Psion Organiser II - Technical Manual

Technical Reference Manual

Manual reproduced from archived web information in good faith and not for profit - no liability held - Martin Reid June 2021 - from org2/techref

INTRODUCTION

The contents of this manual were salvaged from the Archived PSION2.ORG website (http://archive.psion2.org/org2/techref/) created by Dave Woolnough and maintained by Jaap Scherphuis and Boris Cornet (RIP).

Boris Cornet wrote… Behind the features described in the Operating Manuals lies the operating system and hardware. The information in this document was compiled from various sources to give a complete description of how the operating system and hardware work together. It covers all base models - CM, XP, LZ - and their POS variants. The intended audience are machine code programmers and hardware developers. If you are looking for general information, see the republished manuals page first!

Table of Content

Sources and Acknowledgements

PSION Organiser II Technical Manual, Release 1.2 (1986)
Organiser II Model LZ /LZ64 Tchnical Manual, Version 1.0 (1989)
FLASHPAKS (supplement to the Organiser II Technical Manual, 1991)
COMMS LINK Machine Code Interface documentation
Controlling the RS232 port from mchine code programs
PSION Organiser II Training Manual by Gregory Paul
The PSION Link Protocol by Jaap Scherphuis
The LZ Password System by Jaap Scherphuis
Machine Code Tutor by Jaap Scherphuis
LCD Information by Zac Schroff
some posts from org2.com's chat

SYSTEM VERSIONS

SYSTEM VERSIONS

The table below lists all organiser models. Note that the Organiser Mark 1 is included for completeness, but is not covered by this manual.

Most of the base models were also produced as POS (Point Of Sale) versions, either with a full alphanumeric keyboard (POS-250,350,4xx), or a numeric one (POS-100,200,296).

<table>
<tr><th>Model</th><th>Label</th><th>Display</th><th>RAM</th><th>ROM</th><th>POS-Version</th></tr>
<tr><td>Mark 1</td><td>-</td><td>1x16</td><td>2k</td><td>4k</td><td>POS-100</td></tr>
<tr><td>CM</td><td>CM</td><td rowspan="5">2x16</td><td>8K</td><td rowspan="3">32K</td><td></td></tr>
<tr><td>XP</td><td rowspan="3">XP</td><td>16K</td><td rowspan="2">POS-250
POS-200</td></tr>
<tr><td>LA</td><td rowspan="2">32K</td></tr>
<tr><td>multilingual LA</td><td>64K</td><td></td></tr>
<tr><td colspan="2"></td><td>96K</td><td>32K</td><td>POS-350
POS-296</td></tr>
<tr><td>LZ</td><td>LZ</td><td rowspan="2">4x20</td><td>32K</td><td rowspan="2">64K</td><td>POS-432</td></tr>
<tr><td>LP</td><td>LZ64</td><td>64K</td><td>POS-464</td></tr>
</table>

The difference between the normal operating system and the POS variants is in the handling of the top level. The normal operating system enters the top level menu and then dispatches to the various services available, while the POS machines do the following:

1. Boot all devices.
2. Scan all devices for an OPL procedure called BOOT
3. If the procedure is not found then display INSERT PACK and switch off after 1 second.
4. If the procedure is found then run the procedure.
5. Repeat the above steps.

Also, the numeric-only keyboard models switch to numeric mode automatically.

IDENTIFICATION BYTES

In order to allow the machine type to be determined as well as the software revision level of the ROM, there are 4 identification bytes. They occupy the following addresses:

- $FFE7 - Language(s)-byte
- $FFE8 - Model-byte
- $FFE9 - Version number
- $FFCB - Model-byte-2 (introduced with model LZ)

Language-byte

ADDRESS: $FFE7

VALUES: $00 - English
$01 - French
$02 - German
$80 - Eleven languages: English, French, German, Spanish, Italian, Portuguese, Swedish, Danish, Norwegian, Dutch, Turkish (English default)
$81 - English, French, German (English default)
$82 - English, Swedish, Danish (English default)

This was introduced in version 3.6 of the Organiser OS. Earlier versions are english only and this byte usually is $FF.

Bit 7 of this byte set, indicates that the machine is multi-lingual. The value of the remaining seven bits then gives either the language or selection of languages implemented. Further values may have been added as new language variants were introduced.

See also: Foreign Languages

Model-Byte

ADDRESS: $FFE8

VALUES: See below.

The base model type and the special model type are held together in the model byte. The base model type is held in the bottom 3 bits of this byte and the special model type is a single bit set in the remaining 5 bits. The base model type is for identifying the ROM/RAM configuration AND the Operating System.

NOTE: The ROM/RAM configuration alone does not determine the base model type.

<table>
<tr><td>Bits 0-2</td><td>BASE MODEL TYPE
VALUE MODEL RAM ROM
0 CM 8K 32K
1 XP 16K 32K
2 LA 32K 32K
4 POS350 96K 32K
5 LZ64(LP) 64K 64K
6 LZ 32K 64K
7 see MODEL-BYTE-2</td></tr>
<tr><td>Bit 3</td><td>indicates: see MODEL-BYTE-2 for special features</td></tr>
<tr><td>Bits 4-7</td><td>indicate special models
bit 4 set: POS350,464 (??)
bit 7 set: other POS (??)</td></tr>
</table>

Note that these features are totally independent.

Model-byte-2

ADDRESS: $FFCB

VALUES: Bits 0-2 reserved for future base model types
Bit 3,4 reserved for extra features
Bit 5 set in LZ/OS from OS Vers. 4.5 onward
Bit 6 extended LCD
Bit 7 LZ functionality

Version Number

ADDRESS: $FFE9

VALUES: upper nibble: major version number
lower nibble: minor version number

Example: The LZ was initially released at version 4.2 with a lot of bugs but the version number is nevertheless stored correctly as $42 (66 decimal) in $FFE9.

OPERATING SYSTEM VARIANTS

CM/OS

- $FFE8 = $00

The CM/OS is the base operating system from which all other operating system variants are derived. As such any software written to run on the CM/OS is guaranteed to run on all other operating system variants. In other words software is upwardly compatible between the operating systems.

XP/OS

- $FFE8 = $01 - XP machine.
- $FFE8 = $11 - XP POS 250 machine.
- $FFE8 = $41 - XP Alpha POS 200 machine.
- $FFE8 = $81 - XP POS 200 machine.

As for the CM/OS but will support the following extra facilities:

- 128K Datapacks
- Rampacks
- Bar Code Readers
- Swipe Readers

LA/OS

- $FFE8 = $02 - LA machine.
- $FFE8 = $12 - LA POS 250 machine.
- $FFE8 = $42 - LA Alpha POS 200 machine.
- $FFE8 = $82 - LA POS 200 machine.

As for the XP/OS with the exception that the memory allocator is completely different.

This is because the system variables are located at $2000 in the CM/OS and XP/OS as external RAM only starts at this address. However in the LA machines external RAM starts at address $400. In order to keep the operating systems compatible the system variables are still located at address $2000 in the LA machines.

The free RAM is therefor in two sections, between $400 and $2000 and above the system variables. In order to allow the operating system to use the extra 7K of RAM the memory allocator effectively joins these two discontinuous areas of RAM together.

Note that XP and LA were both labelled XP.

LZ/OS

- $FFE8 = $0E LZ/OS - LZ machine.
- $FFE8 = $1E LZ/OS - POS-432 machine.
- $FFE8 = $0D LP/OS - LZ64 (LP) machine.
- $FFE8 = $1D LP/OS - POS-464 machine.

The operating system in the LZ is fully back-compatible with the previous Organiser II operating systems but has many extensions. All system variables are at the same addresses as on the standard Organiser and all system services will work exactly as they did before. Some system services have been extended to use the 4-line screen and there are 51 new services together with new system variables.

FOREIGN LANGUAGES

Foreign languages were introduced in version 3.6 of the Organiser OS. Earlier CM/XP machines were English only. Besides models for a single foreign languages also a multi-lingual XP, offering eleven languages, was produced.

The LZ operating system is multi-lingual and offers English, French and German. All Organiser text, menus and error messages are automatically translated when a new language is selected. The LCD display in the LZ facilitates foreign characters and these can be typed in when the appropriate language is selected - SHIFT-RIGHT-ARROW is used to select foreign characters for the next key press only. For example, when the language is set to French, SHIFT-RIGHT arrow followed by 'a' produces the character 'á'..

The following section applies only to multilingual machines.

LANGUAGE SELECTION ON COLD START

When the machine is switched on from a 'cold start', the language menu is presented (after displaying the copyright message). One language MUST be selected (ON/CLEAR will do nothing) before the top-level menu is displayed in the language chosen.

Devices are also rebooted whenever a language is selected.

These two functions (displaying the language menu and booting the devices) can be disabled to allow bootable software to 'take over' the machine on a cold start. To do this the flags in XTB_COLD described below are used.

CHANGING LANGUAGE

To switch languages under program control, the system service TL$LSET must be used. BTB_LANG should not be poked directly.

SYSTEM VARIABLES

BTB_LANG ($2186)

VALUES: 0 for English, 1 for French, 2 for German, 3 for Spanish, 4 for Italian, 5 for Portuguese, 6 for Swedish, 7 for Danish, 8 for Norwegian, 9 for Dutch, 10 for Turkish

The default value, 0, is the value before any language has been chosen so that devices booted on cold start will boot in English.

BTB_LANG can be be read at any time but should not be written to directly (TL$LSET must be used).

XTB_COLD ($232E)

XTB_COLD stores 2 flags:

- Bit 7 is set to disable the language selection menu.
- Bit 0 is set to disable the 'second boot' after language selection.

These flags can be read or written to at any time. Note that if bit 7 is set, the second boot will still take place unless bit 0 is set. XTB_COLD is reset to zero after language selection is attempted.

SYSTEM SERVICES

TL$LSET

Sets the language. If the language requested is not available, English is selected. The top-level menu is re-initialised so any 'inserted items' will be removed.

It is possible to get an error if there is insufficient room in RAM for the new top-level menu (different languages have different sized top-level menus). If an error occurs, the language is unchanged.

CM/XP/LA RELEASE NOTES

What follows are notes for the different releases of CM and XP machines. This list was published in 1987, further versions have been released later.

Last known release: 3.7

All significant enhancements and known bugs are listed. All the bugs noted are fixed for the following release.

VERSION 2.3 AND EARLIER

Not released in any quantity.

RELEASE 2.4

Released on 12th May 1986.

ENHANCEMENTS:

- Devices booted when the Organiser is cold booted.

BUGS:

- Machine may crash a on a cold boot if the install code of a device alters the menu.
- Rebooting a machine with more than one device installed may cause a crash.
- An attempt to translate the illegal statement "11SIN" results in a crash.
- If a field is assigned to that is beyond those already assigned to before it may write the data over the diary or operating system variables. This can, in rare cases, result in a crash. The fix is to assign (a null string or zero) to the last field after creating or opening a file and after getting an END OF FILE condition. (The END OF FILE condition is when EOF is non-zero, this means that all values of the fields have been zeroed out.)
- If the diary is backed over midnight and then brought forward, using the arrow keys, any diary entries seem to have disappeared.
- If RAM is full, editing a procedure can give OUT OF MEMORY and leave the screen corrupted.
- If an error is encountered when adding to or subtracting from a calculator variable (M0 to M9) then the variable is corrupted.
- Floating point AND produces unreliable results.
- VIEW(1,"") does not work properly.
- VIEW of a 255 character string doesn't scroll.
- Strings longer than 255 can be declared (e.g. 256 gives a zero length string).

- Declaring arrays which overflow memory size can cause machine to crash at run-time. e.g. LOCAL s$(255,100) or LOCAL a(10000).
- INPUT A.A$ artificially limits the length input to 252 minus the record size (as found from RECSIZE). It should allow 254 characters.
- If the machine is turned off in the minute before an alarm is due it won't go off for 34 minutes.
- If packs are accessed at the same time as the buzzer is used (e.g. from a key click) the bottom byte of the pack can, very rarely, be blown to zero.

RELEASE 2.6

Released on 15 October 1986.

ENHANCEMENTS:

- Checksum on RAM pack header.
- Battery checking improved.
- Can call OPL programs on device D.
- INFO calculates FREE memory as percentage free of the total memory less the operating system.
- OPL string comparison become case dependent.
- UDGs preserved when the Organiser is turned off.
- Intelligent, faster NEXT - works on one pack at a time.
- A POSITION to an illegal place, e.g. past the end of file, used to leave the position alone. It now it positions to the last or first record as appropriate.

BUGS:

- If after a successful CLOSE another CLOSE is done when no files are open it can crash the system.
- It is possible but unlikely to get spurious alarms when loading devices.
- If the ON/CLEAR is pressed at the same moment as an alarm it may be missed.
- When a translator error is detected some memory may be lost until the language is run.
- Using MENU with an item with more than 17 letters causes an infinite loop.
- Deleting a null diary entry causes the next diary entry to be deleted.
- TRAP DELETE "A:"+XXX$: can cause a crash if there is an error in XXX$:.
- Using logical name D can cause OPEN to fail with error number zero. As an error of zero is not reported at the top level, it is safest not use logical name D.
- POS200 ONLY. The menu cell is grown every time the machine is booted so memory eventually fills up. Solution is to cold boot the machine.

INTRODUCED BUGS:

- Sizing 8, 16 & 32K packs can fail even when the pack is good.
- Erasing records on RAMPAKS followed by APPEND may corrupt the PACK. Can be solved by avoiding using UPDATE and doing FIRST after using ERASE.
- Copying MK1 packs fails with PACK CHANGED error.

RELEASE 3.1

Released on 30 January 1987

ENHANCEMENTS:

- If a file is OPENned/CREATEd/DELETEd without a specified device it takes the last device used (as opposed to a random device).
- Reports CHR$(256) and negative arguments to LEFT$, RIGHT$ and MID$ as errors.
- Intelligent NEXT improved works with 1 file on each device.
- LA split device loader.

- 128K RAM packs supported.
- Faster deleting on RAM packs.

LZ RELEASE NOTES

What follows are notes for the different releases of LZ machines. This list was published in 1989, further versions have been released later.

Last known version: 4.6

All significant enhancements and known bugs are listed. All the bugs noted are fixed for the following release.

RELEASE 4.2

Released on: 16th March 1989

- Memory 'grabbed' by devices (e.g. capture buffer in CommsLink) is not shown in INFO.
- If LOW BATTERY is detected during a cold start, garbage ALARMS are set.
- SORTING a file which contains other records interspersed within its own will cause 'extra' records to be added at the front of the file.
- SORT does not order records containing the same character repeated different numbers of times correctly, e.g. AA and A.
- If there is an error in when saving a DIARY (e.g. PACK FULL) during TIDY, the DIARY is still be deleted.
- If a match string > 10 characters is passed to any of the WILD CARD OPL FUNCTIONS (e.g. DIRW) the machine will crash.
- OPL VIEW of a null string will crash the machine. Also DISP(0,"string") will crash.
- If part of a password is typed on entry to a notepad and then left to time out - it beeps and clears the password, then prompts for the password again.
- If memory is full and a notepad is attempted to be entered, OUT OF MEMORY is displayed the first time but the 2nd time a new notepad is created.
- "Load..." is displayed without clearing the screen on entry to the notepad.
- In the DIARY "nn entries shortened" says 10 when there is 100 etc.
- In WORLD "The Hague" is shown as "The Hage"
- Alarms go off accurate to within 1 minute only. They should be exact.
- If part of an OPL procedure or NOTEPAD is received with COMMS-LINK and the connection is broken, the machine may crash.
- A bootable pack created on the DEVELOPER runs OK in 2-line mode but if CommsLink is present as well, it runs in 4-line mode.
- There is a chance of getting a garbage ALARM going off during RESET.
- There are some errors in the GERMAN text.
- Calling OPL EDIT with an 80 character prompt, e.g. AT 20,4 :PRINT "X": :EDIT A$ does not work correctly.

RELEASE 4.3

Released on: 24th April 1989

- If a "workday" alarm is set for the Friday before the end of a month whose last day falls on a Saturday or Sunday, some corruption may occur when the alarm goes off or if you attempt to set an alarm on the Saturday or Sunday following the Friday.
  On version V4.2 you will get a DEVICE MISSING error but on V4.3 it will appear to be OK. In both versions there may be corruption and there may be a possibility of the machine TRAP'ing.
  The dates it will occur on are as follows:

2021:	2021-01-29,	2021-02-26,	2021-07-30,	2021-10-29,
2022:	2022-04-29,	2022-07-29,		2022-12-30,
2023:	2023-04-28,	2023-09-29,		2023-12-29,
2024:	2024-03-29,	2024-06-28,	2024-08-30,	2024-11-29,
2025:	2025-05-30,	2025-08-29,		2025-11-28,

  etc.

RELEASE 4.4

Released on: 8th May 1989

OPERATING SYSTEM

SYSTEM INTERFACE

CALLING SYSTEM SERVICES

The interface to the operating system is via the SWI hardware instruction followed by the vector number of the operating system service required, i.e.

```
       SWI                ; software interrupt
       .BYTE VECTOR_NUMBER ; required service vector
HERE:
```

After execution of the SWI call, execution continues after the byte containing the vector number, at the label HERE in the example above.

Parameters to the service are passed in one or more of the A,B (or D), and X registers and for a small number of services in memory locations UTW_S0 and UTW_S1 which are zero page locations. Certain services also require information in the runtime buffer RTT_BF. The description of each service details the information required to be passed to it. In addition the results of a system service are returned in the machine registers A,B (or D) or X as required.

REGISTER PRESERVATION

In general it should be assumed that all registers (A,B and X) are trashed by system services. If this is not the case then the system service description will explicitly state which registers are preserved.

ERROR HANDLING

All system services which indicate that they have error returns may set the carry flag on exit from the system service and return the error code in the B register. Note that system services such as KB$GETK, which do not return any errors, may not clear the carry flag.

THE OS MACRO

Throughout the operating system description examples will be provided which use a macro called OS which is as follows:

```
    .MACRO      OS XX
    .BYTE       $3f,XX
    .ENDM
```

Using this macro to get a key, for example, will be coded as follows:

```
                OS KB$GETK
```

and calling a system service which can return an error such as AL$GRAB should be called as follows:

```
                CLR A
                CLR B
                OS  AL$GRAB
                BCC 1$
                ; HANDLE THE ERROR WHOSE CODE IS IN THE B REGISTER.
 1$:            ; CALL SUCCESSFUL AND X HAS THE TAG OF THE CELL.
```


MEMORY STORAGE

In the following description refer to chapter Memory Usage for the addresses of the variables described.

