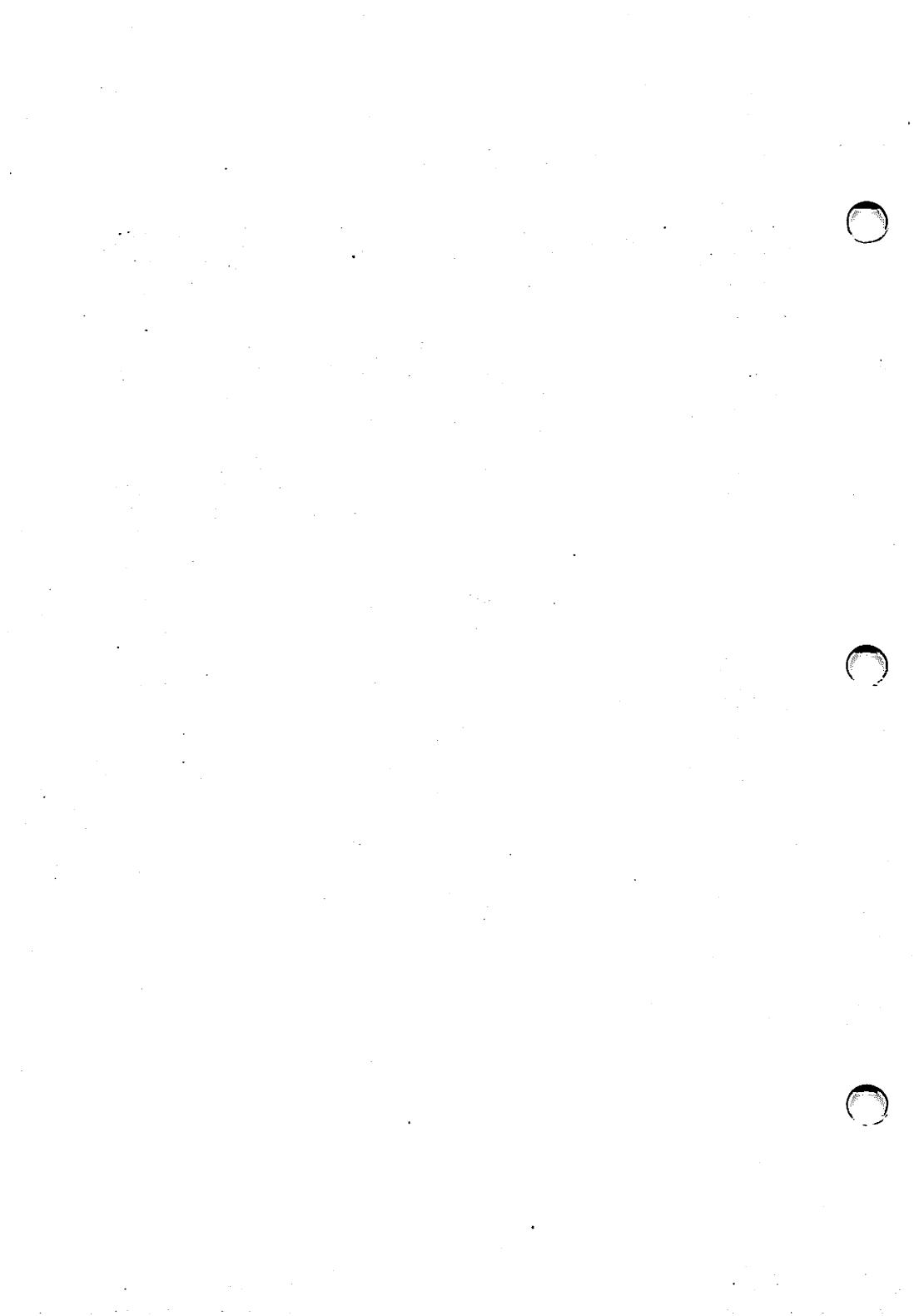
When the driver is in the state sconnect, the application should issue a "Kon" command to clear down the call under the following circumstances:

Completion of sending data in the absence of any further data to be recovered.

Completion of reception of data in the absence of any data to be transmitted.

Detection of carrier from the other party.

If data is not received from the other party within 15 seconds of having last transmitted data.



## 12. MASM

This Chapter describes the MASM macro assembler for the 65SC816. The assembler executes on an ARM second processor under 65TURBO, and provides a powerful tool for developing software in 65SC816 assembly language.

A knowledge of the architecture and instruction set of the 65SC816 is assumed. An introduction to the chip is given in the Introduction to Communicator Programming. A summary of the instruction mnemonics and addressing mode prefixes is given later in this section. MASM has a powerful set of operators and directives which are described in sections 2 to 6. Section 7 describes how to use the assembler to create object code files.

One of MASM's features is the ability to produce cross-reference information. This is used by the program XREF to provide information about the usage of labels in an assembly. XREF is described in section 8.

Section 9 describes the known bugs and restrictions associated with MASM.

MASM source files may be prepared on any editor which uses the carriage return character to terminate lines and does not insert other control characters into the text. TWIN is an example of a suitable editor. Either disc or network filing systems may be used to store source and object files.

## 12.1 General source file format

A MASM source file consists of lines of text. Each line may contain an optional label, an optional instruction (real or directive) with its operand, and an optional comment.

Labels must start in the first column of the line. They may be of any length, but only the first six characters are significant. Throughout MASM, upper and lower case are treated the same (except within quoted strings). Labels may contain letters, digits, \$ and \_ . They may not start with a digit. It is possible to have local labels which are two-digit numbers. These are described in the section on the ROUT directive. Example labels are:

```
LOOP  
01  
LANGENTRY
```

The instruction is separated from the label field by at least one space. Instructions are mnemonics followed by addressing mode letters, or directives. Examples are:

```
LOOP      LDAIM ">"  
          INX  
          MACRO  
          {
```

The second example has a label, and the third and fourth examples are directives. A full list of instruction and addressing mode formats for 65SC816 instructions is given in the Introduction to Communicator Programming.

Comments start with a semicolon ; and occupy the rest of the source line. A comment is allowed to start in the label field if it is the only thing on the line. Examples are:

```
; Just a comment on this line  
RSTART ; Go here to restart the language  
JSR GETXY ; Get the entry address in XY
```

Note that the label and instruction fields are 'expanded out' in assembly listings to make them align.

A source line may also contain an operating system \* command if the \* appears as the first character on the line. An example is:

\*FX5,4

The source file ends either with the directive END, which should be followed by a new line character, or a LNK directive to chain another source file.

## 12.2 The 65SC816 mnemonics

This section lists the complete set of mnemonics recognised by MASM. Where there are two alternatives, the preferred one is described and the alternative is cross-referenced to it. For example, BCS is the preferred mnemonic, BGE being the alternative.

All mnemonics are three letters long except for JMPL (jump long).

AND - logical AND accumulator with memory

ASL - Arithmetic shift left

BCC - Branch if C clear

BCS - Branch if C set

BEQ - Branch if Z set

BGE - See BCS

BIT - Bit test

BLT - See BCC

BMI - Branch if N set

BNE - Branch if Z clear

BPL - Branch if M clear

BRA - Branch always

BRK - Force a break

BRL - Branch always long

BVC - Branch if V clear

BVS - Branch if V set

CLC - Clear the C flag

CLD - Clear the D flag

CLI - Clear the I flag mask

CLR - See STZ

CLV - Clear the V flag

CMP - Compare accumulator with memory

COP - Coprocessor instruction

CPA - See CMP

CPX - Compare X with memory

CPY - Compare Y with memory

DEA - See DEC

DEC - Decrement accumulator or memory

DEX - Decrement X

DEY - Decrement Y

EOR - Exclusive-OR memory and accumulator

INA - See INC

INC - Increment accumulator or memory

INX - Increment X

INY - Increment Y

JMPL - Jump long to new location

JMP - Jump to new location

JSL - Jump to subroutine long

JSR - Jump to subroutine

LDA - Load the accumulator from memory

LDX - Load the X index register from memory

LDY - Load the Y index register from memory

LSR - Logical shift right memory or accumulator

MVN - Move block negative

MVP - Move block positive

NOP - No operation

ORA - Logical OR memory and accumulator

PEA - Push effective address

## *Chapter 12*

PEI - Push effective indirect address  
PER - Push effective relative address  
PHA - Push the accumulator  
PHB - Push the data bank register  
PHD - Push direct page register  
PHK - Push the program bank register  
PHP - Push the processor status register  
PHX - Push the X index register  
PHY - Push the Y index register  
PLA - Pull the accumulator  
PLB - Pull the data bank register  
PLD - Pull direct page register  
PLP - Pull the processor status register  
PLX - Pull the X index register  
PLY - Pull the Y index register  
ROL - Rotate left memory or accumulator  
ROR - Rotate right memory or accumulator  
RSP - Reset status bits  
RTI - Return from interrupt  
RTL - Return from subroutine long  
RTS - Return from subroutine  
SBC - Subtract from accumulator  
SEC - Set C flag  
SED - Set D flag  
SEI - Set I flag  
SEP - Set status bits  
STA - Store accumulator in memory  
STP - Stop the clock  
STX - Store the X index register memory  
STY - Store the Y index register memory  
STZ - Store zero in memory  
SWA - Swap A and H  
TAD - Transfer A to D  
TAS - Transfer A to S  
TAX - Transfer A to X  
TAY - Transfer A to Y  
TCD - See TAD  
TCS - See TAS  
TDA - Transfer D to A  
TDC - See TDA  
TRB - Test and reset bits  
TSB - Test and set bits  
TSA - Transfer S to A  
TSC - See TSA  
TSX - Transfer S to X  
TXA - Transfer X to A  
TXS - Transfer X to S  
TXY - Transfer X to Y  
TYX - Transfer Y to X  
WAI - Wait for interrupt  
XBA - See SWA  
XCE - Exchange carry and emulation bits

## 12.3 The addressing mode suffixes

MASM uses suffixes after the mnemonic to describe the address mode. The suffix may be between zero and three characters long. For example, the implied addressing mode uses no suffix, whereas stack indirect indexed uses SIY. A list of the suffixes used by MASM is given below. Where an address mode may only be used with a particular instruction (eg BRAL) the whole instruction is listed rather than just the suffix.

Addressing mode	Suffix
Immediate	IM op
Absolute	{B} op
Absolute long	{L} op
Direct page	zap
Accumulator	A
Implied	
Direct page indirect indexed	IY zap
Direct page indirect indexed long	LIY zap
Direct page indexed indirect	IX zap
Direct page,X	ZX zap
Direct page,Y	ZY zap
Absolute,X	(BIA)X op
Absolute long,X	(L)AX op
Absolute,Y	AY op
Relative	op
Relative long	BRAL op
Absolute indirect	JMI op
Absolute indirect long	JML op
Direct page indirect	I zap
Direct page indirect long	LI zap
Absolute indexed indirect	IX op
Stack	op
Stack relative	S op
Stack relative indirect	SIY op
Block move	op,op

Letters in curly brackets { } are optional, and the vertical bar | denotes a choice. The word op stands for an operand expression and zap stands for a 'zero-page' expression, ie one whose value is less than 256. Some explanation of the way in which MASM chooses between direct, absolute and long address modes is required. An instruction with no suffix, for example:

### LDA label

will use direct page addressing if label is less than 256. It will use absolute addressing if the most significant byte of label is the same as the data bank for which code is currently being assembled. In other cases (ie where the most significant byte of label is not the same as the assembler's idea of the data bank, or label is a forward reference) the long addressing mode is used.

If it is known that a forward-referenced label will in fact be an absolute location, it is possible to force the assembler into generating an absolute instruction by using the B suffix thus:

### LDAB label

This may also be used with the AX addressing mode; BX may be used to force absolute addressing. It is possible to force the long addressing mode where absolute might otherwise be used by applying the L suffix. For example, LDAL label will generate a three-byte address even if label is an absolute location.

## 12.4 Banks and bank-independence

The address range of the 65SC816 is divided into 256 banks of 64K bytes each. Data and routines accessed within the current bank are called absolute; other addresses are direct page (in bank zero) or long.

It is desirable to make software bank-independent. This means that it does not matter in which bank the software runs for it to execute correctly. It is desirable so that when software is put into EPROM or ROM, it may occupy any of the ROM banks provided. The important condition for bank-independence is that there should be no long references generated. To enforce this rule, MASM has a directive NOLONG which will cause an error to occur if any attempt is made to generate a three byte address. The exception is in references to bank zero. The operating system routine entry points reside here, so all useful programs have to use them. Note that long branches are still permitted after a NOLONG as these are bank- (and indeed position-) independent.

## 12.5 MASM expressions

Operands of instructions and directives in MASM are created using the expression evaluator built in to the assembler. Expressions consist of operands (numbers and strings) and operators which acts on them. Typical operands are labels, hexadecimal constants and quoted strings. Examples of MASM operators are +, -, :AND: and :CSB:.

### 12.5.1 Types of operand

This section describes the form of operands which appear in MASM expressions.

### 12.5.2 Symbols

A symbol in MASM is created by putting a label at the start of line, or by declaring it with the \* directive. The value assigned to a label will be the current value of the location counter. This is initialised to 0 at the start of the assembly and may be set by the ORG directive. For example:

```
ORG &32000  
START  CMPIM 1
```

The label START will be set to the value &32000. Once defined, a label may not have its value altered. An attempt to do so will give a 'Label already defined' error.

The other way of defining a symbol is using the \* directive. This is followed by the value (an expression) to assign to the symbol:

```
VDUVARS * WRKSPC+&100
```

Once defined, a symbol may be used in expressions. In the last example, WRKSPC is a symbol which has already been defined. MASM is a two-pass assembler, so up to one level of 'forward referencing' is allowed. For example, this will work:

```
A * B+1  
B * 1
```

but this will not:

```
A * B+1  
B * C+1  
C * 1
```

Labels in programs are often used as destinations for branches and jumps, or accessing data:

```
LOOP    INC COUNT
       BNE LOOP
       LDAIY PRGPTR
```

### 12.5.3 Numeric constants

Numbers may be specified in decimal or hexadecimal. Hex constants are preceded by the ampersand character &. MASM stores numbers in four-byte, two's complement form. Examples are:

```
1234
&FFEE
&FF0080
```

Numbers may be preceded by the unary operators + and -.

Another form of numeric constant is a string of up to two characters. The first character of the string is the least significant byte of the number. A common example is in immediate operands:

```
PROMPT  LDAIM ">"
        JSRL OUTCH
```

The immediate operand evaluates to 62, which is the ASCII code of >. Another example is

```
CASE    AND .NOT: ("a"-"A")
```

### 12.5.4 String constants

Some directives take string operands. A string is a sequence of characters within double quotes. The double quote character may be included by putting it twice. An example of using strings is embedding text into the program using the = (insert bytes) directive:

```
JSR VSTRING = "Press any key to continue" NOP
```

Several operators are provided for dealing with strings.

### 12.5.5 Variables

MASM is unusual amongst microcomputer-based assemblers, in that it provides variables of three types. These are numeric, string and logical. Variables are similar to labels, in that they can be used as operands in expressions, but differ in that they may have their values altered once they are declared.

A common use of variables is within the MASM WHILE directive structure, and also within macro definitions. As variables are described in their own Chapter, they will not be discussed in detail here.

### 12.5.6 Location counters

MASM keeps track of two location counters, one for the program code and one for variable space. The directives ORG and ^ set the program and variable location counters respectively. ORG must be used before any code is generated in a source file.

The code location counter is accessed through the . character, and the variable location counter through @. An example use of . is:

```
TABSZE * .TABSTRT
```

### 12.5.7 Numeric operators

Operators exist in MASM for manipulating its operands. Those dealing specifically with numbers are dealt with in this section.

The five common arithmetic operations are provided:

+	Add/unary plus
-	Subtract/unary minus
*	Multiply
/	Divide
:MOD:	Remainder after division

As usual, the multiplicative operators have higher precedences than the additive ones, and the unary operators have the highest priority of all. Logical operations are also provided:

:OR:	Inclusive-OR
:EOR:	Exclusive-OR
:AND:	AND
:NOT:	NOT

These act bit-wise on the (numeric) operands, eg 1 :OR: 2 is 3.