The six words of memory storage labeled UTW_S0 to UTW_S5 are a set of scratch variables used by the operating system which any service may trash as required. Thus no values can be held in these words while making a call to an operating system service, although they may be used for storing intermediate values between calls to the operating system.

The seven words of memory storage labeled UTW_R0 to UTW_R6 are a set of fixed variables which are not trashed by the operating system service routines. The service routines actually use these variables but always push their contents on the stack before use and then recover them by popping them off the stack again. Application programs may use these variables as long as they maintain their integrity by pushing and popping.

As a code saving device there is a system service to push and pop these variables as follows:

```
      OS     BT$PPRG     ; PSH UTW_R0
      .BYTE  1           ; INSTRUCTION BYTE TO BT$PPRG
      ; CAN NOW USE UTW_R0
      OS     BT$PPRG     ; POP    UTW_R0
     .BYT    $81         ; INSTRUCTION BYTE TO BT$PPRG
```

The byte following the call to BT$PPRG instructs the service whether to push or pop the variables from the stack and which variables to push or pop. The format of the byte is as follows:

- BIT 7 - If set then pop the variables else push the variables
- BIT 6 - If set then push or pop UTW_R6
- BIT 5 - If set then push or pop UTW_R5
- BIT 4 - If set then push or pop UTW_R4
- BIT 3 - If set then push or pop UTW_R3
- BIT 2 - If set then push or pop UTW_R2
- BIT 1 - If set then push or pop UTW_R1
- BIT 0 - If set then push or pop UTW_R0

Thus if the byte value is $5 then UTW_R2 and UTW_R0 will be pushed.

When pushing, the higher address variables are pushed first and when popping, the lower address variables are popped first. Thus if UTW_R5 and UTW_R2 are pushed and UTW_R2 and UTW_R1 are popped then UTW_R1 will get the old value of UTW_R2 and UTW_R2 will get the old value of UTW_R5.

SYSTEM CONSTANTS AND MACROS

The files system.inc (CM/XP) and syslz.inc (LZ) contain a set of useful constants and macros used in the examples.

ERROR NUMBERS

Included in the file errors.inc are all the symbolic names of the error numbers returned by the operating system services.

These error numbers are always returned in the B register after an operating system service has signaled an error by returning with the carry flag set.

VECTOR NUMBERS

Included in the file swi.inc are all the operating system service names and numbers. The system services are briefly explained in this manual. Please refer to the System Services page for an in depth discussion.

Note that vectors 0 to 125 are available on all machines, 126 is available from OS version 2.5 on and 127 from OS version 2.7 on. Vector 128 is available on multi-lingual machines only.

Vectors 129 to 179 are only available on LZ machines.

A call to an unavailable vector may cause a system crash, therefor it is advisable to check the system version before calling one of these services.

POWER UP

The machine can be switched on in 3 different ways.

1. By pressing the ON/CLEAR key.
2. By the clock counter expiring in the semi-custom chip.
3. By asserting the ON line from the top slot.

On power up the program counter is loaded from the system restart vector at address $FFFE in the operating system ROM. The system restart vector is set to address $8000 in all versions of the operating system, which is the base address of the operating system ROM. The restart code is then executed in the following sequence.

1. Zero page RAM is enabled by setting the RAME bit in PORT 5 of the processor.
2. The machine stack is initialized temporarily to the top of zero page RAM, i.e. at address $FF.
3. The LCD display is cleared.
4. The control lines to the datapack bus are initialized to a known state.

The next task performed by the operating system is to determine whether a warm or cold start is required. Essentially a cold start is when the machine is starting up for the first time after a power failure and as such all operating system variables must be initialized. A warm start is when the machine only needs to carry on from when it was powered down, as all RAM values are still valid.

The Hitachi HD6303X microprocessor has an internal flag to determine whether power to the internal RAM in the processor has failed at any stage. This flag is the top bit in PORT 5. If the flag is clear then power to the internal RAM has failed at some stage and so a cold start is required. If the flag is still set then the internal RAM is still intact and so a warm start is required. As the external RAM is on the same power rail as the processor the above flag also serves to describe the validity of the external RAM.

COLD START

On executing a cold start the machine performs the following procedures:

1. Perform a simple RAM test on the external RAM.
2. Determine the last address in the external RAM.
3. Check that this address is valid for the three memory models available, i.e. $3FFF, $5FFF or $7FFF.
4. If the value does not correspond to one of the valid values then the RAM must be faulty, so the buzzer is sounded and the machine powered down.
5. Use the last address of valid RAM plus 1 to initialize BTA_RTOP.
6. Set the machine stack address to BTA_RTOP.
7. Subtract 256 from BTA_RTOP and use the value to initialize RTA_SP and BTA_SBAS. This leaves 256 bytes for the machine stack.
8. Initialize all soft vectors and all operating system variables.
9. Enable NMI interrupts to the processor.
10. Start the timer to provide keyboard scan interrupts.
11. Test that the battery voltage is over 5 volts. If it is not then display low battery message for 4 seconds and then switch off the machine.
12. Boot any devices.
13. Show copyright message.
14. Start at the top level menu.

Multi-lingual machines present the language menu after displaying the copyright message. One language must be selected (ON/CLEAR will do nothing) before the top-level menu is displayed in the language chosen. Devices are rebooted whenever a language is selected.

WARM START

On deciding that a warm start is required the operating system performs the following code:

```
LDS     BTA_SAVSTACK
LDX     BTA_WRM
JMP     0,X
```

The power off service BT$SWOF stores the stack pointer in BTA_SAVSTACK before switching off the machine to enable the machine to restart at the same place it was at when the machine was switched off. With the stack restored to the value that it had in BT$SWOF, an "RTS" will continue execution after the call to BT$SWOF. Next the vector for warm starting is used to jump to a warm start routine. The warm start routine may be replaced by another routine but the operating system warm start routine should be called as well. If it is not the operating system will not work correctly. See section Vectors on replacing vectored routines.

The system warm start routine performs the following actions:

1. Switch non-maskable interrupts (NMI) on to the processor.
   This is done by testing the ADDRESS SCA_NMIMPU
2. Wait for the first NMI to occur.
   This is necessary as on each switch-on the clock counter in the semi custom chip gains a second and so the first NMI must be ignored.
3. Test the hardware flag ACOUT in POB_PORT5.
   If the flag is true then the counter has expired and the time needs updating as follows:
   1. Update the system clock by the length of time the machine has been switched off for.
   2. Determine whether an alarm is due currently and if so carry on with the power-on sequence, if not then just switch off the machine again.

   Otherwise the machine was switched on with the ON/CLEAR key or by an external interface and the system clock is updated by the time the machine has been switched off for.
4. Restore the display to its state before power down.
5. Set the ALARM_TO_DO flag so that any alarms will be serviced.
6. Reset the AUTO_SWITCH_OFF countdown and check that it's greater than 15 seconds. If it is not then make it 15 seconds.
7. Check for low battery as in cold starting.
8. Restore the interrupt status and the value in TIMER 1 STATUS CONTROL REGISTER (TSCR1) which were saved by the BT$SWOF service.
9. Wait for the ON/CLEAR key to be released.
10. Restore the USER-DEFINED GRAPHICS as saved by the BT$SWOF service.
11. Flush the keyboard buffer and restart keyboard scan interrupts.
12. Execute a return instruction which resumes execution after the call to the BT$SWOF service.

POWER DOWN

The machine can be switched off by calling the POWER DOWN service routine BT$SWOF. This routine is vectored through BTA_SOF as follows:

```
STS     BTA_SAVSTACK
LDX     BTA_SOF
JMP     0,X
```

The power down routine may be replaced by another routine but the operating system power down routine should be called as well. If it is not the operating system will not work correctly. See section Vectors on replacing vectored routines.

The system power down service performs the following steps:

1. Save the current value of the status flag, to preserve the interrupt status.
2. Save the value in the TIMER 1 STATUS CONTROL REGISTER (TSCR1).
3. Disable interrupts.
4. Clear the LCD display.
5. Save the current values of the USER DEFINED GRAPHICS in the LCD.
6. Switch off the interface slots.
7. Reset the COUNTER to 0.
8. Check whether any alarms are due to run in the next 34 minutes and 8 seconds. If so, adjust the counter so that the machine is woken up in time to service the alarm. See section Keeping Time for more information.
9. Switch non-maskable interrupts to the COUNTER.
10. Disable the internal RAM by clearing the RAME bit in POB_RCR.
11. Test the address SCA_SWITCHOFF to put the processor in standby mode.

The LZ may be powered down for a specified time, if so, step 8 is performed as if an alarm was due at that time.

INTERRUPTS

The operating system uses four of the ten hardware interrupts to perform various services. The remainder are just directed to a RTI instruction. However, all interrupts are vectored through RAM so that they may be intercepted or replaced in total. For the interrupts used by the operating system it is recommended that the interrupt only be intercepted and not be replaced if it is required that the operating system performs according to specification.

NON-MASKABLE INTERRUPTS

The semi-custom chip will deliver an NMI to the processor exactly once every second to provide an accurate system clock, provided the address SCA_NMIMPU has been accessed.

The NMIs to the processor can be disabled by accessing the address SCA_NMICOUNTER. Note that this is an unusual feature since NMIs, as their name implies, cannot normally be disabled.

The NMI service routine performs the following actions:

1. Clear the flag BTB_NMFL.
   This is to allow code to detect that an NMI has occurred as follows:

```
2.              INC BTB_NMFL
3.          1$: SLP          ; Go to sleep until an interrupt
4.              TST BTB_NMFL ; Test NMI flag
5.              BNE 1$       ; Still set - so wrong interrupt
```

```
        ; NMI interrupt occurred.
```

6. Test the BTB_IGNM (ignore NMI flag, set on warm starting)If it is zero then set the flag and return from the interrupt.
7. Update the system time by one second.
8. Check whether the seconds count is exactly zero, i.e. a whole minute.
   If it is then check the alarms enabled flag AMB_EI. If it is set then set AMB_DOIT which will trigger a scan of the alarms pending lists at the next TIMER 1 interrupt. Thus alarms are scanned only every minute and are under control of the processor interrupt flag.
9. Check the auto switch off flag TMB_SWOF.
   If it's set then decrement the time out count in TMW_TOUT if it is not already zero.
10. Return from the interrupt.

TIMER 1 COMPARE INTERRUPT

The timer 1 compare interrupt is used to scan the keyboard to allow keyboard buffering and to provide a timing service. This interrupt is referred to in the documentation as the KEYBOARD INTERRUPT (KI).

The interrupt is generated every time the free running counter matches the count in the timer 1 compare register. As the free running counter is being clocked by the system clock of 921600 HZ, extremely accurate timing can be performed using this interrupt.

The time between interrupts is controlled by the variable KBW_TDEL which is initialized on cold start to be $B3DD. This value makes the KI interrupt occur exactly every 50 milliseconds and is used extensively by the operating system for timing purposes. The value in KBW_TDEL can be changed, but all system timing will be destroyed as a result and the operating system may fail to perform correctly.

The KI service routine performs the following steps:

1. Reset the free running counter to zero.
2. Set the timer 1 output compare register to the value in KBW_TDEL.
3. Check the alarm service required flag AMB_DOIT. If it is set then run the alarm service and clear the flag AMB_DOIT.
4. Increment the frame counter TMW_FRAM.
5. Decrement the word DPW_REDY if it's not already zero. This flag is used to perform system timing for the operating system. If a delay of 150 milliseconds were required then the following code could be used to achieve that delay:

```
6.                      LDX     #3        ; Three KI interrupts at 50 ms each
7.                      STX     DPW_REDY:
8.                  1$: SLP               ; Wait for interrupt
9.                      LDD     DPW_REDY: ; See if decremented to zero yet
```

```
                    BNE     1$        ; 150 ms now elapsed.
```

10. Poll the keyboard. See section Keyboard Polling for more information.

SOFTWARE INTERRUPT

This interrupt is used to provide the interface between applications and the operating system. When a SWI instruction is executed the following code is executed:

```
        LDX     BTA_SWI
        JMP     0,X
```

The SWI service routine provided by the operating system is as follows:

```
        PUL  A
        STA  A,BTB_4DONTUSE          ;SAVE STATUS FLAG
        PUL  B
        PUL  A
        STD  BTW_1DONTUSE            ;SAVE THE D REGISTER
        PULX
        STX  BTW_2DONTUSE            ;SAVE THE X REGISTER
        PULX                         ;GET THE ADDRESS OF THE BYTE
        LDA  B,0,X                   ;FOLLOWING THE SWI INSTRUCTION
        INX                          ;INCREMENT TO SKIP OVER THE BYTE
        PSHX                         ;PUSH BACK AS RETURN ADDRESS
        LDX  BTA_VECT                ;GET THE VECTOR TABLE ADDRESS
        ABX                          ;DOUBLE THE VECTOR NUMBER AND
        ABX                          ;ADD TO VECTOR TABLE
        LDX  0,X                     ;GET ADDRESS OF ROUTINE
        PSHX                         ;SAVE ON STACK FOR DUMMY RETURN
        LDX  BTW_2DONTUSE            ;RESTORE X REGISTER
        LDD  BTW_1DONTUSE            ;RESTORE D REGISTER
        PSH  A                       ;SAVE A REGISTER
        LDA  A,BTB_4DONTUSE          ;GET STATUS FLAGS
        TAP                          ;RESTORE STATUS FLAGS
        PUL  A                       ;RESTORE A REGISTER
        RTS                          ;JUMP TO REQUIRED ROUTINE
```

TRAP INTERRUPT

The operating system routine to handle the trap interrupt simply clears the LCD display and displays "TRAP" on the screen until the battery is removed and the machine is COLD started again. It is intended that this trap remain free for use with a debugger and as such should not be used.

VECTORS

In order to provide flexibility in the operating system all the hardware interrupts are vectored through RAM vectors so that they may be intercepted or even replaced. In addition a number of operating system services are provided through vectors as well so that they may be intercepted or replaced more easily than intercepting the SWI interrupt.

If the operating system is using an interrupt, it is strongly recommended that the interrupt is just intercepted and not replaced entirely. This can be done in the following way for example to intercept the NMI interrupt routine.

Initialization code:

```
        LDX     BTA_NMI         ; CURRENT SERVICE ROUTINE ADDRESS
        STX     OLD_NMI         ; SAVE THE ADDRESS SOMEWHERE
        LDX     #NEW_NMI        ; NEW NMI ROUTINE ADDRESS
        STX     BTA_NMI         ; RE-DIRECT THE OPERATING SYSTEM
```


New NMI routine:

```
    NEW_NMI:
        ; PERFORM NEW NMI USER CODE
        LDX     OLD_NMI         ; OLD NMI ROUTINE ADDRESS
        JMP     0,X             ; RUN THE OLD ROUTINE
```

HARDWARE VECTORS

The following is a list of hardware interrupts and their vectors in RAM. Those preceded by a * are not used by the operating system.

1	* IRQ2	BTA_2IQ	Interrupt request 2
2	* CMI	BTA_CMI	Timer 2 counter match interrupt
3	TRAP	BTA_TRP	Trap exception interrupt
4	* SIO	BTA_SIO	Serial input/output interrupt
5	* TOI	BTA_TOI	Timer 1 overflow interrupt
6	OCI	BTA_OCI	Timer output compare interrupt
7	* ICI	BTA_ICI	Timer 1 input capture interrupt
8	* IRQ1	BTA_1IQ	Interrupt request 1 interrupt
9	SWI	BTA_SWI	Software interrupt
10	NMI	BTA_NMI	Non-maskable interrupt

SOFTWARE VECTORS

1	WARM	BTA_WRM	Warm start vector
2	SWOF	BTA_SOF	Power down vector (switch off)
3	POLL	BTA_POLL	Keyboard poll routine vector
4	TRAN	BTA_TRAN	Key translate routine vector

SYSTEM SERVICES

BT$NMDN

This system service routine switches off NMIs to the processor.

NOTE: As the NMI interrupt is used to provide the system clock the system time will be invalid. It is possible to switch off NMIs and still maintain a valid system time. See the system service BT$NOF.

If NMIs are switched off with this service then they should be switched back on with the BT$NMEN service.

BT$NMEN

This system service routine switches on NMIs to the processor. BT$NMEN should only be used to reestablish NMIs if they have been disabled with the BT$NMDN service.

BT$NOF

This system service routine switches off NMIs to the processor in such a way that the system time will be preserved.

The NMI interrupt is generated from the semi-custom chip every second to provide a very accurate system clock. When NMIs are switched off the processor, an internal counter in the semi-custom chip is connected to the NMI line so that NMIs can still be counted.

This service ensures that the counter in the semi-custom chip is reset properly so that on switching NMIs back to the processor the time can be updated properly. In order to do this it is imperative that the BT$NON service is used to switch on NMIs. The maximum time that can be stored in the counter is 2048 seconds, so BT$NON must be called before this time has elapsed.

There is a disadvantage to this pair of services, in that on restoring NMIs to the processor the BT$NON service must wait for the first NMI before counting the number of NMIs that have occurred while NMIs were switched off. This means that depending on when in the cycle the BT$NON service is called a delay of between 0 and 1 second can occur. See chapter System Timing for more information on the timing services.

Finally note that the counter used to count NMIs when they are disabled from the processor is also used to scan the keyboard and as such any keyboard interrupt occurring after BT$NOF has been called will destroy the NMI counter. Thus to disable NMIs and preserve the system time, interrupts should also be disabled before calling BT$NOF. Keyboard services must not be called, as they will poll the keyboard directly when interrupts are disabled.

EXAMPLE

```
        TPA                     ; GET CURRENT STATUS
```

```
        PSH     A               ; SAVE IT
        SEI                     ; DISABLE INTERRUPTS
        OS      BT$NOF          ; SWITCH NMIS OFF.
        ; CODE PERFORMED WITH NO INTERRUPTS OCCURRING.
        OS      BT$NON          ; SWITCH NMIS BACK ON AND UPDATE TIME.
        PUL     A               ; RESTORE PREVIOUS STATUS
        TAP                     ; SET STATUS REGISTER
```

BT$NON

This system service routine switches on NMIs to the processor. BT$NON should only be used to reestablish NMIs if they have been disabled with the BT$NOF service.

BT$PPRG

This system service routine will push or pop the seven variables UTW_R0 to UTW_R6 on the machine stack.

BT$SWOF

This system service may be called to switch off the Organiser II.

BT$TOFF (LZ only)

A new facility on the LZ is the ability to switch off for a specified time before switching back on automatically. This can be done from OPL with the "OFF x%" command and from machine code using BT$TOFF. The maximum time to be off is 1800 seconds (30 minutes) since the system clock must be updated. The routine can, of course, be called repeatedly to 'apparently' switch off for longer times.

EXAMPLE: Switch the machine off for 12 hours.

```
   LDA A,#24     ;24 1/2 hours needed
   PSH A
   LDD #1800     ;to switch off for 30 minutes
   OS BT$TOFF
   PUL A
   DEC A
   BNE 1$        ;repeat 24 times
   RTS
```

Note: If an ALARM is due before the time to switch back on, the machine will switch on early to service the alarm.

MEMORY USAGE

INTRODUCTION

This chapter covers all aspects of memory usage, RAM, ROM, registers and I/0. To use an Organiser effectively it is essential to understand the way memory is used.

The six scratch registers UTW_S0 to UTW_S5 may be used for local values. They are used by the operating system and are not guaranteed to be preserved by operating system calls. The seven scratch register UTW_R0 to UTW_R6 are maintained values. If they are used, the existing values must be pushed and restored before returning from that routine.

MEMORY MAP

Model CM

```
$FFFF   I------------------------------I
        I          System ROM          I
$8000   I------------------------------I
        I                              I
        I           Not Used           I
        I                              I
$4000   I------------------------------I
        I       Processor Stack        I
$3F00   I------------------------------I
        I        Language Stack        I
        I         (grows down)         I
        I                              I
        I          (grows up)          I
        I       Allocated Cells        I
        I..............................I
        I       System Variables       I
$2000   I------------------------------I
        I           Not Used           I
$0400   I------------------------------I
        I       Hardware Addresses     I
$0100   I------------------------------I
        I  Transient Application Area  I
$00E0   I------------------------------I
        I        System Variables      I
$0040   I------------------------------I
        I           Not used           I
$0020   I------------------------------I
        I       Internal Registers     I
$0000   I------------------------------I
```

Model XP

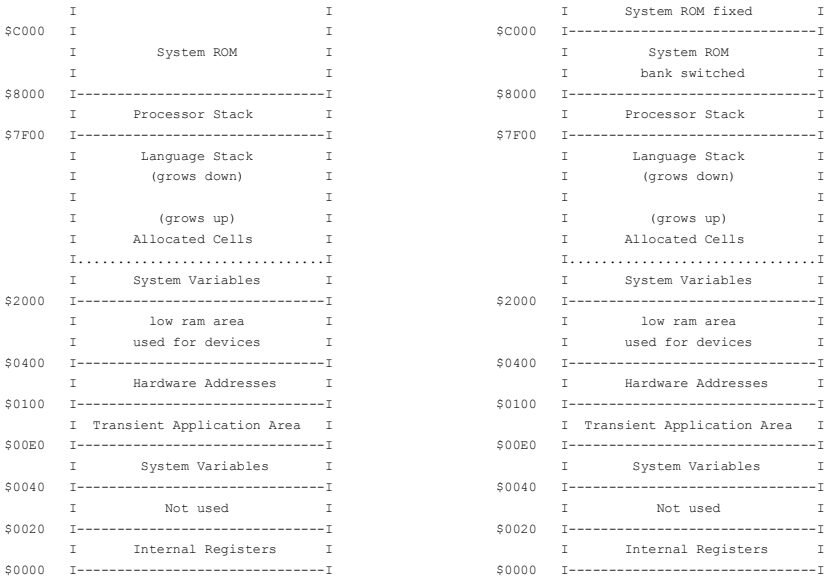
```
$FFFF   I------------------------------I
        I          System ROM          I
$8000   I------------------------------I
        I           Not Used           I
$6000   I------------------------------I
        I       Processor Stack        I
$5F00   I------------------------------I
        I        Language Stack        I
        I         (grows down)         I
        I                              I
        I                              I
        I                              I
        I          (grows up)          I
        I       Allocated Cells        I
        I..............................I
        I       System Variables       I
$2000   I------------------------------I
        I           Not Used           I
$0400   I------------------------------I
        I       Hardware Addresses     I
$0100   I------------------------------I
        I  Transient Application Area  I
$00E0   I------------------------------I
        I        System Variables      I
$0040   I------------------------------I
        I           Not used           I
$0020   I------------------------------I
        I       Internal Registers     I
$0000   I------------------------------I
```

Model LA

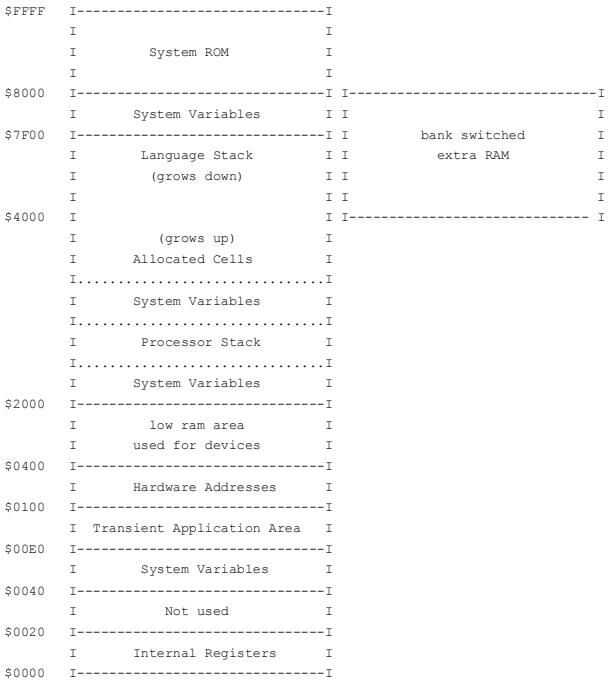
```
$FFFF   I------------------------------I
```

Model LA (multilingual)

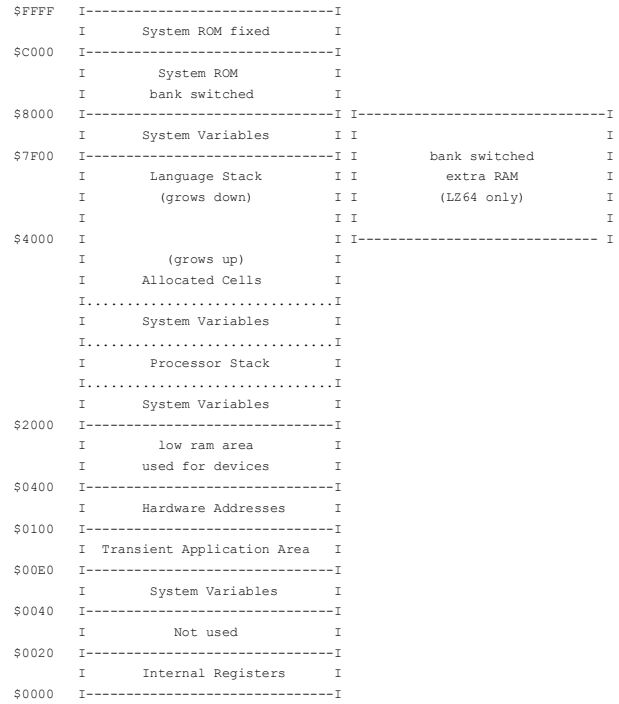
```
$FFFF   I------------------------------I
```



Model POS-350



Models LZ and LZ 64



RAM MEMORY

ZERO PAGE

Zero page RAM runs from $40 to $FF. From $40 to $DF is used by the operating system; from $E0 to $FF is the transient application area. It is safe to assume that these variables get trashed by all device drivers. In particular it should be noted that a WARM BOOT uses $F8 to $FF.

For example the RS232 uses $E0 to $FF for the setup parameters so that the code is faster and more compact. Every time COMM is re-invoked these parameters are copied down from the dedicated application space (at DV_SPAR - $214F).

NON-ZERO PAGE

On the different models RAM is at the following addresses:

The top $100 bytes of RAM is reserved for the processor stack in models CM, XP and LA. In models POS-350, LZ and LZ64 the processor stack is situated inmidst the system variables (BTT_MSTACK). Models LZ and LZ64 use the top $100 bytes for additional system variables.

The RAM in an POS-350 and an LZ64 is arranged in 3 (LZ64) or 5 (POS-350) "banks" of 16K which are switched as required. On the LZ64 and the POS-350, the 16K RAM between addresses $4000 and $8000 is used in the same way as on the CM/XP/LZ models, for example to hold the current diary (and on the LZ, the current notepad), except that "PACK A:" starts in the highest RAM bank and, as it grows, works its way through the other banks and then into the main bank which the other functions (diary etc.) use. PEEKing and POKEing between addresses $4000 and $8000 on an LZ64 or an POS-350 will only access the bank which is currently selected, and is therefore not recommended.

At $2000 the non-zero page variables start. Immediately thereafter is allocated space. This consists of 16 pre-allocated cells for use by the operating system and 16 cells which can be used by applications for permanent use.

The pre-allocated areas are (in order):

Permanent cell	Code/data from booted devices
Top level menu cell	
Diary cell	
Language text cell	Editing/translating OPL/CALC
Symbol table cell	Translating OPL/CALC
Global record cell	Translating OPL/CALC

QCODE output cell	Translating/running OPL/CALC
4 field name tables	Field names for files opened in OPL
4 file buffers	Buffers for files opened in OPL
Database cell	All data held in device A:

Device A: has the same format as a datapack and contains both OPL procedures and databases. There are just a few differences between device A: and devices B: & C:

1. The pack header of device A: is undefined.
2. When an item is deleted the space is recovered (as in a rampack).
3. Uses PKW_CMAD instead of PKW_CPAK.
4. The pack size is variable.
5. Accessing is faster.
6. The data is less secure, erased by resetting the machine.

The allocated area grows up towards the language stack. Before any operation involving memory both the allocator and the language ensure there is at least $100 bytes between RTA_SP and ALA_FREE. If there isn't 'OUT OF MEMORY' or 'PACK FULL' is reported. These messages are functionally the same in the case of device A:, it depends on which routine the out of memory condition is detected.

NON-RAM MEMORY MAP

In the memory map certain areas contain system registers and I/O rather than RAM.

INTERNAL REGISTERS

The addresses in the range $01 to $1F are used as internal registers by the 6303 processor. Their use is as follows:

Address	Organiser Name	Register	Read/Write
$01	POB_DDR2	Port 2 data direction register	W
$02	Not used	Port 1	R/W
$03	POB_PORT2	Port 2	R/W
$04	Not used	Port 3 data direction register	W
$05	Not used	Not used	-
$06	Not used	Port 3	R/W
$07	Not used	Port 4	R/W
$08	POB_TCSR1	Timer Control/Status	R/W
$09	POW_FRC	Free running counter - high	R/W
$0A	Not used	Free running counter - low	R/W
$0B	POW_OCR1	Output compare register - high	R/W
$0C	Not used	Output compare register - low	R/W
$0D	Not used	Input capture register - high	R
$0E	Not used	Input capture register - low	R
$0F	POB_TCSR2	Timer control/Status register 2	R/W
$10	POB_RMCR	Rate, mode control register	R/W
$11	POB_TRCSR	Tx/Rx control status register	R/W
$12	POB_RDR	Receive data register	R
$13	POB_TDR	Transmit data register	W
$14	POB_RCR	RAM/Port 5 control register	R/W
$15	POB_PORT5	Port 5	R
$16	POB_DDR6	Port 6 data direction register	W
$17	POB_PORT6	Port 6	R/W
$18	Not used	Port 7	R/W
$19	Not used	Output compare register - high	R/W
$1A	Not used	Output compare register - low	R/W
$1B	POB_TCSR3	Timer control/Status register 2	R/W
$1C	POB_TCONR	Timer constant register	W
$1D	Not used	Timer 2 Up counter	R/W
$1E	Not used	Not used	-
$1F	Not used	Test register (do not use)	-

See System Board for a full description of the Organiser use of the timers and ports and the Hitachi 6303X User Guide for a more detailed general description of the ports.

MEMORY MAPPED I/O

In the memory map between $180 and $3FF lie the semi-custom chip addresses:

Address	Organiser Name	Description
$0180	SCA_LCDCONTROL	Liquid Crystal Display (LCD) registers
$0181	SCA_LCDDATA	
$01C0	SCA_SWITCHOFF	Switch off
$0200	SCA_PULSEENABLE	Pulse enable
$0240	SCA_PULSEDISABLE	Pulse disable
$0280	SCA_ALARMHIGH	Buzzer
$02C0	SCA_ALARMLOW	
$0300	SCA_COUNTERRESET	Counter for kybd + clock
$0340	SCA_COUNTERCLOCK	
$0380	SCA_NMIMPU	Enable NMI to processor
$03C0	SCA_NMICOUNTER	Enable NMI to counter

The LZ, P350 and multi-lingual XP need more control lines for their memory management, and for this the following semi-custom addresses are used:

$0360	SCA_BANKRESET	Reset ROM and RAM to first banks (LZ/P350/M-XP)
$03A0	SCA_NEXTRAM	Select next RAM bank (LZ/P350)
$03E0	SCA_NEXTROM	Select next ROM bank (LZ/M-XP).

The semi-custom chip does not decode all the address lines. For the LCD driver $180,$182,$184 up to $1BE are identical, as are $181,$183, up to $1BF. For the other addresses the bottom 6 bits are ignored (except that LZ only ignores only bottom 5 bits).

These addresses should be accessed with great care. To prevent casual accessing, Peeking and POKEing from OPL to the range $0182 to $03FF and in the range $00 to $3F are disallowed.

MEMORY USAGE

OPERATING SYSTEM

NAMING CONVENTION

All variable names have the following format:

```
XXY_ABCD
```

where XX is the abbreviated name of the module which uses the variable, Y is either b for byte variables, or w for word variables, or t for tables, or a for address, and ABCD is the name of the variable of which only the first 3 letters are significant.

The abbreviated module names are:

ac	language table actions
al	allocator
am	alarm
bt	boot routines
bz	buzzer
ca	calculator (LZ)
di	diary
dp	display
dv	devices
ed	edit
er	error handler
fl	files
fn	floating poin number functions
im	maths table actions
it	table routines
kb	keyboard
lg	language translator
lx	language lexer
mn	menu

mt	maths BCD routines
nt	notepad (LZ)
pk	datapacks
rt	run time
tl	top level
tm	time
ut	utilities
wl	world (LZ)
xf	extra file functions (LZ)
xt	extra utilities (LZ)

SYSTEM VARIABLES

The System Variables are described in detail at the System Variables Page.

BUFFERS

RTT_FILE ($22E9) - File Control Blocks

RTT_FILE contains 4 file control blocks with the following structure:

Byte 0	Type, the type of that file, between $90 and $FE
Byte 1	Device on which file exists (0 for A:, 1 for B: and so on)
Word 2	Current record number

PKT_ID ($20D7) - Pack IDs

Contains 4 pack IDs, these are the 10 bytes header of each pack. See Datapack ID String for details.

RTT_NUMB ($20FF) - Calculator Variables

Each memory, M0 to M9 has 8 bytes allocated to it. The format of each is the same format as a floating point number.

AMT_TAB ($22F9) - Alarm Table

Six bytes for each entry in the format:

Byte 0	Year
Byte 1	Month
Byte 2	Day
Byte 3	Hour
Byte 4	Minutes
Byte 5	Zero if canceled, Repeat flags

TIMING

There are a number of variables which are measured in ticks.

A tick is (KBW_TDEL + 35) / 921600. The default value of KBW_TDEL is $B3DD, which gives a tick of 50 milliseconds delay.

I/O DRIVER SPACE

At DVT_SPAR ($214F) 56 bytes are reserved for I/O drivers.

This space is allocated by PSION to officially supported devices. This area may not be used for any other purpose.

FREE MEMORY

Every time an OPL operand or operator is executed, it checks that at least 256 bytes is free - no operand or operator can increase the size of the stack by more than 256 bytes. Before increasing the size of an allocated cell the allocator checks there will be, at the end of the operation, at least 256 bytes free.

ALLOCATOR

There are 32 allocator cells available; the first 16 are pre-allocated to the operating system, the others are available for applications.

The allocator scheme is very simple. Each cell has one word associated with it (pointed to by a tag). If the cell is assigned it gives the start address of the cell, otherwise it is zero. The first non-zero word following gives the length of the cell by subtraction.

When a cell is grown all the allocated cells above it are moved up in memory, when a cell is shrunk all the cells above are moved down.

You may deduce from this that frequent growing and contracting of cells can slow an application down considerably.

If any cell is altered in size, for example by calling one of the allocator functions or by adding or deleting a record from device A:, any of the other cells may move. It is therefore essential to re-calculate the base address of a cell every time something could have moved it.

Tag	Name	Description of cell
$2000	PERMCELL	Permanent cell - for device driver code & data
$2002	MENUCELL	Top level menu cell
$2004	DIRYCELL	Diary cell
$2006	TEXTCELL	Language text cell - used when translating
$2008	SYMBCELL	Symbol table cell - used when translating
$200A	GLOBCELL	Global record cell - used when translating
$200C	OCODCELL	QCODE output cell - used when translating
$200E	FSY1CELL	Field name symbol table 1 - only used in OPL
$2010	FSY2CELL	Field name symbol table 2 - only used in OPL
$2012	FSY3CELL	Field name symbol table 3 - only used in OPL
$2014	FSY4CELL	Field name symbol table 4 - only used in OPL
$2016	FBF1CELL	File buffer 1 - only used in OPL
$2018	FBF2CELL	File buffer 2 - only used in OPL
$201A	FBF3CELL	File buffer 3 - only used in OPL
$201C	FBF4CELL	File buffer 4 - only used in OPL
$201E	DATACELL	Database cell - device A:
$2020 - $203E		Free for use by applications

When device drivers are loaded in, the code is fixed up and becomes non-relocatable. So once a device is loaded it cannot be moved which is why it is placed in the lowest cell.