There are four binary operators for shifting values. The first operand is the value to be shifted, and the second the number of places to shift. The operators are:

:SHR1:	Shift right	LSB	of left operand
:SHR2:	Shift right	2 LSBs	of left operand
:SHL1:	Shift left	LSB	of left operand
:SHL2:	Shift left	2 LSBs	of left operand

:SHR: and :SHL: are synonyms for :SHR2: and :SHL2: respectively. An example is 1 :SHL: 8 which is &100.

There are another four operators which rotate the left operand instead of shift it. That is, for rotates left the MS bit of the top byte rotated is placed in the LS of the bottom byte, and similarly (but opposite) for rotates right.

:ROR1:	Rotate right	LSB	of left operand
:ROR2:	Rotate right	2 LSBs	of left operand
:ROL1:	Rotate left	LSB	of left operand
:ROL2:	Rotate left	2 LSBs	of left operand

:ROR: and :ROL: are synonyms for :ROR2: and :ROL2: respectively.

Three unary operators extract the lower, middle or upper byte of a three-byte number. These are:

:LSB:	Least significant byte of operand
:CSB:	Middle byte of operand
:HSB:	Third byte of operand

These are usually used for extracting particular bytes from operands which are being used as addressed. Note that the following pairs of expressions are equivalent:

:LSB: op and	op :AND: &FF
:CSB: op and	(op :SHR: 8) :AND: &FF
:HSB: op and	(op :AND: &FF0000) / &10000

The operator ! sets bit 7 (ie the eighth bit) of its operand, so that:

! op is equivalent to op :OR: &80

A common use is to set the top bit of the character at the end of a text string. For example:

= "Press SPAC",!"E"

A routine to print the string would recognise the "E" at the last character as its top bit is set.

The (unary) operator / swaps the bottom two bytes of its operand, so that:

/&123456 yields &125634

Brackets ( and ) may be used to alter the order of evaluation of expressions. For example, \* has higher priority than +, so the expression:

1+2\*3

yields 7. By bracketing the first part, the + can be made to be evaluated first, so that:

(1+2)\*3

will give 9 as its result.

### 12.5.8 String operators

Several operators are included in MASM for manipulating strings. The operands of these operators are string constants, string variables or macro parameters. The last two are discussed in detail later.

The string operators are summarised below using examples.

"AB" :CC: "CD"	yields "ABCD"	(concatenates two strings)
"ABC" :LEFT: 2	yields "AB"	(leftmost n characters)
"ABC" :RIGHT: 2	yields "BC"	(rightmost n characters)
:LEN: "ABCD"	yields 4	(length of string)
:LEN: ""	yields 0	
:STR: &1234	yields "1234"	(operand as a 4 digit hex string)
:STR: 128	yields "0080"	
:STR: (I=0)	yields "F"	(operand as string "F" or "T")
:STR: (I=1)	yields "T"	
:CHR: &41	yields "A"	(operand as an ASCII character)
:CHR: 126	yields "~"	

The last two examples of :STR: usage have logical operands. The logical operators are discussed in the next section.

### 12.5.9 Logical and comparison operators

MASM allows comparisons to be made between numbers and strings. The results of these comparisons are logical, or boolean, values which are used by certain directives (eg ASSERT). Logical results may also be combined using equivalents of the :AND: group of operators.

There are six comparison operators:

- = Equal to
- > Greater than
- < Less than
- <> Not equal to
- <= Less than or equal to
- >= Greater than or equal to

They may be used to compare numbers or strings. Numeric comparisons are signed, four-byte ones. String comparisons are slightly different from those provided in languages such as BASIC. String equality and inequality are as expected: two strings are equal if they are the same length and consist of identical characters in each position.

## *Chapter 12*

In MASM one string is 'less than' another if it is leading substring of it. That is, if the two operands are op1 and op2, op1 < op2 if the first n characters of op2 are the same as op1, where n is the length of op1. The rest of the relational may be defined in terms of this:

```
op1 > op2    if    op2 < op1  
op1 <= op2   if   (op1 < op2) or (op1=op2)  
op1 >= op2   if   (op2 < op1) or (op1=op2)
```

Some examples will make it clearer:

```
"A" < "A"  is FALSE  
"A" <= "A" is TRUE  
"A" < "AB" is TRUE  
"A" > "C"  is FALSE  
"AB" > "A" is TRUE
```

Results from comparisons may be combined using the operators :LOR:, :LEOR:, :LAND: and :LNOT:. These are the logical equivalents of the bit-wise operators described above. Examples of expressions using them are:

```
:LNOT: (SYM1>SYM2)  
(:LEN: "AA" = 2) :LAND: (:STR: SYM = "ABCD")
```

### **12.5.10 Operator precedence**

MASM assigns a precedence or priority to each of its operators. This decreases the number of brackets required to ensure that expressions are evaluated correctly. For example, \* has a higher precedence than + so the expression a+b\*c is evaluated in the 'intuitive' order a+(b\*c), rather than the left-to-right (a+b)\*c.

The table below summarises the precedences of MASM operators, with the higher priorities appearing earlier in the table.

#### *Group One - Unary operators*

Unary +  
Unary -  
Unary /  
! operator  
:NOT: :LNOT:  
:LSB: :CSB: :HSB:  
:LEN:  
:CHR:  
:STR:

#### *Group Two - Multiplicative operators*

\*

Binary /  
:MOD:

#### *Group Three - 'Additive' operators*

Binary +  
Binary -  
:ROR1: :ROR2: :ROL1: :ROL2:  
:SHR1: :SHR2: :SHL1: :SHL2:  
:AND: :OR: :EOR:  
:LEFT: :RIGHT:  
:CC:

***Group Four - Relational operators***

= < >  
 < >= <=

***Group Five - Logical operators***

:LAND: :LOR: :LEOR:

Operators falling in the same group have the same precedence. All binary operators are left-associative so that a-b-c is (a-b)-c.

## 12.6 Common MASM directives

This section describes all of the MASM directives which are not covered in the sections on macros and the WHILE directive.

### 12.6.1 ASSERT - Make an assertion

This directive is used to trap errors which may occur during the assembly. Such errors are not assembly errors as such, but conditions which may have been violated in allocating space for tables etc. The directive is followed by a logical expression. If the expression is TRUE, the assembly carries on; if it is FALSE, assembly stops with an 'Assert fails' error. The test is only made during the second pass. An example is:

ASSERT :HSB: CURFILE = 0

### 12.6.2 BANK - Set assumed data bank value

The 65SC816 uses 'absolute' addressing for data accesses in its current data bank (held in the B register) and 'long' addressing for other accesses. The assembler has to be kept informed of the current value of B so that it can generate the correct instructions. By default it assumes that B is zero, and will generate long instructions for data accesses outside of bank zero.

To set the data bank register to, say, B4BASIC, and keep the assembler up to date, the following might be used:

LDAIM B4BASIC ; Assume 8 bit mode PHA PLB BANK B4BASIC

The NOBANK directive is used to stop the assembler from assuming any bank, and make it always generate long instructions.

### **12.6.3 EMODE - Assemble for emulate mode**

When the instruction

**EMODE 1**

is encountered, the assembler enters 'emulation' mode. That is, it will only generate instructions which may be executed when the 65SC816 is in emulation mode. This entails preventing the generation of instructions which use two-byte A, X or Y registers or memory locations. The directive is usually used in conjunction with the instructions

**SEC**

**XCE**

which put the processor into emulation mode. Use of the RMODE directive is not allowed in emulation mode.

After the directive

**EMODE 0**

'native' mode is entered and two-byte instructions may be generated. The default state of the assembler is 'native' mode. Native mode is entered using the instructions:

**CLC**

**XCE**

### **12.6.4 END - end of assembly**

This marks the end of the source file, and of the current pass of the assembly. Only one source file should include this directive. It is usually made the last line of the file, as any subsequent text is ignored. There must be a carriage return after the END directive, otherwise a 'Line too long' error will be given.

### **12.6.5 LNK - link next source file**

This directive is followed by a string which is the name of the next source file to assemble. All symbols, location counters, macros etc are preserved between the two files. The directive is followed by a filename which may be up to ten characters long. An example is:

**LNK PMON02**

### **12.6.6 LONG - Allow long instructions**

This directive allows the assembler to generate 'long' instructions to any bank. This means that there is no guarantee that the code will be 'bank-independent'. This is the default state of the assembler.