When the Organiser is rebooted (by pressing the ON/CLEAR key at the top level) it first removes all devices currently loaded and then boots in all the device drivers from the devices B:, C: and D:. This also happens if the language is changed on multilingual organisers.

LANGUAGE

OPL is a stack based language. Almost every operand/operator has some effect on the stack size. The one exception is file buffers. When a file is opened the file buffer cell is grown to 256 and the field names' cell grown to take the field names. When a file is closed both these cells are reduced back to zero length.

For details of the buffer structure see section Records and Fields.

When OPL starts running, the stack is initialised to BTA_SBAS. If this is changed, the next time OPL is run it will start from the new address. The only thing which resets BTA_SBAS is resetting the machine.

PERMANENT MEMORY

For many applications it is sufficient to have permanent data in an allocated cell and to access it indirectly from the address of the cell.

For other applications it is desirable to be able to use a permanent fixed area of memory either for machine code or for data. An application can grab permanent memory in two different ways. It must be noted that if a number of applications use these techniques there is no reliable way to free the space selectively .

1. Lower BTA_SBAS by the amount of memory you want (this can be done in an OPL procedure).
2. Grow PERMCELL by the amount of memory you want, then poke a new higher address to ALA_FREE. Note that all devices should be rebooted after this as their internal addresses will no longer be valid.

The only way to automatically reset these two allocations are a TOP LEVEL RESET or, of course, a COLD BOOT.

SYSTEM SERVICES

Services that change the size of a cell will not move the base of the cell. However all cells which come after this cell will be moved.

Warning: The parameters are not checked. If incorrect values for tag or offset are supplied results will be unpredictable and potentially catastrophic. The user may write his own shell to protect against this bug.

AL$FREE

Frees a cell. A cell is freed by making its entry zero in the allocator table. This is the reverse of AL$GRAB.

AL$GRAB

Allocates a new memory cell.

As the allocator may move the base of a cell any time a request is made to the allocator, the base addresses of all cells are held in the table of addresses starting at ALA_BASE. Thus if the tag returned from this call is stored in a memory address called CELL then the following code should always be used to get the base of the memory cell into a memory address called BASE.

```
        LDX     CELL    ; Load the tag
        LDX     0,X     ; Get the de-referenced address
        STX     BASE    ; Save the real address
```

This procedure should be called any time a routine is called which could cause the allocator to move the memory cells around.

AL$GROW

Increase a cell.

The base of the cell will not move as the extra space is added after the start of the cell. However all cells which come after this cell will be moved to make room for the extra space.

AL$REPL

Increase or decrease a cell.

The base of the cell will not move as the extra space is added or deleted after the start of the cell. However all cells which come after this cell may be moved.

AL$SHNK

Decrease a cell.

AL$SIZE

Return the size of a cell.

AL$ZERO

Decreases the size of a cell to zero but does not de-allocate the cell.

FILING SYSTEM

GENERAL

This section describes the format of file-structured datapacks, then the system services available for file and record management. System services which access datapacks directly are discussed in chapter low level pack access.

In the following sections "datapack" means (E)EPROMs, ROMs on external devices, external RAM packs, and internal RAM (the device A:). The operating system handles the different device types in the same way apart from delete operations. All device types use the same record structure. Device dependencies will be pointed out as necessary.

RECORDS

Datapacks contain two types of records, preceded by either a byte or word length. This is to al low long records while minimising the overhead for text files which typically contain short records. The first record is always at byte $0A in the datapack. The record structure is terminated by a byte $FF.

SHORT RECORDS

Short records are of the format:

Byte	0	1	2...
	length	record type	1 to 254 data bytes...
Range	1 to 254	$81 to $FE	

Example:

"HELLO" saved using SAVE. Record type is $90.

$05 $90 $48 $45 $4C $4C $4F

Note that the length byte does not include itself or the record type. The record length can not be zero. The maximum length is $FE, because a first byte $FF is used to terminate the record structure. The record type is in the range $81 to $FE inclusive. When writing to more than one file, the record type is used to identify which file each record belongs to, since the records may be written in any order.

LONG RECORDS

Long records are used primarily for saving blocks of data which will not be regularly altered, which are intended to be loaded and saved in one piece. Long records are also used to contain information which is to be hidden from the file system.

Long records have the following format:

Byte	0	1	2	3	4...
	$02	$80	length word		data...

Example:

long record "HELLO"

$02 $80 $00 $05 $48 $45 $4C $4C $4F

null long record

$02 $80 $00 $00

Long records do not have a length byte as such, the $02 is purely for error handling (see below). The $80 is a special record type which identifies a long record.

DELETED RECORDS

On EPROM datapacks, short records are marked as deleted by clearing the top bit of the record type. On EPROMs long records are never deleted - this will be discussed later. On a RAM device, the space occupied by a record is recovered when the record is deleted, so there will not be any long records with a record type < $80 on either RAM or EPROM devices.

Example:

short record "AAA" of record type $91 on an EPROM

$03 **$91** $41 $41 $41

when deleted becomes:

$03 **$11** $41 $41 $41

DATA FILES

There are two kinds of files; data files accessible to OPL (of which the file "MAIN" is an example), and block files such as OPL procedures. In future we will refer to the first kind simply as 'files'.

Each file has a name, which is a short record of type $81, and a number of data records which are short records all of the same record type in the range $90 to $FE. The name and data records of a file can be interspersed by any number of other records and files. Also the name need not come before the first record of a file.

Each file has data records of a unique record type, so there can be up to 111 files on a device. An attempt to create more files will produce the error "DIRECTORY FULL". A file may contain up to $FFFE data records, given a sufficiently large pack.

Filenames have the following format:

Byte	0	1	2-9	10
	$09	$81	file name, space filled	record type

Example: the file MAIN (all records of type $90 will be in MAIN)

$09 $81 $4D $41 $49 $4E $20 $20 $20 $20 $90

Filenames are always padded with spaces to make them eight bytes long. Byte 10 in the above diagram is the record type which will be used for all data records in the file. It can take values in the range $90 to $FE inclusive.

When a file is created, the operating system first searches through all the filenames on the datapack to find the lowest free record type and this is then allocated to the new file. The new record type is then stored in the new file name record. Since several files can be open at once, the records of a file can be mixed with any other type of records. No facilities are provided to allow reordering of records in a file, or to allow the insertion of records in the middle of a file.

When a file is deleted, all records of the appropriate type are deleted one by one, then the filename is deleted. On EPROMs the filename is deleted by overwriting the record byte (i.e. $81 in the above example) in the filename with $01.

The total overhead for each file is 11 bytes for the filename plus two bytes per record.

FIND, SAVE AND "MAIN"

The top level commands FIND and SAVE work with short records of type $90 directly, not by opening the file MAIN. This means that if MAIN is deleted, problems can occur. SAVE will still save records of type $90, but these records can not be re-read by OPL. The record type $90 is 'reserved' for MAIN, so that no other file can have this record type. When blank packs are sized, the filename MAIN is created.

Note: User programs should never delete MAIN, since this will cause problems with certain OS versions. To create a new MAIN, delete all the records in a loop.

BLOCK FILES

Block files are a special class of files with a name immediately followed by a single long data record. No records are allowed between the block filename and the long record. They are intended for storing data which will be re-saved as a whole each time the data is changed. Long records may be used alone without a preceding block filename - for example to enclose a machine code program on a datapack. Long records with no preceding block filename are ignored by the file system.

Format of a block file:

Byte	0	1	2-9	10	11	12	13 - 14	15...
	$09	TYPE	file name	$00	$02	$80	length	data...

Example: block file ABCD of block type $83, containing four bytes of data.

$09 $83 $41 $42 $43 $44 $20 $20 $20 $20 $00 $02 $80 $00 $04 xx xx xx xx

The record type, byte 1 in the above diagram, is in the range $82-$8F inclusive. This byte is used to distinguish different classes of block files containing different kinds of information, and will be referred to as the block file type. Because block files have their data immediately after the filename they do not need a data record type, so byte 10, which was used as the data record type in ordinary filenames is not used in block files. This byte is reserved by Psion. Note also that the number of block files on a device is limited only by the space available, and there can be up to fourteen different kinds of block files with types $82 to $8F.

On EPROMS, a block file is deleted by clearing the top bit in the record type byte of the name (byte 1, the $83 in the above example). The total overhead for each block file is 15 bytes.

OPL PROCEDURES

OPL procedures are saved as block files of type $83. The data block has the following format:

Byte	0	1	2...	word			
	length of Q-code	Q-code (if any)	length of source	1st source line	$00	2nd source line	$00...

The length of the Q-code may be zero, if the procedure was saved with the SAVE option in the TRAN SAVE QUIT sub-menu or was received from PC. The length of the source may be zero if the procedure has been copied object-only. Each line of the source is terminated by a zero.

The DIR option in the PROG sub-menu works by finding all type $83 records.

CM/XP DIARY FILES

These are block files of type $82. Its data block consists of a list of entries which is terminated by a zero byte. Each entry has the form:

Byte	0	1	2	3	4	5	6	7...
	length of the text	year	month	day	hour	minute	alarm	text...
Range		0-99	0-11	0-30	0-23	0-59	0-60	

The alarm flag is 0 for no alarm, otherwise one more than the number of minutes early the alarm has to go off.

LZ DIARY FILES

On LZ machines the diary is no longer saved as a block file of type $82, but instead as an ordinary data file. Each entry is saved as a separate record.

For example

```
Mon 8 May 1989 Wk19
6:00a
5:15p ABCDEF
5:45p
```

would be:

```
1989050817150216
ABCDEF
```

The first field specifies the date of the entry (1989, 05, 08, for 8th May 1989), the time (17 15, for 5:15pm), the duration (02) and the alarm byte (16, for 15 minutes before - again, 00 would mean "no alarm"). The second field is the text of the entry. The order of things in the first field, and the use of zeroes where necessary to keep things the same length (e.g. "02" for a duration of 2) make the file easy to sort in OPL, or otherwise.

Note, though, that the "Restore" option in the diary will restore entries in any order from a file. The "Xrestore" option can be used to load a CM/XP diary from a pack, in which case each entry is given a duration of 2 (=30 minutes).

COMMSLINK SETUP FILES

These are block files of type $84. Its data block consists of 27 bytes exactly, containing the parameters:

Byte	0	1	2	3	4	5	6
	baud	parity	data bits	stop bits	handshake	minute	echo
Range	0-9	0-4	0-1	0-1	0-7	0-2	0-1

Byte	7	8	9-11	12-15	16-19	20-23	24-27
	width	timeout	REOL	REOF	RTRN	TEOL	TTRN
Range	0-254	0-255	string	string	string	string	string

All string have a maximum length of 2 and a leading length byte.

LZ NOTEPAD FILES

These are block files of type $87. Its data block consists of two parts, the header, and the data, both preceded by a length word.

Header:

Byte	0	2	3	4...
	header length	flags	$00	title+":"

If a password was used:

Byte	0	2	3	4-13	14...
	header length	flags	$09	password code	title+":"

Flags: The bits are set as for the LG$EDIT service, so Bit 7 is set if the notepad is numbered, bit 3 must be set, bits 1 and 2 are clear.

Data:

Byte	0	2	3...		
	data length	$00	1st line	$00	2nd line...

OTHER BLOCK FILES

Other block file types:

$85 Spreadsheet files

$86 Pager setup files

$8E General use

PSION states that all other types are reserved. These might be used by PSION or third-party software.

Block file types are important in keeping different types of data distinct: disaster would result if OPL procedures could be loaded into the diary, for example. Applications tend to assume that block files of their own type are in the correct format, and usually perform no further checking.

SUMMARY OF RECORD STRUCTURE

This is the algorithm for scanning through the record structure of a datapack:

```
SET_PACK_ADDRESS ($0A)
do
     LENGTH_BYTE = NEXT_BYTE
     RECORD_TYPE = NEXT_BYTE
     if LENGTH_BYTE = 0
         ERROR (246) -- "no pack"
         stop
     endif
     if RECORD_TYPE = $80        -- if long record
         BLOCK_LENGTH = NEXT_WORD
         SKIP_BYTES (BLOCK_LENGTH)
     else
         if RECORD_TYPE <> $FF   -- if valid short record
             SKIP_BYTES (LENGTH_BYTE)
         endif
     endif
 until LENGTH_BYTE = $FF         -- found end of pack
```

ERROR HANDLING

If a length byte of zero is seen the pack is assumed to have been pulled out, and a NO PACK error is reported. Errors can occur in various places when writing to EPROM devices, which can disrupt the record structure. The file system error handling attempts to mark the data as deleted, or creates special 'invalid' records so that the remainder of the datapack can still be used. In certain exceptional cases this mechanism can fail, and a "READ PACK" error will result when invalid record structure is detected. This error is reported whenever the last record extends beyond the end of the pack. An "END OF FILE" error will also be reported when accessing a block file if the block file name is not followed immediately by a long record.

When writing a block file, any error will cause the operating system to first delete the long record, then the block file name. This is the reason for the $02 before the $80 in a long record - if writing the length word fails, the $80 is re-written as a zero, which forms a deleted short record enclosing the bad length word. When a block file is opened, if a valid long record beginning with $02 $80 is not found, the error "END OF FILE" (238) is reported.

When writing a short record the following errors may occur:

1. When writing the length byte fails, no further bytes are written to the record. The record is left with a second byte $ff, which is taken as a special invalid record type. If this type of record is seen the bad length byte is ignored, and the next record is assumed to begin immediately after the $FF. This prevents a record with a bad length byte from going off the end of the pack. However, the method is fallible because the length byte could be $FF, denoting the end of the datapack, or could even be zero (although zero is very unlikely).

2. When writing the record type or a data byte fails, the operating system attempts to delete the record. If this fails, a spurious record will remain, which may become part of the wrong file since the record type is wrong.

Some of the errors such as "PACK NOT BLANK", and "WRITE PACK ERROR" may only occur when sizing a blank EPROM device. Apart from the obvious, "END OF FILE" can mean: an illegal block file (as discussed above), or more generally "not found".

SUMMARY OF RECORD TYPES

Record type	
00	invalid long record - ignored
01	deleted data file name
02 - 0F	deleted block file name, following long record is regarded as deleted
10 - 7E	deleted short records
7F	deleted $FF (should not occur)
80	long record - length word follows
81	data file name
82 - 8F	block file name, followed by a long record
90	record from MAIN, top level FIND/SAVE record
91 - FE	data records from files
FF	invalid record - length byte will be ignored

Example of record structure:

```
09 81 4D 41 49 4E 20 20 20 20 90  filename "MAIN"
04 90 41 41 41 41                 record "AAAA" in MAIN
09 81 41 42 43 20 20 20 20 20 91  filename "ABC",
                                  data records are type $91
03 91 42 42 42                    record "BBB" in ABC
01 10 41                          deleted record "A" in MAIN
09 85 42 4C 4F 43 4B 20 20 20 00  block file "BLOCK", type $85
02 80 00 05 01 02 03 04 05        contains 5 bytes of data
09 02 4F 4C 44 20 20 20 20 20 00  deleted block file "OLD",
02 80 00 01 FF                    of type $82 (diary),
                                  contained 1 byte of data
F7 FF                             invalid short record
09 03 42 41 44 20 20 20 20 20 00  invalid block file "BAD"
02 00 00 FF                       of type $83 (OPL proc),
                                  deleted when length word
                                  failed


FF                                end of datapack
```

FILE SYSTEM VARIABLES

Some of the file system's variables can be usefully read by the user, but these variables should be treated as **read only**. Writing to these variables may produce unpredictable results, and will cause incompatibility with future OS versions.

$96	FLB_RECT	Current record type in use. Set by FL$RECT, and implicitly set by FL$OPEN, FL$CRET etc.
$97	FLB_CPAK	Current device used by file system Set by FL$SETP, and implicitly set by FL$OPEN, FL$CRET etc.
$9B	FLW_CREC	Current record number - 1 is the first record. Set by many routines including FL$RSET, FL$NEXT.

ORGANISER I COMPATIBILITY

Organiser I datapacks are structured differently to Organiser II datapacks - the details will not be discussed here. The main differences are:

1. Organiser I datapacks have a first byte $FC - a value not used by the Organiser II. These packs are treated as read-only by the Organiser II.

2. Organiser I program packs have a first byte of $03; these packs are not supported by the organiser II.
3. Records saved on Organiser I datapacks with SAVE are stored in a packed six-bit format, which can be read by the Organiser II. These records are treated as part of an imaginary file "MAIN". All read-only file system operations on "MAIN" will work on an Organiser I datapack, including COPY. Other types of records, such as POPL programs are not supported.

The Organiser I datapacks themselves are all compatible with the Organiser II, and can be used on the Organiser II after UV-erasure. Also there is no reason why a special user program could not read an Organiser I datapack, by accessing the datapack directly. If some special data other than SAVEd records is to be transferred, the easiest method is to use two RS232 leads.

SYSTEM SERVICES

This section briefly describes the system services used by OPL to handle files, records and block files. The names of these services are prefixed by FL$. A detailed explanation is given on the system services page.

It is more efficient to use record types directly to access several files at once, rather than opening one file, then the other. This is the equivalent of the OPL command USE. The services FL$OPEN, and FL$CRET return the record type in use by a file. FL$FREC will provide details of a record including its address in the pack for any user wishing to perform direct pack accessing, however such programs may not be compatible with future OS versions.

Programmers should note that calls to the FL$ services may be interspersed with PK$ calls - which may change currently selected device or change the current pack address - provided that PK$SETP is called to re-select the correct device for the file system before calling any further FL$ services. As always, any programs directly accessing the datapack hardware should notify the OS by calling PK$SETP.

FL$BACK

Sets the current file position back to the previous record.

FL$BCAT

When called repeatedly, returns each filename of a given record type on a device.

FL$BDEL

Delete a named block file. On RAM devices the block file name and the following long record is deleted and the space is freed. On EPROMs the long record is not affected.

FL$BOPN

Finds a named block file of the given block file type. It returns the length of the data, and the pack address is set to the start of the data, ready for PK$READ to load the data into RAM.

FL$BSAV

Called in preparation for saving a block file, FL$BSAV saves a block filename followed by the first four bytes of a long record: $0280 and the length word. Then a call to PK$SAVE must be made to save the data. FL$BSAV checks that there is sufficient room on the pack for both the filename and the long record before writing to the datapack.

FL$CATL

When called repeatedly, returns each the name of each file on a device.

Calling FL$CATL is equivalent to calling FL$BCAT for file type $81.

FL$COPY

Copies files or block files from one device to another, as in the top level COPY or PROG COPY menu options. The copy-from device must not be the same as the copy-to device. A file may be copied to a different name on the target device. If a device only is specified in the copy-to string, the file is copied with the same name.

If the file already exists on the TO device then the records will be appended to the file otherwise a new file of the appropriate name will be created.

FL$CRET

Creates a file. If the file already exists then FL$CRET returns error "FILE EXISTS", otherwise the file is created and the record type to be used by the data records in the file is returned

FL$DELN

Delete a named file.

FL$ERAS

Erase the current record in the current file. On EPROMs this is done by clearing the top bit of the record type. On RAM devices the space occupied by the record is recovered.

FL$FFND

Searches through all the records of a particular type on current datapack for a record beginning with a given search string. Also used to find file names.

FL$FIND

Finds the next record of the current record type on the current datapack which contains the given search string. If a match occurs, FL$FIND leaves the current file position on the found record, if there is no match, reports error "END OF FILE" and leaves the current position at the end of file.

FL$FREC

Returns information about the n'th record of the current record type on the current datapack. If the record exists, the three byte pack address of the start of the record is returned.

FL$NEXT

Adds one to the current record number. The next file operation, such as FL$READ will now read the next record of the current type.

FL$OPEN

Opens a previously existing file.

FL$PARS

Checks that a file name is legal, and converts it to a standard form (D:Name where D is the device 'A'..'D', and the filename is between one and eight characters long.

FL$READ

Read the record from the current position of the current record type into not memory.

FL$RECT

Sets the 'current record type' to the given value.

FL$RENM

Changes the name of a file. Files can only be renamed onto the same device. The old name is deleted and then the new name is saved. On an EPROM device, this means that only eleven more bytes used by the new filename are saved.

FL$RSET

Sets the 'current record number' to a given value

FL$SETP

Selects a datapack. This datapack will then be used by other file system services.

FL$SIZE

Returns statistics about the current datapack, and current file or current record type. Counts the number of records of the current type, finds the end of the current datapack and returns the amount of free space.

FL$WRIT

Appends a new record of the current record type to the records on the current device. Records of length 255 are truncated to 254 characters. The current record number is set to the number of records of the current type.

FL$WRIT checks that there is sufficient space free first.

TL$CPYX

Performs the top level COPY, complete with "FROM" and "TO" prompts.

FL$WPAR (LZ only)

Validates wild card filenames. Returns length of body of filename and the file type for the extension.

FL$WCAT (LZ only)

Provides a wild card catalogue of the file names on a device. Works in the same way as FL$BCAT and FL$CATL except that it returns only those files which are consistent with the match string.

FL$NCAT (LZ only)
Same as FL$WCAT but gets the n'th match.

FL$WCPY (LZ only)
Wild file copy, works in a similar way to FL$COPY.

FL$WDEL (LZ only)
Wild file delete, works in a similar way to FL$DELN and FL$BDEL except that wild card characters and file extensions are allowed.

FL$WFND (LZ only)
Wild FIND, works exactly like FL$FIND but the wild card characters *, +, can be used.

FL$FDEL (LZ only)
Fast delete of a given number of records starting from a given record. If the number is -1 this deletes to end of file.

FL$GETX (LZ only)
Get the file extension for a given numeric file type ($81 to $8F)

FL$VALX (LZ only)
Returns the numeric file type for an extension.

See also chapter Packs for PK$READ and PK$SAVE.

SYSTEM TIMING

GENERAL

System timing is controlled by two interrupts:

1. The NMI interrupt handles the real-time clock and auto-switch off delay.
2. The KEYBOARD INTERRUPT provides a frame-counter and handles keyboard delays and display timing etc.

REAL TIME CLOCK

The clock time is stored in binary in 6 fields from $20C5 to $20CA:

VARIABLE	ADDRESS	DESCRIPTION	RANGE
TMB_YEAR	$20C5	YEAR	0 - 99 0 - 255 (LZ)
TMB_MONS	$20C6	MONTH	0 - 11
TMB_DAYS	$20C7	DATE OF MONTH	0 - 30
TMB_HOUR	$20C8	HOUR	0 - 23
TMB_MINS	$20C9	MINUTES	0 - 59
TMB_SECS	$20CA	SECONDS	0 - 59

Note that:

1. A month of 0 represents JANUARY and a date of 0 represents the 1st day of the month.
2. On a cold start only, the clock is initialised to 1 JAN 1987 00:00:00
3. The real-time clock should not be read directly from these variables because it may be being updated by an NMI. Instead system service TM$TGET should be used to get a valid time. Similarly these variables should not be written to without checking for an NMI.
4. With **Models CM, XP and LA** TMB_YEAR rolls over to 00 after 99. This behavior is known as the **y2k-bug**. It is possible to poke a higher value into TMB_YEAR but the date is nevertheless displayed incorrectly and TMB_YEAR will again roll over to 00 at the end of the year.
5. The time stored always represents G.M.T., so on LZ machines the world times and daylight-saving are calculated as offsets from this.

On the LZ there is another time-related system variable:

TMB_24F ($20A6)
TMB_24F stores 2 flags:

- Bit 7 is set for daylight-saving ON, clear for OFF.
- Bit 0 is set for 24 hour mode, clear for 12 hour mode.

These flags can be read or written to at any time.

KEEPING TIME WITH NMI ON

An NMI interrupt is generated from the semi-custom chip every second to provide an accurate real-time clock. When the machine is on, the NMI interrupt updates the time by 1 second, see TM$UPDT.

KEEPING TIME WITH NMI OFF

When NMIs are switched off the processor, (e.g. when the machine is switched off) an internal counter in the semi-custom chip is connected to the NMI line so that NMIs can still be counted, enabling the time to be updated when restoring NMIs to the processor.

The counter has 11 bits, so the maximum time it can store is 2048 secs. When the counter reaches this value, the ACOUT bit of PORT 5 goes high and the machine switches on. Hence the machine can be forced to switch on automatically at any time up to 34mins 8secs after switching off, be pre-counting the counter. This is done when an alarm is due.

Whenever the machine switches on it updates the real-time clock by the amount in the counter (less any pre-counting). If ACOUT is high, and no alarm was due, the machine switches back off immediately after updating the clock. This can be seen as the screen flashes on for an instant, every 34 mins and 8 secs while the machine is off.

To disable NMIs and keep the time, system services BT$NOF and BT$NON should be used.

AUTO-SWITCH-OFF TIME OUT

The following three variables control the auto-switch-off:

VARIABLE	ADDRESS	DESCRIPTION
TMB_SWOF	$007C	AUTO-SWITCH-OFF FLAG
TMW_TOUT	$007D,$007E	TIME LEFT BEFORE SWITCH OFF
TMW_TCNT	$20CD,$20CE	DEFAULT NUMBER OF SECONDS TO TIME-OUT

The time before the machine switches off (in secs) is stored in TMW_TCNT and is set to $012C on cold start (5 mins). The contents of TMW_TCNT are copied into TMW_TOUT whenever a key is pressed or KB$GETK is called and TMW_TOUT is decremented until zero by the NMI routine every second. If TMW_TOUT is found to be zero in KB$TEST, the machine will switch off.

The contents of TMW_TCNT can be changed at any time to alter the auto-switch-off time, up to a maximum of 65535 seconds (18 hours, 12 mins and 15 secs). If it is set to less than 15, the machine will will still stay on for 15 secs.

To disable the auto-switch-off completely, TMB_SWOF should be set to zero. This will inhibit the NMI from decrementing TMW_TOUT and prevent KB$TEST testing it.

KEYBOARD INTERRUPT TIMERS

The timer 1 compare interrupt is used to scan the keyboard to allow keyboard buffering and to provide a timing service. The time between interrupts is controlled by the variable KBW_TDEL which is initialised on cold start to be $B3DD. This value makes the KI interrupt occur exactly every 50 milliseconds and is used extensively by the operating system for timing purposes.

TMW_FRAM

TMW_FRAM is incremented by 1 on each keyboard interrupt. When $FFFF is reached, it wraps back to $0000. It can be read at any time and used for accurate timing.

DPW_REDY

DPW_REDY is decremented by 1 on each keyboard interrupt until zero is reached. It can be used to provide delays (e.g. TM$WAIT stores D in DPW_REDY and waits for it to reach zero - see below).

SYSTEM SERVICES

Note that some services work from any buffer containing a 6 byte time representation exactly like the real-time clock (TMB_YEAR...). These routines should not operate on the real-time clock itself if an NMI is imminent, so NMIs must be checked for or TM$TGET should be used to copy the time to another buffer.

TM$DAYV

Calculates the day of the week for a given date.

TM$TGET

Get a copy of the real-time clock into a buffer. It is not possible to read the clock directly in case it is being updated by an NMI.

TM$UPDT

Updates a time buffer by a given number of minutes and seconds.

TM$WAIT

Waits for a number of ticks (1 tick is the interval between keyboard interrupts, controlled by KBW_TDEL and set to 50ms by default). If interrupts are disabled (I flag set) then this routine waits for the given number x 50ms.

TM$NDYS (LZ only)

Returns the number of days since 01/01/1900

TM$WEEK (LZ only)

Returns the week number of the supplied date. The first MONDAY of the year starts WEEK 0. Any dates before this MONDAY are in the last week of the previous year. Applications (e.g. the DIARY) convert the week number from 0-52 to 1-53.

TM$DNAM (LZ only)

Returns a 3 letter day name for a given day of the week in the currently selected language.

TM$MNAM (LZ only)

Returns a 3 letter month name for a given month in the currently selected language.

TM$TSET (LZ only)

Sets the system time to a given time. The time should not be simply poke'd into the system time variables since an NMI may be updating the time.

SYSTEM BOARD

GENERAL

Internally the Organiser consists of two circuit boards. The system board holds all the digital electronics and has integral interfaces to the display and keyboard. The power supply board controls power regulation and distribution to the system and also carries connectors to the I/O slots and buzzer. The two boards are connected together by a 27 way strip connector.

This section describes the system board hardware, and the following two sections complete the Organiser hardware description.

The System board is a CMOS 8 bit computer including the following:

- 8 bit HD6303X processor running at 0.912 MHz
- 32 kbyte ROM containing system code
- 8-96 kbyte static ram option slots
- dot matrix LCD and driver ICs
- PCB pads and interface for 36 key keyboard
- real time clock running from 32768 Hz crystal
- 27 way connector to PSU board for external I/O, power and power control

The board has been engineered to minimise space and power requirements. Small size is achieved through the extensive use of surface-mounted components and by design of a semi-custom IC to perform control functions. Low power is achieved by the use of CMOS circuitry throughout and by taking advantage of the special power saving modes of the processor.

CIRCUIT DESCRIPTION

This section describes the circuit in general terms, and specific areas are covered in more detail later.

There are 8 positions for ICs on the board, some of which are optional to provide different memory configurations. All ICs are CMOS with low power standby modes, and all are surface-mounted.

IC1 is the HD6303XFP microprocessor in an 80-pin flat package. This is an 8 bit processor derived from the 6800 family, with standard 8 bit data and 16 bit address busses. In addition it has three 8 bit I/O ports inbuilt. An oscillator provides an input frequency of 3.6864 MHz, which is divided by four internally to an operating frequency of 0.9216 MHz. Processor startup and shutdown are controlled by the STBY_B and RES_B signals from the control IC.

IC2 and IC3 control the LCD display. The HD44780 (IC2) is the master driver with inbuilt character-generator ROM and display data RAM. On the LZ, it was replaced by a customised 66780 chip. The LCD is accessed in the processor memory-map, decoded by the EOUT signal from the control IC. IC3 is a slave driver to extend display width to 16 (20) characters. The LCD plate is mounted to the board through conductive rubber connector strips. The display has either two rows of 16 characters (models CM, XP) or four rows of 20 characters, with each character as an 8 by 5 dot matrix.

IC4 is a semi-custom IC to supervise circuit operation. It's main functions are:

- To decode and select devices in the processor memory-map. Address lines A6-A15 are input and are used together with the processor E clock to decode memory device blocks (CS1_B to CS6_B), the LCD driver (to the EOUT signal) and internal control latches. Internal latches are used to control a variety of functions including processor startup and shutdown, polling the keyboard, clock output to the processor NMI interrupt, the buzzer ALARM signal and the PULSE signal to the power supply.
- To provide the real-time clock facility there is a 32768 Hz oscillator circuit. This is divided down in two stages. The first stage provides a 1 Hz output which is normally switched to the processor NMI input to update the clock when the processor is on. When the Organiser is off, the second stage is used to record elapsed time since switch-off, and when the Organiser is next switched on the processor reads the elapsed time to re-synchronise its time registers. The Organiser is automatically switched on if the second stage divider times-out, after an interval of 34 minutes 8 seconds.
- To control processor startup and shutdown sequences. Startup is initiated by either a rising edge at the AC input or by timeout of the second stage counter. The ON_B signal to the power supply immediately goes low to switch on supply rail regulators. After a time delay to allow power rails to settle, the processor is started by sequencing the STBY_B and RES_B signals. Shutdown is initiated by the processor itself, by accessing a control latch. STBY_B and RES_B are immediately brought low, and the ON_B signal set high.
- To poll the keyboard. The keyboard is arranged as a 7 by 5 switch matrix. The seven outputs K1-K7 are open-drain, and to poll the matrix they are set low one at a time, with the rest floating. At each step the processor reads the inputs KBD1-KBD5 via port 5, detecting a key pressed if an input is low. The processor controls the K1-K7 outputs via control latches within the control IC. The AC key is separate to the matrix, being input directly to the IC4 AC input and to a separate bit of the processor port 5. The keyboard switches are PCB pads underneath the Organiser keys. When a key is pressed a conductive rubber pad shorts the key contacts.

The control IC is reset on "cold start" i.e. when power is first applied to the board.

Power to the board is supplied through the VCC1, VCC2 and V_LCD rails from the power supply. VCC1 is always present and powers all circuit ICs except the LCD drivers. Power consumption is typically 30 microamp when the Organiser is off, mainly due to the real-time clock oscillator. RAM data in both the processor and external RAM devices are retained in this mode. When the Organiser is on (ON_B low) the VCC2 and V_LCD rails are switched on to power the LCD drivers. Contrast adjustment is achieved by adjusting the V_LCD voltage. Power consumption in this state can be 20 milliamp with the processor running code, reducing to 4 milliamp when the processor is set into its "sleep" mode. Capacitors C6-C8 decouple the VCC1 rail.

MICROPROCESSOR

The HD6303XFP processor is a member of the 6301-6303 family (which are CMOS parts derived from the 6800 series). This member is a romless part with a full 64k external memory-map, 1 MHz maximum operating speed and mounted in an 80 pin flat package.

OPERATING MODES

The processor has five operating states: standby, reset, active, halt and sleep. The Organiser does not use halt mode, and reset is only used during the switch-on sequence as a transition between standby and active. The three remaining states are used in the following way:

1. Standby mode:
   When the Organiser is powered but is switched off, the processor is in this state (with the SBY_B and RES_B pins both low). The processor is inactive with all port pins tri-state and the oscillator shut down. In this state the power consumption of the processor is negligible, and the internal RAM is retained.
2. Active mode:
   When the Organiser is switched on, the processor is put into Reset mode (SBY_B high and RES_B low) for 30 to 60 millisecs to allow the oscillator and E clock to start up, and is then set into Active mode (SBY_B and RES_B high). The memory busses are made active and the processor starts running code. The I/O ports remain set as inputs until initialised by the software. In active mode the processor is in control of the whole circuit. To return to Standby mode (Organiser off) the processor accesses a switch-off address in its memory-map. This triggers an "ON/OFF" latch in the control IC which immediately sets SBY_B and RES_B low.
3. Sleep mode:
   This is used to reduce power consumption when the processor is active, and is entirely under software control.

MEMORY_MAP

Processor ports 1,3,4 and 7 control the memory-map. Ports 1 and 4 form the 16 bit address bus A0-A15, port 3 the 8 bit data bus D0-D7, and port 7 supplies the control lines R_B, W_B and R/W_B. External access cycles are decoded and synchronised with the E clock by the control IC.

The memory map is assigned in four address areas:

- $8000-$FFFF are assigned to external ROM devices. 8,16 or 32 kbyte may be present, filled from the top down. Later models (multilingual XP, LZ) have 64k of ROM. The ROM from $8000 to $BFFF is bank-switched in 3 banks of 16k and the ROM from $C000 to $FFFF is fixed.
- $0400-$7FFF are assigned to external RAM. 8,16 or 31 kbyte may be present, filled from $2000 up except for the 31 kbyte option which is from $0400 up. The LZ64 has 63k. The RAM from $4000 to $7FFF is bank switched in 3 banks of 16k and the RAM below $4000 is fixed.
  The POS350 has 95k. No technical information for the memory layout is available, but most likely the RAM from $4000 to $7FFF is bank switched in 5 banks of 16k and the RAM below $4000 is fixed.
- $0100-$03FF are assigned to external I/O devices including the LCD and latches in the control IC.
- $0000-$00FF are reserved for internal processor RAM and registers.

External devices are described further in the following sections.

MEMORY DEVICES AND OPTIONS

ROM

System software is carried on the board in the form of ROM (strictly speaking they are One-time programmable CMOS EPROM devices). Four options are catered for:

- 8 kbytes at addresses $E000-$FFFF
- 16 Kbytes at addresses $C000-$FFFF
- 32 Kbytes at addresses $8000-$FFFF
- 64 Kbytes: 3 banks of 16 Kbytes at addresses $8000-BFFF and 16 Kbytes at address $C000-FFFF.

Note that in all options the processor restart and interrupt addresses at the top of the memory-map are included in the ROM area.