### **12.6.7 NOBANK - Prevent assumed bank register**

By default the assembler assumes that the data bank register in the processor will be set to &00 when the code is executed. Thus it generates absolute addressing instructions for references to data in bank &00 and long addressing for other instructions. This is desirable as the operating system 'indirections' are in bank zero. When the directive

**NOBANK**

is executed, MASM starts to generate long instructions for all data access instructions, as it does not assume any particular data bank.

### 12.6.8 NOLONG - Disallow long instructions

When writing ROM software for Communicator, programmers should aim to make it 'bank-independent', that is capable of running in any bank without alteration. A necessary property of such code is that it does not contain any 'long' references to either programs or data, except to operating system routines in bank 0. When the directive:

**NOLONG**

is encountered, the assembler will cause an error to be generated when an attempt is made to generate any long reference except to bank 0. This checking is turned off with LONG.

### 12.6.9 OPT - set listing options

This directive is used to set several options while assembling. The directive is followed by an expression, only the lowest four bits of which are used. The result of setting each bit is:

- Bit 0 (1) Turn listing on if enabled by PRINT ON.
- Bit 1 (2) Turn listing off.
- Bit 2 (4) Start a new page.
- Bit 3 (8) Restart line numbers from 0001.

Thus the directive:

**OPT 4+8**

will start the listing at a new page from line 0001, assuming listing is enabled. Note that the listing may be enabled and disabled from MASM's command mode using the PRINT ON/OFF command.

### 12.6.10 ORG - set code origin

The ORG directive sets the value of the program location counter. At most, one such directive should occur per source file. The ORG cannot occur after code has been generated. The default value of ORG is &000000. An example is:

**ORG &FF8000**

### 12.6.11 RMODE - Reset one-byte register bits

The assembler has to be kept informed of whether the code to be generated will be executed by the 65SC816 in one-byte or two-byte register/memory mode. The RSP and SEP instructions control the appropriate bits in the status register. See the Introduction to Communicator Programming for details of the X and M status bits.

The RMODE directive takes the same operand as the latest RSP instruction, to inform the assembler that a mode change has been made. The instruction and directive usually appear in pairs, as in this example:

**RSPIM &30;16 bit A and X/Y  
RMODE &30**

### **12.6.12 ROUT - Declare start of routine**

The ROUT directive marks the start of a routine which will contain local labels. A local label is a two digit number which appears in the label field, and may optionally be followed by the name of the routine. The current routine name is the label on the last ROUT directive line. References to local labels are # followed by the label number, and optionally the routine name. The small routines below illustrate most elements of routines and local labels:

```
MULT    ROUT
       LDAIY 33
       BNE #00
       LDAIM 0
00MULT   RTS
DIV      ROUT
         DEX
         BNE #00DIV
         LDAIM &FF
00      RTS
```

In addition, there may be up to two letters following the # and before the digits. The first letter is:

Nothing	Search backward and forward for the label
B	Search backward for the label
F	Search forward for the label

The second letter is:

Nothing	Search at this macro level and above
A	Search at any macro level
T	Search at this macro level only

This refers to the fact that when local labels are created, MASM saves information relating to the current macro level. This will be 0 (global) if the label was defined outside of a MACRO definition, 1 inside the first level of macro invocation, 2 in a macro called by another macro etc. Note that a MACRO directive does an implicit ROUT.

Usually, the macro level information does not have to be specified and the default of # followed by two digits suffices in local label references.

If references to macros at levels other than the current one are never required, MASM can be prevented from saving level information for local labels by issuing the command:

**MLEVEL OFF**

from the MASM command prompt. This prevents the possibility of 'Local label overflow' errors from occurring.

### **12.6.13 SMODE - Set one-byte register bits**

The assembler has to be kept informed of whether the code to be generated will be executed by the 65SC816 in one-byte or two-byte register/memory mode. The RSP and SEP instructions control the appropriate bits in the status register. See the Software Writer's Guide for details of the X and M status bits.

The SMODE directive takes the same operand as the latest SEP instruction, to inform the assembler that a mode change has been made. The instruction and directive usually appear in pairs, as in this example:

```
SEPIM &20 ;8 bit A
SMODE &20
```

### 12.6.14 TTL - set listing title

This directive is followed by a line of text. This is printed at the top of every page in the assembly listing. An example is:

TTL PMON - Machine code monitor, file 03

### 12.6.15 [ - start conditional assembly section

The [ (if) directive is followed by a logical expression. If the expression is TRUE, subsequent code is assembled as usual. If the expression is FALSE, lines of source code are ignored until a matching ] (else) or ] (endif) directive is found. Only lines assembled in conditional assembly will appear on the listing. Examples of conditional assembly are:

```
[ SCNDPROC=1
RDTUBE ; Use macro instead
]
LDAIY PTR
]

[ DEBUG=1
JSR VSTRING
= "Entering 'FIND'",&EA
]
```

### 12.6.16 ! - Print message

This directive is followed by a number and a string. The string is a message which is always printed on a line of its own during the second pass of the assembler. If the number (or rather, numeric expression) is not zero, then the message will also be printed on the first pass, and the assembler will stop with the error 'Stopped'. Both expressions must be fully defined in the first pass, ie must not contain any forward references. An example is:

! 0,"This is printed on the second pass"

Note: there is also an operator ! which sets top bit of the least significant byte of its operand. This should not be confused with the ! directive.

### 12.6.17 \* - Define a symbol

This directive creates and assigns a new label. The value of the label can be an expression, which must be fully defined by the time the directive is encountered on the second pass. Examples are:

```
OSBYTE * OSWORD+3
OSWORD * &FFF1
```

### 12.6.18 ^ - Set variable code origin

This directive is followed by an expression which gives the new value for the variable counter origin. There may be as many of these as required in a file. The variable space counter and its associated directive # are only used to defined addresses of workspace - no code is actually generated.

### 12.6.19 # - Reserve (variable) memory bytes

The # (hash) directive is followed by an expression. The variable location counter @ is incremented by that amount. Almost always, a label is put on the line, so that the address of the first byte just reserved is known. An example is:

ZPTEMP # 16

## *Chapter 12*

This reserves 16 bytes and sets ZPTEMP to the address of the first of them. Another use for # is to keep a sequence of symbols in consecutive order, without having to renumber every time a label is added or deleted:

```
^ 0  
IF # 1  
ELSE # 1  
ENDIF # 1  
GOTO # 0  
GOSUB # 1
```

The GOTO # 0 line does not increment the code pointer, so GOTO and GOSUB will be given the same value.

### **12.6.20 = - Insert bytes into code space**

This embeds values into the code space, incrementing the code location counter . with each byte. The directive is followed by a list of comma-separated expressions. Each expression may be a numeric, in which case the least significant byte is used, or a string, in which case all of the characters in the string are embedded. Examples are:

```
= "Action : ",NOP  
= TABX-START, TABY-START
```

### **12.6.21 & - Insert double bytes into code space**

This has a similar effect to the = directive. Each expression following the directive occupies two bytes in the code space. The two lines below are equivalent:

```
= &12,&34  
& &3412
```

### **12.6.22 % - Fill code space with zeroes**

The % directive fills the object code file with zero bytes from the code location counter. The number of bytes filled is given in the expression following the directive. For example, the two lines below have the same effect:

```
% 10  
= 0,0,0,0,0,0,0,0,0,0
```

## 12.7 Variables

A useful facility in MASM is the provision of variables. These are used mainly in conjunction with the MACRO, WHILE and [ (if) directives. The first two are described in later sections; the last was described in the last section.

### 12.7.1 Types of variable

There are three types of variable: arithmetic, logical and string. A variable is a standard symbol preceded by a \$, for example:

```
$COUNT
$TURBO
$FLAG1
```

To declare a variable, one of the directives GBLA, GBLL or GBLI is used, depending on its type. Examples are:

```
GBLA    $COUNT
GBLL    $FALSE
GBLL    $TRUE
GBLL    $TURBO
GBLS    $ALIST
```

### 12.7.2 Assigning variables

Once defined, a variable may have a value assigned to it. Again, there are three directives, one for each type. Examples are:

```
$COUNT  SETA    0
$COUNT  SETA    $COUNT+1
$FALSE   SETL    I=0
$TRUE    SETL    I=1
$TURBO   SETL    $TRUE
$ALIST   SETS    "Addresses: "
$ALIST   SETS    "$ALIST" :CC: :STR: $LABEL
```

Variables, unlike normal symbols, may be assigned as often as required during the assembly. On the other hand, they may just be used as constants for conditional assembly directives, as in:

```
[      $TURBO
LDAIY  PTR
|
LDAAY  BUFF
]
```

This enables different versions of a program to be generated by changing a few definitions in a header file.

### 12.7.3 String variable substitution

When arithmetic and logical variables are used in expressions, their values are inserted as if a constant of that type had been used. For example:

```
$ADDR  SETA    &8000
      LDAAX  $ADDR+&20
```

is interpreted exactly as:

```
LDAAX    &8000+$20
```

String variables act slightly differently, in that including a string variable in an expression (or anywhere on a line in fact) has the effect of substituting the string's value at that position in the line. For example:

```
$ASTRG  SETS    "MESSAGE:"  
        =      $ASTRG
```

is incorrect, as once the string has been substituted in the second line, the line will read:

```
=      MESSAGE:
```

That is, the quotes are not part of the string variable and are therefore not substituted. There are two correct solutions:

```
$ASTRG  SETS    """MESSAGE:"""  
        =      $ASTRG  
$ASTRG  SETS    "MESSAGE:"  
        =      "$ASTRG"
```

The latter is the preferred method. As mentioned above, a string variable may appear anywhere in the line. For example in

```
$MNM   SETS    "LDA"  
$MNM   TAB
```

the second line is the same as

```
LDA    TAB
```

If a \$ character is required before a string which is the same as a string variable name, it must be given twice as in:

```
$FLAG   GBLS    $FLAG  
$FLAG   SETS    "FALSE"  
! 0,$$FLAG$$
```

which will display \$FLAGS during the second pass. In fact \$\$ may be used anywhere to stand for \$.

#### 12.7.4 Local variables

The variables discussed up until now have been global - they are present during the whole of the assembly from the point they are created. There is another type of variable, local, which is declared using LCLA, LCLL or LCLS and exists only for the invocation of the macro in which it is created. Local variables are discussed in the next section.

## 12.8 Macros

A macro is a list of instructions which may be inserted into the source text at any point by citing the macro's name as an instruction.

### 12.8.1 Defining macros

To define a macro, the directives MACRO and MEND are used. MACRO marks the start of the macro definition and MEND its end. The general format of a definition is:

```
MACRO
$LABEL  MNAME
$LABEL      ;Macro body
           ;More macro body
MEND
```

MNAME is the name of the macro, and is used when the macro is invoked later in the assembly. \$LABEL is a (local) string variable which is automatically assigned to the label field on the invoking line. A practical example of a macro is:

```
MACRO
$LABEL  BYTEAX
$LABEL  SEPIM    &30
       SMODE    &30
MEND
```

The macro contains the instruction and directive required to put the processor and assembler into one-byte mode. Examples of invoking it are:

```
BYTEAX
LOOP   BYTEAX
```

In the second example \$LABEL would have the value "LOOP", so lines actually assembled would be:

```
LOOP   SEPIM    &30
       SMODE    &30
```

### 12.8.2 Macro parameters

Macros are more useful when the act of invoking them can have a different effect each time. To this end, macro definitions may take string parameters which are substituted for actual values when the macro is invoked. As an example, the macro below generates the code to call an operating system routine in bank zero. If this routine is \$OSB, a count is incremented, and the calling address added to a string of addresses.

```
MACRO
$LABEL  CALLOS  $ADDR
       (
       $ADDR=$OSB
$OSBCNT SETA    $OSBCNT+1
$SOSBLST SETS    "$SOSBLST" :CC: :STR: .
       )
$LABEL  JSRL    $ADDR
MEND
```

## *Chapter 12*

The parameter is \$ADDR, and acts exactly as a string variable whose value is the corresponding string on the invoking line. Consider first the case when the routine is not £OSB, for example:

```
RESTRT CALLOS £MM
```

The instructions assembled would be:

```
RESTRT JSRL £MM
```

The [ will fail, and the MEND will be the next instruction to be executed. Now the case where the routine is £OSB. The assembled code will be

```
$OSBCNT SETA $OSBCNT+1  
$OSBLST SETS "$OSBLST" .CC: .STR: .  
JSRL £OSB
```

The two variables \$OSBCNT and \$OSBLST are assumed to be globals which were declared before the first invocation of CALLOS:

```
GBLA $OSBCNT  
GBLS $OSBLST
```

Note that as the variables are initialised to zero and "" respectively, there is no need to follow the definitions by SET directives. In the next section, a WHILE loop will be used to print the contents of \$OSBLST using the ! directive.

If it is required to follow a parameter immediately by a letter or digit, . may be used to avoid ambiguity. For example, if \$INS is a parameter, then the line

```
$INSIM
```

would be taken as the variable \$INSIM rather than \$INS followed by the address mode IM. To separate them, use

```
$INS.IM
```

The . is ignored, but marks the end of the parameter name.

### 12.8.3 Default parameters and missing parameters

There may be as many parameters as desired to a macro. The names are separated by commas in the definition line, as are the strings to be used for the particular invocation. Examples of corresponding definitions and calls are:

```
RORN SCOUNT,$LR  
RORN 4,L  
RORN 2,R  
SWAP $REG1,$REG2,$USING  
SWAP A,X,Y  
SWAP X,Y,A
```

Sometimes, not all of the parameters have to be given values in the invocation. To miss out a parameter, just type the comma to go on to the next parameter. For example, to miss out the first, second, and third parameters respectively of SWAP use:

```
SWAP ,X,Y  
SWAP A,,Y  
SWAP A,X,
```

When a parameter is missed out like this, the corresponding parameter is a null string.

It is also desirable to have default values for parameters. These are specified in the definition using = after the parameter name:

```
MACRO
SWAP      $R1="X",$R2="Y",$USE="A"
```

To use a default value in an invocation, I is used for the parameter:

```
SWAP      I,J
```

#### 12.8.4 Nesting macro calls

Macro calls may be nested. That is, a macro called during the main assembly may call another macro to generate code. In fact, macros may call themselves, that is they can be recursive. Usually, it is better (ie more efficient) to use WHILE loops instead of recursion.

Every time a macro is invoked, the macro depth level is increased by one. Whenever a MEND or MEXIT directive is encountered, the macro depth level is decreased by one. When local labels are created, the current macro level is stored with the label's information. This enables the programmer to refer to labels which were created at particular macro levels (see section 3).

#### 12.8.5 Local variables

Sometimes it may be necessary to store values which are only required for the duration of a particular macro call. It is possible to use global variables inside macro definitions, but it is better to use local variables which won't interfere with any previously-defined global values. Local variables are declared using the directives LCLA, LCLL and LCLS. These correspond directly to GBLA, GBLL and GBLS. The SET directives are still used to assign values to local variables.

The most important use of local variables is in making copies of parameters which may be changed. Because of the way in which MASM performs substitution of parameters in source lines, parameters may not be used with SET directives. The following, therefore, is incorrect:

```
MACRO
TEST      $PARM
$PARM    SETS      "$PARM" :CC: "_1"
....
```

The following would have to be used instead:

```
MACRO
TEST      $PARM
LCLS      SP
$P      SETS      "$PARM" :CC: "_1"
....
```

Note that LCL directives have to appear before anything else in the body of a macro.