The ROMs used are byte-wide CMOS devices in 28 pin flat packages, with access times of 250 ns or better. The types used are:

- 27C64FP with 8 kbyte capacity
- 27C256FP with 32 kbyte capacity
- the LZ may use other chips (no info available)

ROMs installed are powered at all times, and have a typical power consumption in standby mode (with the CS_B pin high) of 1 microamp.

RAM

As with the ROM above, there are different options for RAM in the memory map:

- 8 Kbytes at addresses $2000-$3FFF
- 16 Kbytes at addresses $2000-$5FFF
- 31 Kbytes at addresses $0400-$7FFF
- 63 Kbytes: 3 banks of 16 Kbytes at address $4000 to $7FFF and 15k at address $0400-$4000
- 95 Kbytes: No technical information for the memory layout is available, but most likely the RAM is organised like the 63 kbyte option except that there are 5 banks of 16k.

Note that address $2000 is used by the system for the start of system variables by all options.

The RAMs used are byte-wide CMOS devices in 28 pin flat packages, with access times of 250 Ns or better. The types used are:

- 6264LFP with 8 kbyte capacity
- 62256LFP with 32 kbyte capacity
- the LZ may use other chips (no info available)

RAMS installed are powered at all times, and hence retain their data when the Organiser is off. In the standby mode their power consumption is typically 2 microamp.

For the more than 16 kbyte options, a bigger chip is used but the bottom 1 kbyte ($0000-$03FF) is never accessed.

In addition to the RAM devices above, their are two other areas of RAM on the board and common to all options:

- The processor includes 192 bytes of RAM at addresses $0040-$00FF, and this is also retained when the Organiser is off.
- The LCD driver has its own RAM for display data. This is not in the processor memory map and is not retained when the Organiser is off. However UDGs (user defined graphics) are retained by the OS from version 2.6 on.

MEMORY DECODING AND LINKS

The six memory selection signals from the control IC (CS1_B to CS6_B) are mapped to the following memory areas:

- CS1_B $8000-$FFFF (32k)
- CS2_B $E000-$FFFF (8k)
- CS3_B $C000-$DFFF (8k)
- CS4_B $4000-$5FFF (8k)
- CS5_B $2000-$3FFF (8k)
- CS6_B $0400-$7FFF (31k)

They are normally high, and go to their active-low state when the relevant memory area is addressed by the processor. These six outputs cover all ROM/RAM options, and a maximum of four can be used at any time.

Note that the links options below are valid for models CM,XP and LA only (they may also be valid for the models LZ and LZ64 but there is no technical information available).

Links L1-L8 on the board are used to match the correct signals to the available memory options. They are arranged as four pairs: L1-L2, L3-L4, L5-L6, and L7-L8. Of each pair only one should be fitted, with the other left open-circuit

If a 32 kbyte ROM is fitted in IC5 then links L1 and L4 should be fitted. L1 routes CS1_B to the ROM to decode it in the $8000-$FFFF range. L4 makes the A14 address line available to the ROM.

If an 8 kbyte ROM is fitted in IC5 then links L2 and L3 should be fitted. L2 routes CS2_B to the ROM to decode it in the $E000-$FFFF range. L3 pulls the ROM pin 27 high since A14 is not required.

If a 32 kbyte RAM is used in IC8 then L6 should be fitted, to route the CS6_B signal to the RAM and decode it in the $0400-$7FFF range.

If an 8 kbyte RAM is used in IC8 then L5 should be fitted to route the CS5_B signal to the RAM and decode it in the $2000-$3FFF range.

Links L7 and L8 set the state of the control IC CTRL input. L7 is normally fitted, and in this case the CS1_B to CS6_B outputs are internally gated with the processor E clock so that they are active only when the E clock is high. If L8 is fitted the decode outputs are dependent on the address lines only.

BANK SWITCHING

The bank-switching is carried out by accessing the following addresses (reading or writing):

- $360 - Reset ROM/RAM to bank 0
- $3E0 - Select next ROM bank
- $3A0 - Select next RAM bank

Note that the operating system may switch ROM banks when any system service is called and during interrupts.

MEMORY MAPPED I/O

ADDRESS ASSIGNMENT

The previous sections have covered memory areas $0000-$0100 (processor internal functions) and $0400-$FFFF (memory devices). The area between these ($0100-$03FF) is used to decode the LCD and latches within the control IC. The control IC decodes these from its address inputs A6-A15, and since A0-A5 are not available each function must span addresses in blocks of 64 bytes or multiples of this. The functions and their address ranges are:

$0100-$017F	not used
$0180-$01BF	LCD ENABLE
$01C0-$01FF	SWITCH OFF
$0200-$023F	PULSE ENABLE
$0240-$027F	PULSE DISABLE
$0280-$02BF	ALARM SET

$02C0-$02FF	ALARM RESET
$0300-$033F	COUNTER RESET
$0340-$037F	COUNTER CLOCK
$0380-$03BF	NMI ENABLE
$03C0-$03FF	NMI DISABLE

The LCD ENABLE function is a simple decoding one which is output to the EOUT signal on pin 39. This is normally low, and is set high when any address in the range is selected. The LCD is covered further in section LCD display. All the other functions listed perform actions within the control IC which are address-controlled, latched events. Address-controlled means that any processor access to an address within the range will cause the event, irrespective of whether it is a read or write access or of data on the data bus. Once an access has occurred, the affected latch remains in the state set until a further access alters it. If a latch is set, then further accesses to set it will have no effect and a reset access is required to change its state.

PULSE SIGNAL

The PULSE output is a control signal to the power supply board, used in generating the voltages necessary to program datapacks. It is controlled by an internal PULSE latch. When set, the PULSE signal is enabled and a 32 kHz square-wave signal of between 40-60 percent duty cycle is output to the PULSE output pin. When reset, the output is disabled and is low. The latch is automatically reset when the Organiser is off.

Caution should be used when accessing the PULSE latch, as damage could occur to the power supply if it is left enabled for too long. PULSE is only used by the Organiser during datapack programming, and in a strictly controlled loop using the READY signal as a feedback input. In this loop, PULSE is disabled as soon as the READY input goes high. This is discussed further in the power supply section.

ALARM SIGNAL

The ALARM signal is a direct output from the ALARM latch. It is used to drive the piezoelectric buzzer element mounted from the power supply board.

When the ALARM latch is set, the output signal goes high and the voltage is applied across the buzzer element. When reset, the signal is removed. ALARM may be left in either state, but the buzzer only produces sound at transitions between the two states. To produce a tone, the software must access the ALARM set and reset functions alternately to produce the frequency required.

The alarm signal is also used as an interlock in the power supply circuit, to allow datapack programming voltages to be applied to the packs. This is described further in the power supply section.

CLOCK AND KEYBOARD

The real-time clock and keyboard poll are both functions dealt with by the control IC. Although at first sight they are completely independent functions, they are linked together in the control IC since the keyboard poll outputs K1-K7 are part of the clock divider chain. For this reason they are described together in this section.

DIVIDER CHAIN

The clock divider chain is implemented in the control IC as a 27 bit binary counter split into two stages:

Stage 1 is a 15 bit free-running binary counter clocked by the 32768 Hz oscillator input. Each cycle of the clock increments the counter, and when all bits are high the next cycle resets them all to low. In other terms, each bit of the counter alternates high and low at a frequency of one half the previous bit. Hence the last bit, bit 15, oscillates at a frequency of 1 Hz.

Three of the bits of this stage may appear at output pins:

- Bit 0 (32768 Hz) is gated with the PULSE latch, and when PULSE is enabled appears at the output pin.
- Bit 5 (1024 Hz) is gated with the processor RES_B output, and when the processor is active (RES_B high), appears at the OSC output.
- Bit 15 (1 Hz) is gated with the NMI latch, and when enabled appears at the NMI output to interrupt the processor every second. When NMI is disabled, the 1 Hz signal is gated internally to clock the stage 2 counter.

Stage 2 is a 12 bit binary counter which may be clocked from one of two sources: either from the 1 Hz signal if NMI is disabled, or by a processor access to the COUNTER CLOCK area of its memory-map. In addition this stage may be reset by an access to the COUNTER RESET function. Eight of the twelve bits of this stage appear at the output pins. Bits 1-7 appear as the keyboard poll outputs K1-K7 respectively. Since they are open-drain outputs they are pulled low when the counter bits are low, and float when the counter bits are high. Bit 12 appears as the ACOUT signal, and when high internally sets the ON/OFF latch to start a switch-on sequence.

With the two stages linked together (i.e. with NMI disabled), the last bit (ACOUT) would have a cycle time of 68 mins 16 secs if left as a free-running counter. In practice the counter is never left to free run, and if started from a reset condition is interrupted after half a cycle (34 mins 8 sec) when it switches the Organiser on.

KEEPING TIME

The date and time are kept and updated by the processor in its internal RAM. When the Organiser is on, it is normally receiving an NMI interrupt every second, and so can update the time on a second by second basis. Clearly when the Organiser is off this cannot be done, and in this case the stage 2 counter is used instead to keep track of elapsed time since the Organiser was switched off.

To explain this process, imagine that the clock is set exactly with the processor running and receiving NMI interrupts every second. The time is incremented immediately following each interrupt. When the Organiser switches off it follows the following sequence:

1. Wait for NMI and increment clock
2. Access COUNTER RESET address to reset the stage 2 counter
3. Access SWITCH OFF address. This automatically disables the NMI output and switches the 1 Hz signal to start clocking the stage 2 counter.

The next and subsequent 1 Hz cycles will increment the stage 2 counter, and this will continue for 34 mins 8 secs until the last bit (ACOUT) is set high. This starts the switch-on sequence to restart the processor. When running, the processor enables the NMI latch to start updating the clock directly every second. It also reads the state of the ACOUT signal, and because it is high it knows the clock is 34 minutes 8 seconds slow, and adds this to its time registers. Hence the time and date are accurate again and being updated every second.

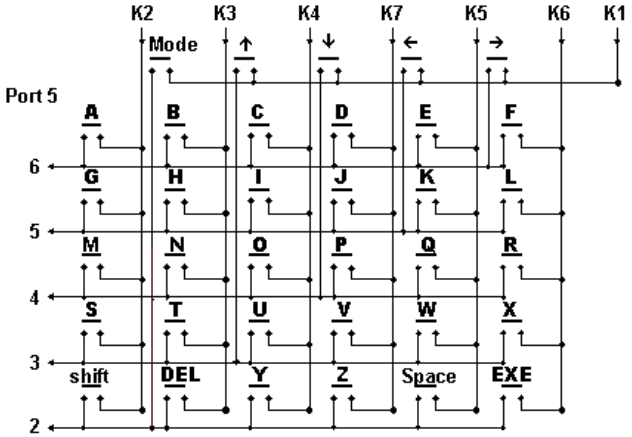
This explains the general mechanism of keeping time when the processor is off, using the stage 2 counter. A few other details need clarifying to explain the system fully:

1. On switch on, the test on the ACOUT signal determines the reason for the processor to be started. If ACOUT is high then a counter timeout has occurred as indicated above. In this case the processor will update its time registers as described and immediately switch off again. When left off, the Organiser keeps time by automatically switching on every 34 min 8 sec, updating the time and switching off again.
2. If ACOUT is low when the processor starts, then a counter timeout is not responsible (i.e. the AC key or the external AC input must have been activated). In this case the same startup procedure is followed, but the processor does not immediately know how much elapsed time to add to bring its clock registers up to date (i.e. how long since the Organiser was switched off). To find out, it repeatedly accesses the COUNTER CLOCK address until the ACOUT signal goes high. The number of clock cycles required effectively gives the number of seconds until the next counter timeout was due. This can be subtracted from 34 min 8 sec to give the elapsed time since switch off, and this time is added to the time registers.
3. If the Organiser is about to switch off and an alarm has been set within the next 34 min 8 sec, it can pre-load the stage 2 counter instead of resetting it just before switch-off. To do this it first RESETs the counter and then clocks it using the COUNTER CLOCK address. Each clock will reduce the time to the next counter timeout by one second.
4. The stage 2 counter is normally used to keep track of elapsed time when the Organiser is off, but the same mechanism can also be used when the processor is running. This is sometimes done when running time-critical code where NMI interrupts would be unacceptable.
5. Two adjustments are required to make the descriptions above fully accurate. Firstly, in any sequence where time-keeping is switched from direct NMI interrupt to stage 2 counter and back again, one second is gained and must be adjusted for in the software. This is a result of the hardware mechanism used to switch the 1 Hz signal between NMI and the counter. Secondly, during the switch-on sequence following a counter timeout (ACOUT high), an extra clock cycle to the counter may have occurred between the initiation cycle and the time that the processor switches to NMI interrupt. To detect if this has happened, the processor clocks the counter through (as after an AC press) until ACOUT switches

THE KEYBOARD

The AC key at the keyboard top left is a special case since it is used to switch the Organiser on. As such it is the only key on the keyboard whose function cannot be totally software defined. The AC key switches the AC signal on the board, and is input both to the control IC AC input and to the processor port 5 bit 7. It is normally low, and is pulled high on pressing the key. Pressing AC when the Organiser is off will set the ON/OFF latch in the control IC and start a switch-on sequence. When the processor is running, it polls this key by reading port 5 bit 7 (1=AC pressed, 0=AC not pressed). The external AC input from the Organiser top slot is in parallel with the AC key. This is used to switch the Organiser on from an external input, but it is disabled whenever the ON/OFF latch is set and so cannot be polled.

The other 35 keys on the keyboard are arranged as a 7 by 5 switch matrix. They are polled using the K1-K7 outputs from the control IC and the KBD1-KBD5 inputs to the processor port 5. The inputs are normally high, and are pulled low when a key is pressed and the relevant output is set low. The keys are arranged in the following way:



To poll the matrix, the processor first resets the stage 2 counter. All outputs K1-K7 will now be pulled low and all rows of the matrix accessed simultaneously; i.e. if any key is pressed then one of the port 5 inputs will be pulled low. Conversely, if all inputs are high then no keys are pressed and no further polling is required. If a key press is detected at this stage then the processor polls each row of keys in turn to isolate which key is responsible.

To do this it accesses the COUNTER CLOCK address until the K7 output is low but K1-K6 are all floating. The first row of the matrix above are now accessed, and depression of the D,J,P,V or S keys is detected if a low is present at the corresponding bit of port 5. To poll the next row, more COUNTER CLOCK accesses are required until the K6 output is low with K7 and K1-K5 floating. This process is repeated 7 times until all rows have been read.

Because this process uses the stage 2 counter, the keyboard can only be polled when the NMI latch is set to directly interrupt the processor.

LCD DISPLAY

The following paragraphs describes the operation of the two-line models:

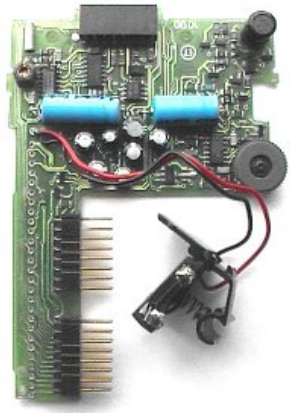
The Organiser display is 32 characters arranged in two lines of 16 characters each. In this configuration the HD44780 display driver provides the 16 common lines and the display is driven with a 1/16 duty cycle. The two display lines are each 7 dots tall plus a separate cursor line. Each character is 5 dots wide and so 80 segment lines are required. The first 40 of these (left half of the display) are provided by the HD44780, and the rest by the HD44100 slave driver.

The display on the LZ and LZ64 uses the same LCD drivers as on the standard Organiser but is arranged as 4 lines by 20 characters and the HD44780 was replaced by a customised HD66780 chip to allow for a customized character set.

The display is accessed by the processor in the memory area $0180-$01BF as described above. The two registers are selected by the A0 address line, so even addresses in this range access the Instruction Register, odd ones the Data Register. The 8 bit mode is used to transfer data to the processor, and this must be selected when the drivers are initialised.

The display drivers and LCD plate are powered by the VCC2 and V_LCD rails from the power supply board. These are switched off whenever the Organiser is off, and so the LCD must be initialised at each processor startup. The intermediate voltages required are provided by the resistor chain R1-R5. Contrast adjustment is controlled by the thumbwheel on the power supply board, by adjusting the V_LCD voltage between limits of +0.6 and -3 volts. Power required is typically 2 milliamp from VCC2 and 0.5 milliamp from V_LCD.

POWER SUPPLY BOARD



GENERAL

The Power Supply Board controls power regulation and distribution to the system, and also carries signals from the System board to the three external device connectors and buzzer element. Connections through to the slots are dealt with in sections Interface Slots and External Interfacing; this section concentrates on the power supply functions.

The board has been designed to operate under a wide range of conditions. Supply voltage from the battery or external source may vary between 5.5 and 11 volts. Regulated supplies are required at current levels between 30 microamp and 170 milliamp in various operating states.

POWER SUPPLY REQUIREMENTS

The power supply provides five supply rails to various parts of the system:

- The VCC1 rail supplies the main System board circuit, and is required at all times. When the Organiser is off this rail supports the real-time clock and system board RAM, and a rail voltage of 3-5 volts is required with a load of typically 30 microamp. When the Organiser is on, this rail must be regulated to 5 volt +/- 5 percent under loads up to 25 mA.
- The VCC2 rail supplies the system board LCD drivers. It is not required when the Organiser is off, and when on should be regulated to 5 volts +/- 5 percent with a load of typically 1 ma
- The V_LCD rail is the negative voltage rail for the System board LCD. It is not required when the Organiser is off, and when on should be adjustable by the thumbwheel to provide a V_LCD voltage of between +1.2 and -3 volts to set display contrast, with a load of typically 100-200 microamp.
- The SVCC rail supplies the external slot devices. It is only required when the slots need to be active, and so is switched under software control. When on it should be regulated to 5 volts +/- 5 percent with variable loads up to 170 mA.
- The SVPP rail is a secondary rail to slots 1 and 2 of the Organiser. It is only required when the slots need to be active, and is switched with SVCC. When on it is normally at a voltage of 4.5 volts, but when programming datapacks can be raised to 21 volts +/- 2 percent in a pulsed mode.

The power supply provides and switches these rails under control of the system board, from a battery or external supply with a voltage of between 5.5 and 11 volts.

The main operating states are defined by the ON_B and PACON_B signals from the system board. ON_B is high when the Organiser is off, and is pulled low by the control IC to switch the system on. PACON_B is controlled directly by software from the processor port 6, and is pulled low to switch the slot rails on. The main operating states are summarised in the following table:

	ON_B	PACON_B	VCC1	VCC2	V_LCD	SVCC	SVPP
1	high	high	3-5	off	off	off	off
2	low	high	5	5	-3-+1	off	off
3	low	low	5	5	-3-+1	5	4.5
4	low	low	5	5	-3-+1	5	21

In state 1, only the VCC1 rail is active, and total system load on the supply rail is typically 40 microamp, made up of 30 microamp load from the system board and 10 microamp quiescent current in the power supply (i.e. power required by the power supply itself in order to perform its functions.).

In state 2, VCC1,VCC2 and V_LCD are active with VCC1 and VCC2 fully regulated. In this state, total system power consumption is dependent on the processor operating mode. If the processor is fully active and running code, overall power consumption is typically 25 ma When the processor is in sleep mode, total power consumption is reduced to typically 4 ma

In state 3, all rails are active with SVCC regulated to 5 volts, and SVPP at 4.5 volts. Power consumption is dependent on the external devices plugged in, but without any devices present it is typically 25 ma internally since the processor is normally fully active in this state. The power levels of each device type are listed in the relevant sections of the manual.

In state 4, the SVPP rail is raised to 21 volts. This is a special case and only occurs when programming datapacks, in a pulsed mode controlled by the PULSE,READY,ALARM and OE_B signals.

SUPPLY RAILS

VB is the main system supply rail from which all regulated supplies can be derived. This is available at the top slot VB pin, and hence power input to this pin can supply the whole system. This rail should be at a voltage between 5.5 and 11 volts for proper system operation, when supplying up to 200 ma peak system loads.

The battery supplies the VB rail via the forward diode D1. Where loss of voltage across this diode would affect system performance, a direct connection to the relevant regulators is also made (TR6 and TR10).

The PP3 battery should be of a type with a low internal resistance so that it can supply peak system loads of 200 milliamp. Two types are commonly used:

- The long-life alkaline type is not rechargeable and has a typical capacity of 400-500 mAh (milli-amp hours).
- The nickel-cadmium type is rechargeable and has a typical capacity of 100 mAh between charges.

For both types, a new battery will give an off-load voltage of up to 10.5 volts, decreasing through its life to the minimum system supply voltage of 5.5 volts.

STANDBY REGULATOR

The Standby regulator supplies power to the VCC1 rail when the main system regulators are shut down (i.e. with ON_B high). In this mode the System board drains typically 30 microamp to keep the real-time clock running and to retain data. The VCC1 voltage in this state should be between 3 and 5 volts, and current consumption reduces with reducing voltage.

The standby regulator is formed from TR1,TR2,TR3,R2,R3, and R4. TR3 is a pass transistor supplying current to VCC1 from the VB rail. TR1 and TR3 are configured as a darlington pair, with base current supplied from R2. Without any feedback this circuit would hold the VCC1 voltage at 1.2 volts (two VB-E drops) below the VB rail. Feedback is implemented through TR2, which switches on with approx. 5 volts on the base of TR3. TR2 robs base current from TR1 and the circuit settles in this state, holding VCC1 at nominally 4.5 volts.

The circuit is not a good regulator since it relies on the TR2 VB-E drop as its feedback reference. Furthermore, its current capability is dependent on the VB voltage, since this defines the base current available through R2. When VB is lower than 6 volts, the VCC1 standby voltage may fall below its nominal value to compensate. These effects are not large enough to take it outside the 3-5 volt operating range.

The circuit has a low quiescent current, so that it is itself draining the minimum of power from the supply. Under worst case conditions, with VB at 11 volts, the current drawn by the regulator would be approx. 10 microamp, split equally between the R3,R4 divider chain and collector current through TR2.

The C3 capacitor decouples VCC1, and when the supply is removed supports the VCC1 rail until the supply is reconnected. The capacitor will hold VCC1 above 3 volts for 2-3 minutes in this case, but only if the circuit is in standby mode during this time (i.e. Organiser off).

VCC1/VCC2 REGULATOR

To switch the system into active mode, the control IC on the System board pulls the ON_B signal low. When this happens, TR15 is switched on through its base resistor R23, and the VCC2 rail is powered on to a voltage of VCC1 minus 0.2 volts (TR15 VC-E drop). VCC2 powers the main regulator op-amp (OPA1) and the regulator voltage reference (RD1), and the main VCC1/VCC2 regulator is switched on. In turn, this now supplies the VCC1 rail and regulates it to 5 volts +/- 5 percent. While the main regulator is being activated, the VCC1 and VCC2 rails are supported by C3 and the standby regulator. In active mode, the current demand from the main regulator is built up from three elements:

1. The System board requires up to 25 milliamp from VCC1 when the processor is fully active.
2. The LCD drivers on the system board take approx. 1 milliamp from the VCC2 rail (supplied from VCC1 via TR15).
3. The regulator op-amp and reference source take approx. 1 milliamp from the VCC2 rail. (this is effectively the power supply quiescent current when in active mode).

The LM324 op-amp is a quad device having four individual op-amp circuits. These are used in various ways in the circuit. The voltage reference diode (type 9491BJ) is a temperature-compensated bandgap device with a voltage of 1.22 volts +/- 2 percent. This voltage is used in all regulators as a feedback reference.

The VCC1/VCC2 regulator is formed from R5,R6,R7,R18,TR5,TR6,TR4 and the op-amp pins 8,9,10. TR5 and TR6 are PNP pass transistors arranged in tandem, to supply the VCC1 rail from either VB or directly from VBAT. PNP transistors are used to minimise the regulator drop-out voltage (i.e. a VB-E drop is not lost as with NPN pass elements). The tandem arrangement is so that the drop across D1 is not lost when the battery is the power source. The regulator will operate from whichever is the higher of VB and VBAT. The divider chain R5,R18,R6 monitors the VCC1 voltage, and is arranged so that 1.22 volts is present at the op-amp input when VCC1 is 5 volts. The op-amp compares this to the reference voltage, and its output controls the pass transistors via TR4. R7 limits the pass transistor base current. If VCC1 is lower than 5 volts, then TR4 is switched on to draw more current through the pass transistor. Conversely, if VCC1 is higher than 5 volts, TR4 is switched off. In practice, the circuit settles at a VCC1 of 5 volts supplying a current equal to the demand.

The regulator functions down to a supply voltage of approx. 5.3 volts. Below this the VCC1 rail will follow the supply voltage at approx. 0.3 volt below it. In this case TR4 will be switched hard on, since the op-amp input from the divider chain will be below the reference voltage. At a point where VCC1 is approx. 2 percent below its nominal regulated voltage, the op-amp section pins 12,13,14 will switch to set the LOWBAT signal high to the System board.

SVCC REGULATOR

The SVCC rail is used to supply external devices in the three slots. It is switched on and off by the PACON_B signal from the system board, and when on, regulates to 5 volts +/- 5 percent. It also supplies the SVPP rail through a forward diode. The load on the regulator is dependent on devices plugged in. Maximum load is 170 ma, set essentially by the rating of the pass transistors, which is 1 watt. For a worst case supply voltage of 11 volts and an output voltage of 5 volts, this rating is met when 170 ma is drawn.

The regulator design is similar to the VCC1/VCC2 regulator above. TR9 and TR10 are PNP pass transistors from the VB and VBAT rails. R8 and R9 form the feedback divider chain to the op-amp pin 6, and are compared to the reference voltage. TR7 controls the pass transistors, with base current being limited by R10.

The PACON_B signal switches the regulator on and off. When PACON_B is high, TR8 is switched off and base current from the regulator is blocked. Note that in this state TR7 will be hard on since SVCC is below 5 volts, but nevertheless no base current is available to the pass transistors. When PACON_B is low, TR8 is switched on and the regulator can operate. If there is no load, (i.e. no devices are plugged in) then the base current through the pass transistors will be minimal, and so the quiescent current of this regulator is negligible.

V_LCD

V_LCD is the negative rail used by the LCD drivers on the System board. It is adjusted by the thumbwheel variable resistor to set contrast on the display. Current consumption is 100-300 microamp dependent on the voltage set, between +1.2 and -3 volts.

V_LCD is generated directly from the OSC signal from the system board. When the Organiser is on, this gives a 1024 Hz square wave signal from the control IC. R20 and the VR1 variable resistor limit the current, and then drive the series capacitor C8. When OSC is high, charge is pushed from C8 and is drained to ground through the forward diode of D3. When OSC goes low, charge is pulled to C8 making the C8/D3 track negative. The reverse diode of D3 transfers charge from the storage capacitor C1. With no load, the voltage at C1 would be the inverse of the OSC voltage less the D3 diode drop. Note that the maximum positive voltage on the V_LCD rail will be +1.2 volts, since it is tied to ground through the two diode drops of D3.

This form of voltage inverter is not precise and can only supply a low current. In this application, however, the circuit is adjusted by the operator, and the voltage set is not important as long as the contrast range is sufficient.

VOLTAGE PUMP

The voltage pump is used by the system when programming datapacks. Two stages are required to transfer the voltage required to the SVCC rail. Firstly the pump is used to charge up a storage capacitor to 35 volts. This is then regulated to 21 volts and switched to the SVCC rail under program control. The storage capacitor contains enough charge to program an EPROM byte. This sequence is repeated for each byte to be programmed.

The voltage pump is formed from TR11,TR12,R16,R17,L1,D5,ZN1 and C6. The inductor L1 is the energy transfer element and C6 is the charge storage capacitor. Pumping is controlled by the PULSE signal from the System board, and the zener diode ZN1 provides feedback to the processor by setting the READY line high when C6 is charged sufficiently for the next byte programming operation.

In steady-state conditions (before pump operation), the PULSE line will be low, with the darlington pair TR11 and TR12 switched off. C6 will be charged to the VB voltage, less a diode drop across D5. The READY line will be low since C6 is well below the 35 volt switching voltage. Note also that TR14 should be off (i.e. the 21 volt regulator is not switched on) and so their is no load on the C6 rail.

To operate the pump, the processor enables the PULSE signal from the control IC. PULSE then oscillates at 32768 Hz frequency. When PULSE is high, TR11 and TR12 are switched on, with TR12 in saturation bringing its collector voltage to 0.3 volts. Current is now drawn through the inductor d to the V=LdI/dt law, (where V is the voltage across the inductor VB-Vcollector, L is the 1 mH inductance and t is the time from Pulse being set high.). After 15 microsec, PULSE goes low and TR12 is switched off. Current continues to flow through the inductor, raising the voltage at the TR12 collector until it can flow through the forward diode D5 to the storage capacitor C6. The inductor now discharges into the capacitor under its V=L dI/DT law (where V is now the reverse voltage between the TR12 collector and VB, set by the current C6 voltage plus the D5 diode drop). This process is repeated on every cycle of PULSE, and the C6 voltage steadily rises. When C6 reaches the zener voltage of 33 volts, it starts to pull the READY line up, and this will be seen as high by the processor when a further 2 volts are added. (During this sequence the ready line is pulled low by the 47k resistor from the ACOUT signal on the System board, so leakage through the zener is drained). On reading ready high, the processor should disable PULSE and can proceed with programming a datapack byte.

SVPP REGULATOR

The SVPP regulator is used to regulate the C6 storage capacitor voltage to 21 volts and to switch this onto the SVPP rail to the slots. It is only used during datapack programming and in conjunction with the voltage pump above.

The regulator is formed from TR13,TR14,R11,R12,R13,R14,D4 and the op-amp section pins 1,2 and 3. In operation it is similar to the VCC1/VCC2 regulator, with the divider chain R11/R12 forming the feedback loop to set the op-amp input at 1.22 volts when the SVPP rail is at 21 volts. The regulator is switched on and off by a combination of the PACON_B, OE_B and ALARM signals from the system board. To switch on, PACON_B must be low (i.e. SVCC on), and both ALARM and OE_B must be high. The software ensures that this condition only arises when the SVPP regulator is required, and this imposes conditions on the use of the buzzer

when the slots are switched on. If PACON_B is high, base current from the TR14 pass transistor is blocked by TR8. If either ALARM or OE_B are low, then the base of TR13 is pulled low to switch it off irrespective of the state of the op-amp output at pin1. (When the Organiser is on, the op-amp output is normally high since SVPP is normally lower than the 21 volt regulating voltage.)

When the regulator is switched on, it draws current from the C6 storage capacitor, and the voltage at C6 falls as it loses charge. The capacitor has sufficient charge to maintain 21 volts at the regulator output for 50 msec under a SVPP load of 50 milliamp. The software must limit switch-on time to this maximum, switch off the regulator and re-pump the capacitor if it requires any further datapack bytes to be programmed. The SVPP rail can be in one of three states:

- 0V if the PACON_B is high and the slots are switched off
- 4.5 volts when PACON_B is low, fed by the forward diode D2 from the SVCC rail
- 21 volts when the SVPP regulator is switched on.

REMOTE SWITCH ON

The AC_B signal from the top slot can be used to switch the Organiser on. The AC_B line is pulled high to VCC1 by the internal 47k resistor R24. If an external device pulls this input low, then TR16 is switched on, and the AC signal to the system board goes high to switch the system on. Note that this will only happen if the ON_B signal to the emitter of TR16 is high (i.e. Organiser off). As soon as ON_B goes low, the AC_B input is disconnected an has no effect.

BUZZER

The piezoelectric buzzer element is driven from the ALARM signal, through the 1k resistor R15.

DISPLAY

related information: Appendix A

HARDWARE

The display used in the Organiser II models CM, XP and LA is a Hitachi LR039 Liquid Crystal Display (LCD) driven by a Hitachi HD44780 LCD controller.

It consists of two lines of 16 characters. Each character is defined by a 5 x 8 dot matrix with the 8th line normally left blank as it is used for the single line cursor.

The character generator in the LCD controller contains 192 displayable characters (ASCII values 32 to 256). There are also 8 user definable characters which are repeated twice in the character set ASCII values 0 to 7 and 8 to 15. These are called UDG 0-7.

The display on the LZ and LZ64 uses the same LCD drivers as on the standard Organiser but is arranged as 4 lines by 20 characters. The HD44780 was replaced by a customised 66780 chip to allow for foreign characters. The characters from 0 to 127 remain the same except for character 92 which is changed to a "\" character. Most characters from 128 to 255 have changed but some are left for compatibility, e.g.. character 254 is left as a space to use as a blank "non-space" character.

SOFTWARE

2-LINE COMPATIBILITY ON 4 LINE MODELS

The LZ models maintain compatibility with 2-line Organisers by emulating the 2 line screen. Thus, there are two modes of operation: 4-line mode and 2-line compatibility mode. To simplify the writing of 2-line/4-line applications, variables are provided for the address of the screen, the mode, the number of lines and the width of the screen which are valid in either mode.

The system service DP$MSET is provided to switch between modes. Note also that existing bootable applications are automatically put into 2-line mode if they print to the screen.

In 2-line compatibility mode all existing applications should work exactly as though they are running on a 2-line Organiser. The same screen buffer (DPT_TLIN) is used when in this mode.

REGISTERS AND VARIABLE USAGE

The LCD controller contains a control register and a data register. These are accessed via the following addresses:

$0180	SCA_LCDCONTROL	The LCD control register
$0181	SCA_LCDDATA	The LCD data register

The following variables are used by the display system services:

dpt_tlin	16 byte buffer containing contents of top line of the display (not used in 4-line mode)
dpt_blin	16 byte buffer containing contents of bottom line of the display (not used in 4-line mode)
dpb_cpos	Current cursor position (0-31, LZ: 0-79)
dpb_cust	Cursor state (as passed in DP$STAT)
dpb_vlin	Used in view
dpb_vsiz	Used in view
dpb_vdir	Used in view
dpw_sped	Horizontal scrolling rate (in 50ms intervals). Delay before scrolling is four times dpw_sped
dpw_dely	Vertical scrolling rate.
dpw_redy	Decremented every 50ms. Used for timing. See DP$WRDY.
dpa_vadd	Used in view.
dpb_spos	Saved cursor position. Used by DP$SAVE and DP$REST.
dpb_scus	Saved cursor state. Used by DP$SAVE and DP$REST.

dpt_save	32 byte buffer containing saved display text, used by DP$SAVE and DP$REST on CM/XPs..

The following variables are only used by 4-line models. Only dpb_mode was also assigned (and zeroed) on 2-line machines. They can be read at any time but should not be written to directly (DP$MSET or DP$CSET must be used).

dpb_mode	shows the display mode that is currently in operation (0 = 2-line, 1 = 4-line).
dpa_scrn	contains the address of the current screen buffer (in 2-line mode: DPT_TLIN, in 4-line mode: DPT_4LIN)
dpb_nlin	contains the number of screen lines in the current display mode.
dpt_wide	contains the screen width in the current display mode.
dpb_clok	CLOK stores the position of the left most character of the 6 character UDG clock. If bit 7 is set the clock will not be updated (i.e. it isoff).
dpb_cred	"clock ready" flag, used to time the 1/2 second flashing of the colon. Counts down from 9 to 0 and the top bit is toggled for showing the ":" instead of an underline character.

On the LZ, DP$SAVE and DP$REST use DPT_4SAV, the 4-line save buffer, even in 2-line mode, instead of the previous DPT_SAVE which no longer exists.

THE UDG CLOCK (LZ only)

The UDG clock uses UDGs 3-7 to display the digits and UDG 1 for the am/pm indicator if in 12 hour mode. The clock is always right justified in a field of 6 characters.

The system service DP$CSET, can print the clock anywhere on the screen and will initalise the updating of the clock every 1/2 second. DP$CSET can also stop the clock by preventing the updating but the UDGs will remain on the screen until they are overwritten by printing.

The clock is updated in the keyboard interrupt routine by calling DP$CPRN. Every 1/2 second UDG 5 is re-defined to achieve the flashing colon and every time the minutes change, all UDGs used for the clock are redefined

Occasionally, the clock will stop flashing, e.g.. when printing. This is because the keyboard interrupts have been disabled momentarily. The clock will not, of course, lose any time since it always gets the time from the system clock.

EXAMPLE

The following machine code program stores a pound sign in UDG 0 (ASCII characters 0 and 8). The data for the pound sign is read from address DATA.