Another use of local variables is to convert parameters (which are effectively pre-initialised local strings) into numeric values. For example, suppose a macro takes a parameter which will usually be an arithmetic expression. If this parameter is subsequently used as an operand, unpredicted results might occur. The macro below, RWORD, takes an expression as its parameters and reserves that number of words (ie double-bytes) in the variable area:

## *Chapter 12*

```
MACRO
$LABEL RWORD    $WORDS
$LABEL          2*$WORDS
      MEND
```

With the call:

```
RWORD    SIZE
```

the macro has the desired effect and reserves 2\*SIZE bytes. However, with the call:

```
RWORD    SIZE+EXTRA
```

The number of bytes reserved is 2\*SIZE+EXTRA, not 2\*(SIZE+EXTRA) as required. One way around the problem would be to re-write line three of the macro as:

```
$LABEL          2*($WORDS)
```

However, if the parameter were to be used several times in the body of the macro, an alternative using a local arithmetic variable would be better:

```
MACRO
$LABEL RWORD    $WORDS
      LCLA   $$ZE
$SZE  SETA    $WORDS
$LABEL          2*$SZE
      MEND
```

### **12.8.6 Macro substitution method**

The following algorithm is used to substitute variables and parameters when a macro is called:

- Substitute macro parameters throughout the macro
- For each line do the following:

If it is not a directive, substitute for string variables in the line

else if the directive is not LCL or GBL, substitute for strings after the directive

else (for LCL or GBL), do nothing

- Next line, until MEND or MEXIT

### **12.8.7 The MEXIT directive**

If a macro contains some conditional assembly or a WHILE loop, it may be required to terminate the macro from within the construct. To do this, MEND may not be used, as there must only be one MEND per MACRO. Instead, use MEXIT, as in:

```
MACRO
$LABEL MESG    EXT
      (
      " EXT"=""
      MEXIT
      )
      JSR     VSTRING
      =
      " EXT",&EA
      MEND
```

## 12.9 The WHILE loop

In the last section, it was suggested that macros could be called recursively, and that WHILE loops were usually a better alternative.

The WHILE directive has the following form:

```
WHILE <condition>
    ; Body of while
WEND
```

The <condition> may be any expression involving the relational and logical operators described in section two. It may also be a logical variable. As long as the condition yields a TRUE result, the lines between the WHILE and WEND will be repetitively assembled. As the condition is evaluated at the start of the loop, the lines may be assembled zero times (if the condition is initially FALSE).

The simple WHILE loop below assembles the lines:

```
STAAX    &3000
STAAX    &3100
.....
STAAX    &7E00
STAAX    &7F00
```

This would be 80 lines of code if assembled without the loop listed below:

```
$ADDR    GBLA    $ADDR
$ADDR    SETA    $3000
        WHILE   $ADDR<&8000
        LDAAX   $ADDR
$ADDR    SETA    &ADDR+&100
WEND
```

The next example prints the contents of the string variable \$OSBLST which was used to illustrate a use of macros in the last section. The routine is defined as a macro, to make it more general. There are two parameters: a message to print before the list is printed, and a string containing a list of four-digit hex numbers to be printed one per line.

```
MACRO
PRINTST $MESG,$STRG
LCLS $STR
OPT 2
$STR    SETS $STRG
        ! 0,"$MESG"           ; Printing off
        WHILE :LEN: "$STR" > 0
        ! 0,"$STR" :LEFT: 4
$STR    SETS "$STR" :RIGHT: (:LEN: "$STR" -4)
        WEND
        OPT 1                 ; Print the message
        MEND
PRINTST OSB addresses:$OSBLST
END
```

## 12.10 Using MASM

The preceding sections described MASM in terms of the facilities it provides from the source file. This section describes the MASM command mode - what the user sees after calling-up the program.

### 12.10.1 Starting MASM

The assembler is called using the command

\*MASM

This loads the main assembler into the second processor and some subsidiary code into the IO processor (BBC Microcomputer). The MASM prompt is:

Action:

Typing HELP or H. in response to this will produce a list of commands which may be typed in response to the prompt. The commands are:

ASM	Assemble a source file.
GET	Load an object file or files.
HELP	Print a list of MASM commands.
LENGTH	Set the listing page length.
MLEVEL	Allow local label information.
PRINT	Enable assembly listing.
SAVE	Save an object file.
STOP	Make the assembler stop on errors.
SYMBOL	Print a symbol table listing.
TERSE	Enable listing of conditioned-out code.
WIDTH	Set listing page width.
XREF	Enable generation of cross-reference information.

A command may be abbreviated to the minimum string which identifies it uniquely, eg A for ASM.

### 12.10.2 Preparing to assemble

Although a source file may be assembled straightaway using the ASM command, there are other commands which affect the assembly, and which the user may wish to issue first. These are:

LENGTH	Set listing page length.
MLEVEL	Allow local label information.
PRINT	Enable assembly listing.
STOP	Make the assembler stop on errors.
TERSE	Enable listing of conditioned-out code.
WIDTH	Set listing page width.
XREF	Enable generation of cross-reference information.

These commands are dealt with in subsequent sections.

### 12.10.3 LENGTH

If an assembly listing is produced, it defaults to 66 lines per page of listing including title information. This is suitable for most types of printer. Alternate settings may be specified by following the LENGTH command by a number between 0 and 127. For example:

LENGTH 72

At the end of a page, MASM generates a form feed - this also clears the screen.

### 12.10.4 WIDTH

The assembly listing consists of lines which default to 72 characters. As with LENGTH this is a suitable value for most printers, but may be varied between 0 and 127. A suitable choice for the screen is:

WIDTH 39

### 12.10.5 PRINT

By default, no assembly listing is produced by MASM. The command:

PRINT ON

will cause one to be produced for subsequent assemblies and:

PRINT OFF

will turn listing off again. Note that the listing may be selectively enabled and disabled from within the assembly (assuming PRINT ON has been issued) using OPT 1 and OPT 2 respectively.

Enabling the listing does not automatically select and enable the printer: you must do this using the appropriate \*FX and CTRL commands. For example, to make the listing go to the serial printer port to a printer which needs line feed characters, use:

Action: \*FX5,2 (Select serial printer)

Action: \*FX6 (Don't ignore line feeds)

Action: ASM FILE <CTRL B> <RETURN>

.....

.....  
Action: <CTRL C>

### 12.10.6 STOP

If the assembler encounters a non-fatal error during assembly, it usually carries on with the pass. However if the command:

STOP ON

is issued before the the ASM command, any error will cause the assembly to be aborted immediately. The command:

STOP OFF

changes back to the default effect.

### **12.10.7 TERSE**

MASM usually inhibits the listing of code which is in part of a conditional assembly which failed. For example:

```
[ 1=1 ;Line 1  
| ;Line 2  
]
```

Usually only Line 1 would be listed. However, if the command:

**TERSE OFF**

is issued before the assembly, both lines would be listed, although of course only the first would be assembled. The default action can be re-instated using:

**TERSE ON**

### **12.10.8 MLEVEL**

Whenever the assembler encounters a macro call, it saves some information in a table called the 'macro level table'. This is used when local labels are referenced. If there are many macro calls in a source file, the macro level table may become full, even if there are no local labels.

The possibility of a 'macro table full' error can be avoided by giving the command:

**MLEVEL OFF**

before the assembly. However disabling this feature will mean that all labels are taken to be at the same (global) level.

### **12.10.9 XREF**

One of programs to be supplied with MASM is a cross referencer. This uses a file produced by MASM during the assembly to generate cross-reference listings of labels. By default, MASM will not produce a cross-reference file, but after the command

**XREF ON**

all subsequent assemblies will. The feature may be turned off again using:

**XREF OFF**

Use of the cross-referencer is described section 8.

### **12.10.10 Assembling a file**

The central command of MASM is the one given to actually assemble a file or files. The command

**ASM VDU01**

will assemble a file called VDU01. The object code will be placed in a file called X.VDU01, so the directory X should exist if you are using the ADFS or Network filing systems. The load address attributes for the object files are set to the value of the code location counter at the start of the assembly of the corresponding source file.

MASM performs two passes. A message giving the pass number is printed at the start of each one. During the first pass, errors messages may be printed out and, if they are fatal, will cause the assembly to be abandoned. Non-fatal errors will not cause the assembly to stop unless the command STOP ON had been issued before the assembly.

If no errors were detected in the first pass, the second pass will start, and an assembly listing made if PRINT ON was issued.

After giving an ASM command the user is asked for a

Macro library name:

This is the name of a file containing only macro definitions and the directives END and LNK. It is read in only once to save time. If there is no macro library, press RETURN without giving a filename. Macro definitions may, of course, be included in normal files, but will be read twice, once for each pass.

A symbol table may be generated at the end of the assembly using the SYMBOL command (see below).

At the end of a successful assembly there will be a set of object files x.<name> for every source file <name> which was assembled and generated code. These object files may be loaded and saved as one file using the GET and SAVE commands (see below).

### 12.10.11 SYMBOL

To obtain a list of symbols defined during the assembly, use the SYMBOL command. There are three varieties:

SYMBOL A  
SYMBOL N  
SYMBOL S

The first lists symbols in alphabetical order, the second in numerical order. The third type prompts for a symbol name and then prints its value. In a symbol listing, symbols which were declared but not used are marked with an asterisk \*. The symbol listing uses the same WIDTH and LENGTH as the ASM command.

### 12.10.12 GET

The GET command is used to load file into memory at a specified address. After the command

GET file

the user is prompted for:

New, Own or Previous address (N/O/P)

The responses are:

- N - Load at an address given by the user
- O - Load at the address given in the file information
- P - Load immediately after the last file loaded

Note the files always load into the second processor, even if an IO processor address is specified. If the last two characters of the filename are digits, the user will be prompted for an

Offset

This is the number of the last file to load in the sequence. For example:

Action: GET X.VDU01  
Offset: 05

will load X.VDU01 to X.VDU05 at contiguous addresses. A suitable area for loading the object files is &7800 to &B800 which enables up to 16K bytes of code to be loaded at once. Care should be taken not to load above &B800 as this will overwrite the start of the assembler.

## *Chapter 12*

Note: Whenever MASM prompts for an address, a general expression may be entered as described in section two. This enables symbols to be used, for example:

Address:START+DISP

### **12.10.13 SAVE**

Once a set of object files has been loaded into memory, the final file may be created by GETting them as described above and then saving the composite file using the SAVE command. In response to the command

SAVE file

MASM produces four further prompts:

Start address:  
End address:  
Load address:  
Exec. address:

The first two are the addresses in the second processor of the file to be saved. As usual the end address is one greater than the last byte to be saved.

The load address is where the file is to be re-loaded, ie the load address to be put into its file information. The exec. address is the execution address to be stored in the file information.

## **12.11 Using XREF**

The XREF utility operates on file created during assembly by MASM. See section 7 for a description of MASM's XREF command. The purpose of XREF is to create listings of where in the source program symbols were used and defined. It provides comments for symbols which were defined but never referenced, and warning for labels which were given to the cross-referencer but were not present in the source file.

To call the cross-referencer, the command

\*XREF

is used. This will usually be entered from MASM's command prompt, after a successful assembly. The prompt for XREF is:

Action:

Like MASM, XREF responds to the word HELP by listing the available commands. The output printed by XREF is:

ADD  
CLEAR  
HELP  
INIT  
LIST  
RESULT  
SUMMARY  
XREF

A command can be activated by typing the appropriate command name in upper or lower case. The command can be abandoned at any time by pressing ESCAPE; this will return to HELP list followed by the prompt.

Operating system commands may be entered in response to the Action: prompt by prefixing them with a \*.

Below is a description of each of XREF's commands. The first three relate to the entry of symbols which are to be referenced, the last three are to do with the actual cross referencing.

### 12.11.1 ADD

The first thing XREF needs is a list of symbols to look up. This is entered using the command ADD. Once the command has been issued, XREF will repeatedly prompt as follows:

symbol:

The response should be a symbol name whose occurrences in the source are to be listed. Symbols' names correspond in format to those used by MASM, ie up to six alphanumeric (plus \_ and £) characters, starting with a letter.

XREF's response to an illegal symbol name is:

Bad symbol

To end the symbol entry, the illegal name . is used to return to the main prompt. The maximum number of labels which XREF can handle at once is 1024. This should be sufficient for most purposes.

### 12.11.2 CLEAR

CLEAR will remove a single named symbol from the symbol table. There are two usages of clear. The command on its own will cause the prompt 'symbol:' to be printed. In response the name of the symbol to be deleted from the table should be entered.

The alternative usage is to follow the CLEAR command immediately by the symbol's name. In this case, the 'Action:' prompt is printed immediately.

### 12.11.3 INIT

This command will restart the program by clearing out the symbol table. After an INIT, the program is in the same state as when it is called using the \*XREF command.

### 12.11.4 LIST

This command shows the current contents of the symbol table. The symbol names are printed in a simple list across the page, for example,

ERROR FIN LOOP TEST ZEROP

### 12.11.5 XREF

This command causes a MASM-created cross-reference file to be read in, so that the symbols in the table may be matched. The prompt for the command is:

Xref file:

The response should be the name of a file created using MASM's XREF option. XREF will read this file and look up the names in the symbol table. It will then print a summary and return to the prompt stage. If the name given does not refer to a valid cross-reference file then a 'Read error' will be generated.

After the cross-reference has been performed, the results are printed automatically. See section 8.6 for a description of the results listing.

### **12.11.6 RESULT**

This command may be issued after cross-referencing a file with command XREF. The results listing is in two parts. The first part is an alphabetical list of all labels in the symbol table which appear in the source file, giving the line numbers where they are defined and used. The second part is a summary which may be disabled. It gives comments such as 'symbol not used' and 'symbol use only once', and warnings such as 'symbol not in source file'.

If there are many symbols in the table and/or the file which produced the cross-reference file was very large, there may be many entries printed out. A printer may be enabled by typing CTRL B just before RETURN is pressed, and disabled using CTRL C.

### **12.11.7 SUMMARY**

The printing of a summary can be disabled using this command. If you type SUMMARY, the following prompt will appear:

Summary? (Set/Unset):

The reply is S to enable the summary and U to disable it.

## 12.12 MASM errors

This section explains the errors generated by MASM during an assembly. There are two kinds of error, fatal and non-fatal. Fatal errors always cause the assembly to be terminated immediately. Non-fatal errors only cause the assembly to finish if STOP ON has been given, otherwise MASM continues to the end of the current pass, and then stops with a report of the number of errors encountered.

The section describing the fatal errors also covers those errors which may occur in MASM command mode, eg 'Bad command'.

### 12.12.1 Non-fatal errors

#### *Bad bank*

This occurs when an attempt is made to use a BNK directive with an operand which is not between 0 and 255.

#### *Bad directive use*

This occurs when a symbol used in a directive's operand has not been defined yet.

#### *Bad label*

A label must be between one and six characters long and may only contain letters, digits, \_ or \$. It must start with a letter, \_ or \$. Anything in the label field of a line not obeying these rules will cause this error to be generated.

#### *Bad local label number*

These must be two-digit decimal numbers.

#### *Bad mode*

This occurs when an attempt is made to use either the RMODE or SMODE directive when the assembler is in emulation mode (ie after EMU 1).

#### *Bad opcode*

This is caused by an unrecognised instruction/directive in the opcode field of a line of assembly source.

#### *Bad operand*

This is reported when an expression contains an object which should be an operand, eg a label or variable, but isn't recognised as such.

#### *Bad operator*

This is reported when an expression contains an object which should be an operator, eg + or :CC:, but isn't recognised as such.

## *Chapter 12*

### *Bad OPT*

The expression after the OPT directive should lie in the range 0-15.

### *Bad ORG*

The operand of ORG should be an address between &000000 and &FFFFFF. An attempt to use an expression which evaluates outside this range will give a 'Bad ORG' error.

### *Bad routine name*

The name of a routine should conform to the same rules as any other label.

### *Bad string length request*

This occurs when an attempt is made to create a string variable whose length is greater than 127 characters.

### *Bad zero page value*

This is given when an address operand which should be less than 256 isn't, eg LDAZX 4321 will give this error.

### *Badly defined manifest symbol*

This occurs when a symbol is forward-referenced too indirectly for the assembler to resolve it by the end of the first pass. An example of code that might cause this is:

	LDA A	
A	*	B
B	*	1

In this example, the symbol A is defined by a forward reference to B. Thus B will be known by the end of the first pass, but A won't be known until the end of the second pass. The LDA instruction cannot therefore know the value (and hence number of bytes of) its operand, and cannot be assembled.

### *Division by zero*

This occurs during expression evaluation when the right hand operand of a divide or :MOD: operator is zero.

### *Double defined variable*

A variable defined using the GBL directives may only be defined once. Thereafter it may be set to different values using SET directives.

### *End of line missing*

This occurs when the end of the source file is encountered halfway through a source line. All lines in MASM source should be terminated by a carriage return.

### *Expansion line too long*

When string variables and macro parameters are substituted, they may make the source line longer than it was. If the line grows to more than 250 characters after substitution, this error will be given.

### *Label already defined*

Once a label has been set to a certain value, it retains it for the rest of the first pass and all of the second pass. An attempt to redefine a label that has already been set will yield this error.

***Line too long***

A source line must be less than 255 characters long, and terminated by a carriage return character.

***Missing /***

If the assembler comes across an 'else' (/) or 'endif' and has no corresponding 'if' (), this error is given.

***Missing WHILE***

This error occurs when a WEND directive is encountered and the assembler has not had a WHILE to match it with.

***No current macro***

This is given when an attempt is made to define local variables outside of the macro level. Only global variables may be defined outside of macros.

***Offset to <label> out of range***

This occurs in branch instructions when the destination label is more than 129 bytes after or 126 bytes before the first byte of the branch.

***Syntax error***

This means that MASM is unable to make any sense of a line of source code.

***Too late for LCL directive***

The LCL directives must be the initial lines of a macro definition. Inserting other directives or code-generating lines between the MACRO and LCL will cause an error to be given.

***Too late for ORG***

The ORG directive must be given before any code has been generated by the source in the current file.

***Type mismatch***

This is given when strings and numbers are mixed illegally in expressions, eg 123 :CC: "MASM" (both operands of :CC: should be strings).

***Unknown symbol***

This error is caused by an attempt to access a symbol which has not been defined.

***Unknown variable symbol***

This is caused in a similar way to the last error, but refers to symbols that are preceded by the variable sign '\$'.

## **12.12.2 Fatal errors**

### ***Assembly stopped***

This is printed at the end of the first pass of an assembly in which one or more errors were detected.

### ***ASSERT failed***

This is caused when the condition part of an ASSERT directive did not yield a TRUE result. It warns you that something you assumed to be true about the assembly was not.

### ***Bad command***

This is given when MASM does not recognise a command typed in response to the 'Action:' prompt. The HELP command will cause a list of valid commands to be displayed.

### ***Bad expression***

When one of the expressions given in response to the SAVE command's prompts cannot be understood by MASM (eg contains an identifier it doesn't know about), this error is given.

### ***Bad macro definition***

This is caused by an error occurring between the MACRO and MEND parts of a macro definition, eg a badly formed MACRO line.

### ***Bad macro library***

If the response to the ASM command's 'Macro library:' prompt does not yield a string that can be used as a filename, this error is given.

### ***Bad nesting***

This is caused by the end marker of one 'structure' being encountered when another type of structure is still open. Such an error is roughly analogous to ending a FOR loop with an UNTIL in BASIC. The structures of MASM are WHILE..WEND, MACRO..MEND, and [...] .

### ***Bad offset***

This is caused when the response to the GET command's 'Offset:' prompt does not yield a suitable number.

### ***Bad option***

This is caused by specifying an illegal option in one of the MASM commands. For example, only 'A', 'N' and 'S' are valid SYMBOL options.

### ***Bad value***

This occurs in the WIDTH and LENGTH commands when an illegal number is given.

### ***Can't open***

This means that MASM can't open the XREF output file, for example, because a locked file of the same name already exists.

### ***Code overwriting source***

This occurs when the amount of code generated by a program is much larger than the source code generating it. It should never occur if MASM is used properly.

***Doubly defined MACRO***

This error is given when the same macro name occurs in more than one MACRO directive. Macros should only be defined at a single point, and can't be redefined.

***END/LNK in macro***

The only thing that may terminate a macro definition is MEND. If either of the above directives occurs in a macro definition, this error will be given.

***Escape***

This is printed whenever the user presses ESCAPE.

***Expression stack overflow***

This means that an expression in an operand is too complex for MASM to evaluate. It should only occur in expressions where brackets are nested extremely deep.

***File too big***

Source files must be less than 16K. This error means you should split a large source file into two or more smaller ones.

***Heap overflow***

This means MASM has run out of space in which to store string variables. It can only be cured by changing the source program to use shorter strings.

***Local label table overflow***

This is caused by calling macros with local labels very frequently. See the command MLEVEL for a way of curing the problem.

***Mac def in expansion***

This means MASM encountered two MACRO directives without an intervening MEND. Although macro calls may be nested, macro definitions may not.

***Macro nesting too deep***

Macros may only call each other (or themselves) to a level of eight deep. Deeper nesting causes this error.

***Macro parameter table full***

This occurs when the text to be substituted by macro parameters totals more than 250 characters.

***Macro space exhausted***

This means MASM has run out of space to save macro definitions. It is very unlikely that this will happen, but if it does you will have to use fewer or shorter macros.

***No macro being defined***

This is caused when a MEND directive is encountered without a corresponding MACRO.

***No symbols***

This is printed when a SYMBOL command is issued before an ASM or after an ASM in which no symbols were defined.

## *Chapter 12*

### *Not found*

This is given when directory X does not exist on the filing system (ADFS or NFS), so MASM cannot store an object file.

### *Stack fault*

Will only occur if there is a bug in MASM: report it to Acorn.

### *Stack overflow*

This occurs when structures (WHILE..WEND, [...] ) are nested too deeply. It will not occur with sensible use of the structures.

### *Stack underflow*

See 'Stack fault'.

### *Stopped*

This is printed when a ! directive is used and causes assembly to stop.

### *Symbol table overflow*

This occurs when more than 1278 symbols have been encountered in the source.

### *Too many macros*

See 'Macro space exhausted'.

## 12.13 MASM - Problems

The current release of the 65816 assembler is very much an interim product and as such has a number of failings. This Chapter covers the known bugs and limitations associated with MASM and it is strongly recommended that software developers note them. Future releases of the assembler will address these deficiencies.

### 12.13.1 Assignment

{label} = value

Where an undefined value is assigned, the assembler does not flag an error. No bytes are assigned in the output file.

### 12.13.2 Local labels within macros

Under certain conditions, local labels within macros do not work.

Local labels within a macro will fail if their scope is not defined via ROUT.

Local labels will fail if they appear on the same input line as a macro call, thus:

BRA #01

01 FRED

will fail.

but

BRA #01

01 FRED

works OK.

### 12.13.3 Duplicated local labels

MASM does not fault duplicated local labels within a routine. One of the labels will work.

### **12.13.4 Null strings variables in labels**

Consider the case where a string variable is concatenated onto the front of a string to make a label.

Thus:

```
MACRO
return $varname

$varname.yyy RTS

MEND

return xxx
```

is assembled as:

```
xxxyyy RTS
```

If however the string variable \$varname is a null string, then the value is ignored and the ":" concatenator is left on the front of the string making an illegal label thus:

```
return
```

is assembled as:

```
.yyy RTS
```

### **12.13.5 Macro calls within macros**

Nested macro calls result in any locally defined variables being lost.

### 12.13.6 Commented out macro definitions

If you have a commented out macro definition the assembler will fault the MEND as a bad directive.  
eg:

```
[ 1=0  
    MACRO  
    $label TEST $var  
    $label  
    = "Hello $var"  
    MEND  
 ]
```

### 12.13.7 LDYZ

The LDYZ zop mnemonic is sometimes assembled into the wrong opcode.

### 12.13.8 Direct page offset addressing

With the following opcodes when the addressing mode is a calculated direct page offset address, the assembler generates a 24-bit operand and the wrong opcode:

```
STYZ  
STZZ  
BITZ
```

Use macros

```
$STYZ  
$STZZ  
$BITZ
```

instead.

### **12.13.9 Bad direct page with STZZ**

If the mnemonic STZZ is used with a bad direct page address (i.e. > 255) then the wrong opcode is assembled together with a two byte "zero page" address.

### **12.13.10 The GET directive**

If the GET directive is used to include files in what might appear to be reasonable locations within source files, the object files may well end up with the contents of the GET files in them. The use of the GET directive should be restricted to the start of an assembly source chain.

### **12.13.11 The BANG (!) directive**

If the string to be output via the bang directive is not properly delimited with a double-quote ("") then MASM becomes very confused and thinks it is performing its second pass with catastrophic results.

Eg:

```
! 0,"string
```

### **12.13.12 GETting non-existent files**

If line X of a source file attempts to GET a non-existent file then the error message "File not found" is reported for line X of the non-existent file.

### **12.13.13 Code size**

If the size of an object is greater than the source it was generated from it can lead to the error "Code overwriting source". In these situations the problem can be resolved by the insertion of large numbers of null comments in the source file making it of greater size than the resultant object.