Note that on LZ machines this task could also be accomplished by using DP$UDG, but this code will still work.

```
SAVUDG:
; Set LCD address to $40 the address of the first UDG.
        LDA     A,#$40          ;A is LCD address
; Wait for busy low.
1$:     TST     SCA_LCDCONTROL
        BMI     1$
; Store address
        STA     A,SCA_LCDCONTROL
; Now store data
        LDX     #DATA
        LDA     B,#8
2$:     LDA     A,0,X
; Wait for busy low.
3$:     TST     SCA_LCDCONTROL
        BMI     3$
; Store data. (LCD address auto increments)
        STA     A,SCA_LCDDATA
        INX
        DEC     B
        BNE     2$
        RTS

DATA:
        FCB     6,9,9,12,8,24,31,0      ;Pattern for pound sign
```

SYSTEM SERVICES

This section describes the operating system calls for writing to the LCD display.

DP$STAT

Displays or removes block or line cursor at given position.

DP$VIEW

Displays a given string at a given line. If the string doesn't fit, it starts to scroll to the right wrapping around with two spaces in between. Pressing the left and right arrow keys alters the scrolling action (this feature may be disabled). Any other key press terminates.

DP$PRNT

Prints a given number of a single character.

DP$EMIT

Outputs a single character.

Control characters ASCII 0 to 7 display the 8 user definable graphics characters.

Certain of the ASCII control characters are intercepted by this routine to control the display as follows:

DEC	HEX	DESCRIPTION
8	08	Moves cursor left 1 character.
9	09	Moves cursor to the nearest position modulo 8.
10	0A	Goes on to the next line.
11	0B	Moves the cursor to the top left of display.
12	0C	Clears both lines of the display.
13	0D	Returns the cursor to the beginning of line.
14	0E	Clears the 1st line. Cursor at top left.
15	0F	Clears the 2nd line.
16	10	Rings the bell.
17	11	Refreshes display.
18	12	Refreshes 1st line.
19	13	Refreshes 2nd line.
20	14	Refresh 3rd line (LZ).
21	15	refresh 4th line (LZ).
22	16	Clears 3rd line (LZ).
23	17	Clears 4th line (LZ).
24	18	Prints high dotted line (LZ).
25	19	Print low dotted line (LZ).
26	20	Clear to end of line (LZ).

On LZs, to maintain compatibility, the system service DP$EMIT will translate any characters from $80 to $A0 inclusive into a space character if the machine is running in "2-line compatibility mode".

DP$SAVE

Saves the current screen state including the cursor position.

Note that this routine should not be called unless alarms are disabled as the alarm software calls this routine as well. This is explained in the diary section. Also the pack sizing routine (PK$SETP) uses these calls when displaying "SIZING PACK B" etc.

DP$REST

Restores a previously saved copy of the screen

If DP$REST is called without previously calling DP$SAVE then anything can and probably will happen.

DP$WRDY

Waits until the value in dpw_redy has been decremented to 0.

Each keyboard interrupt causes dpw_redy to be decremented by 1 and as such provides a handy 50 millisecond timer. However if interrupts are switched off this routine will still work as it recognises that interrupts are off and decrements dpw_redy once every 50 milliseconds.

Note that this routine behaves as a wait on a timed semaphore. It is used extensively by the LCD driver software to provide scrolling delays.

DP$MSET (LZ only)

Sets the Operating System into 2 or 4 line mode. The following is always carried out:

1. If the top bit of the A register is clear, the UDG CLOCK is switched off.
2. The variable DPB_MCHK is cleared.
3. END if the mode requested is the same as the current mode.
4. The screen is initalised for the mode selected:
    - 2-line: clear 16 character by 2 line window with border
    - 4-line: clear entire 4-line screen.
5. The variables DPA_SCRN, DPB_NLIN, DPB_WIDE are initialised.

DP$PVEW (LZ only)

A partial view routine. Works exactly like DP$VIEW but at the current cursor position. Any characters before the cursor will not scroll but the string will be displayed scrolling on the remainder of the line. This routine is used, for example, in CALC on the LZ.

DP$UDG (LZ only)

Defines or reads a UDG. The operating modes are:

- Define an UDG with 8 bytes (one per pixel line).
- Define an UDG with 5 bytes (one per pixel row).
- Store an UDG to a buffer in 8 byte mode.

Note that interrupts are disabled in DP$UDG.

DP$CSET (LZ only)

Sets the clock-display status. Parameters specify if and where on the screen the clock will appear. The clock is printed when DP$CSET is called and is updated in the keyboard interrupt. The interrupt just re-defines the UDGs and does not write to the screen.

Note that any UDGs previously defined will be overwritten.

DP$CPRN (LZ only)

Redefines the UDGs used for the clock as the current system time.

KEYBOARD

GENERAL

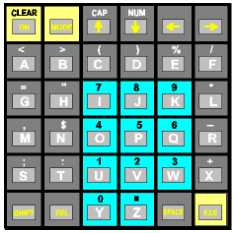
The keyboard consists of 36 keys which are polled on interrupt at a variable interval, initially set to 50ms. The keys auto-repeat at a variable rate and key presses are stored in a 16 character type-ahead buffer. The keyboard lookup table, the polling, the key translation and the shift status can all be altered through the use of the keyboard vectors and variables. The SHIFT key can be disabled and the keys for CAP and NUM can be changed.

The keyboard interrupts also control alarm checking and display timing, and increment a frame counter.

While testing for a key, the datapacks or the machine itself may switch off and a low battery test is made.

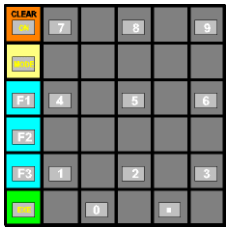
KEYBOARD VARIANTS

Variants on the standard 36-key keyboard were also produced for particular customer requirements. These are either all-numeric or a special alphanumeric layout called "Alpha-POS". The Alpha-POS layout "inverts" the Alpha and Numeric keys by re-labeling the template around the keys, and the keycaps.



standard keyboard

Alpha-POS keyboard



numeric keyboard

OPERATING THE KEYBOARD

The normal keyboard consists of a 6 by 6 matrix of keys, as shown above.

The letter keys normally produce upper-case letters and return the corresponding ASCII value. By holding down the SHIFT key and pressing one of the letters you can access the symbols and numbers marked above the keys.

To select lower-case letters, hold down the SHIFT key and press the CAP key. Repeat this to return the keyboard to upper-case mode.

Pressing the SHIFT and NUM keys, puts you permanently in 'shift' mode so that the function of the SHIFT key is now reversed. The functions of SHIFT NUM, SHIFT CAP and SHIFT DEL, however, are not affected (DEL deletes characters to the left and SHIFT DEL deletes characters from the right).

The top row of keys and some on the bottom row are special. These keys return the following values:

KEY	VALUE
ON/CLEAR	1

MODE	2
UP	3
DOWN	4
LEFT	5
RIGHT	6
SHIFT DEL	7
DEL	8
EXE	13

The SHIFT key, the CAP key and the NUM key do not return values but immediately carry out their function.

The keyboard on the LZ has been extended to allow SHIFT-EXE, SHIFT-SPACE and SHIFT-RIGHT-ARROW. These SHIFT-KEY functions can be disabled by setting bit 7 of KBB_SPEC.

- Shift EXE
  To allow the use of SHIFT-EXE, bit 0 of KBB_SPEC must be set. When set, SHIFT-EXE will be returned as value 14.
- Shift Space
  To allow the use of SHIFT-SPACE, bit 1 of KBB_SPEC must be set. When set, SHIFT-SPACE will be returned as value 15.
- Shift Right-arrow
  SHIFT-RIGHT-ARROW is used to select foreign characters for the next key press only. For example, when the language is set to French, SHIFT-RIGHT arrow followed by 'a' produces the character 'á'.

KEYBOARD SCANNING

The ON/CLEAR key is polled independently of the others. The remaining 35 keys are polled on a 5 by 7 matrix.

THE ON/CLEAR KEY

Testing for the ON/CLEAR key is done by reading bit 7 of port 5 (address $15). If the key is pressed, bit 7 will be set, otherwise bit 7 is clear. Hence, the ON/CLEAR key can be tested directly and very quickly. The following routine waits for it to be pressed:

```
TESTKEY:
        LDA     A,POB_PORT5:    ;READ ADDRESS $15
        BPL     TESTKEY         ;BRANCH IF BIT 7 IS CLEAR
```

In some applications it is necessary to protect against 'key bounce' and it is recommended that a delay of approximately 50ms is used. The following routine waits for the ON/CLEAR key to be released and then pauses for 50ms:

```
DEBOUNCE:
        LDA     A,POB_PORT5:    ;READ ADDRESS $15
        BMI     DEBOUNCE        ;BRANCH IF BIT 7 IS SET
        LDX     #11600          ;FOR DELAY OF 50MS
1$:     DEX
        BNE     1$
```

Note that if keyboard interrupts are enabled while running the above procedure, holding down the ON/CLEAR key will fill up the keyboard buffer.

A system service, KB$BREK, is provided to test if the ON/CLEAR key is pressed or if an ON/CLEAR key is in the keyboard buffer.

THE KEY MATRIX

The key matrix consists of 7 columns, controlled by the SEMI-CUSTOM-CHIP 'COUNTER', and 5 rows which can be read as bits 2 to 6 of PORT 5 (the ON/CLEAR key uses bit 7). The layout is as follows (see also System Board - Keyboard):

input line	port 5 bits				
	KBD5	KBD4	KBD3	KBD2	KBD1
	6	5	4	3	2
K7	D	J	P	V	S
K6	F	L	R	X	EXE
K5	G	K	Q	W	DEL
K4	C	I	O	U	Z
K3	B	H	N	Y	Y
K2	A	G	M	S	SHIFT

K1	right arrow	left arrow	down arrow	up arrow	MODE

To control the COUNTER lines, there are two significant addresses:

1. SCA_COUNTERRESET (address $300) - sets all lines to zero.
2. SCA_COUNTERCLOCK (address $340) - increments COUNTER address by one.

To carry out these functions, simply read or write to the respective address. The following will set the contents of the counter to $3F, i.e. K1 to K6 high and K7 low:

```
        TST     SCA_COUNTERRESET        ;SET COUNTER TO ZERO
        LDA     A,#$3F
1$:     TST     SCA_COUNTERCLOCK        ;INCREMENT COUNTER
        DEC     A
        BNE     1$
```

When a key is pressed, a connection is made from one of K1 to K7 to one of the PORT 5 lines. The lines on PORT 5 are pulled high, so by setting one of K1 to K7 low, a specific key press can be detected. For example, if K7 only is low, the 'D' key can be detected by bit 6 of PORT 5 going low.

Polling the entire keyboard involves setting each of K1 to K7 low in turn, reading PORT 5, and decoding the key. By setting all 7 lines low simultaneously, a quick check for any key can be made. The following routine will test for any key press:

```
        TST     SCA_COUNTERRESET        ;SET K1 TO K7 LOW
        LDA     A,POB_PORT5:            ;READ PORT 5
        BMI     CLRKEY                  ;BRANCH IF ON/CLEAR KEY IS PRESSED
        AND     A,#$7C                  ;IGNORE BITS 7,1 AND 0
        CMP     A,#$7C                  ;CHECK IF ANY BIT IS LOW
        BNE     AKEY                    ;BRANCH IF KEY PRESS DETECTED
```

The method of keyboard scanning is explained in the next section.

KEY SCANNING

The keyboard is scanned as fast as possible using the routine pointed to by the ram vector BTA_POLL. For maximum efficiency, the following method is used:

1. Check if ON/CLEAR key is pressed and exit if it is.
2. Check quickly if any other key is pressed and exit if not.
3. Determine which key is pressed and check if the SHIFT key is also pressed.

Steps 1 and 2 are described above. Step 3 involves setting each of the SEMI-CUSTOM-CHIP COUNTER lines (K1 to K7) low in turn. This is done most efficiently by setting K7 low first then K6 down to K1:

K7	K6	K5	K4	K3	K2	K1	COUNTER VALUE
0	1	1	1	1	1	1	$3F
1	0	1	1	1	1	1	$5F
1	1	0	1	1	1	1	$6F
1	1	1	0	1	1	1	$77
1	1	1	1	0	1	1	$7B
1	1	1	1	1	0	1	$7D
1	1	1	1	1	1	0	$7E

The following code will set the values $3F,$5F... etc. on the COUNTER:

```
        TST     SCA_COUNTERRESET        ;READ ADDRESS $300 TO ZERO COUNTER
        LDA     B,#$40
        PSH     B
        DEC     B                       ;INITIAL VALUE $3F
        BRA     CLOCKB
NEXTCOL:
        PSH     B
CLOCKB:
        TST     SCA_COUNTERCLOCK        ;READ ADDRESS $340 TO INC COUNTER
        DEC     B                       ;DO IT B TIMES
        BNE     CLOCKB
        BSR     READPORT5               ;READ PORT 5 HERE AND DECODE KEY
        PUL     B
        LSR     B
        BNE     NEXTCOL                 ;REPEAT FOR COLUMNS K7 to K1
```

The SHIFT key is in column K2 so scanning of the keyboard must continue, even after one key has been found, to check if SHIFT is also pressed.

This method results in the following approximate keyboard scan times:

KEYBOARD STATE	TIME TAKEN TO SCAN KEYBOARD
no key pressed	0.3ms
on/clear key pressed	0.2ms
any other key pressed	1.9ms

Note that the time taken for the entire keyboard interrupt may be increased by one or more of the following:

1. The variable key click, 1ms by default
2. The 'buffer-full' beep which lasts 10ms.
3. Alarm checking and/or ringing which may last up to 2 mins (see Alarm Checking Interrupts).

KEYBOARD INTERRUPTS

The keyboard scanning routine is called at regular intervals from an interrupt generated by the processor's TIMER 1 counter. This is a 16 bit FREE RUNNING COUNTER (FRC) (address $09,$0A) incremented by the processor clock.

The interrupt is generated when the value in the FRC matches a value set up in the TIMER 1 OUTPUT COMPARE REGISTER 1 (address $0B,$0C) and is directed to the address contained in the ROM at address $FFF4. The ram vector BTA_OCI is then used to jump to the interrupt service routine, i.e.:

```
        LDX     BTA_OCI
        JMP     0,X
```

The code in the interrupt service routine handles the following:

1. Polling the keyboard using a routine at BTA_POLL
2. Translating any key found using a routine at BTA_TRAN
3. Producing a key 'click'
4. Checking for alarms
5. Incrementing a frame-counter
6. Decrementing a display-timer

INITIALISING

Keyboard interrupts are fully initialised on a cold start of the machine only, using the system service KB$INIT. When each interrupt occurs, the first task is to set up the FREE RUNNING COUNTER and OUTPUT COMPARE REGISTER for the next interrupt, i.e.:

```
        LDA     A,POB_TCSR1:    ;MUST BE READ SO THAT A SUBSEQUENT WRITE TO
        CLR     A               ; - OCR1 WILL CLEAR THE OUTPUT COMPARE FLAG
        CLR     B               ; - SEE ACCOMPANYING 6301 BOOK
        STD     POW_FRC:        ;RESET FREE RUNNING COUNTER TO ZERO
        LDD     KBW_TDEL:       ;GET VALUE FOR KEYBOARD INTERRUPT RATE
        STD     POW_OCR1:       ;SET OUTPUT COMPARE REGISTER
```

When the machine switches off, the value in TIMER CONTROL STATUS REGISTER 1 and the state of the INTERRUPT MASK are saved so that on a warm start they can be restored. Also, the keyboard buffer is flushed on a warm start using system service KB$FLSH.

POLLING

The routine to poll the keyboard is called through the ram vector BTA_POLL. The function of this routine is to scan the keyboard and return the key pressed in the A register and to set any flags required by the translating routine at ram vector BTA_TRAN. The following code can be used to poll the keyboard and will return the key pressed in the A register as a value between 0 and 36 (0 means no key):

```
KB_POLL:
        TST     SCA_COUNTERRESET        ;SET COUNTER TO ZERO
        AIM     #<$FF-KY_SHFT>,KBB_STAT:        ;NO SHIFT KEY YET
        LDA     B,POB_PORT5:            ;TEST FOR ON/CLEAR KEY
        BPL     NOTCLR                  ;BRANCH IF NOT PRESSED
        LDA     A,#36                   ;36 REPRESENTS THE ON/CLEAR KEY
        RTS

NOTCLR:
        CLR     A                       ;NO KEY YET
        STA     A,KBB_KNUM:             ;KEY NUMBER BECOMES 0
        PSH     A                       ;PSH 0
        BSR     LOOKKEY                 ;ANY KEY IS PRESSED? SETS Z IF NOT
        PUL     A                       ;DOES NOT AFFECT Z FLAG
```



```
        BEQ     ENDKYBD                 ;BRANCH IF NO KEY FOUND IN LOOKKEY
        LDA     B,#$40
        PSH     B
        DEC     B
        BRA     CLOCKB

NEXTCOL:
        PSH     B
CLOCKB:
        PSH     A
1$:     TST     SCA_COUNTERCLOCK        ;READ ADDRESS $340 TO INC COUNTER
        DEC     B                       ;DO IT B TIMES
        BNE     1$
        BSR     LOOKKEY                 ;SETS B TO ROW NUMBER,IF KEY FOUND
        PUL     A                       ;IN THIS COLUMN
        BEQ     NOPSH                   ;BRANCH IF NO KEY IN THIS COLUMN
        ADD     B,KBB_KNUM:             ;GOT A KEY BUT MUST CONTINUE,
        TBA                             ;TO CHECK IF SHIFT IS ALSO PRESSED
NOPSH:  LDA     B,KBB_KNUM:
        ADD     B,#5
        STA     B,KBB_KNUM:             ;POINT TO NEXT COLUMN
        PUL     B
        LSR     B                       ;SELECT NEXT COLUMN
        BNE     NEXTCOL
ENDKYBD:
        RTS

LOOKKEY:
                ;SETS B TO ROW NUMBER (1-5) OF KEY IN CURRENT COLUMN
                ;B WILL BE ZERO IF NO KEY IN THIS COLUMN (OR JUST SHIFT)
                ;SETS KY_SHFT FLAG IF SHIFT IS PRESSED
                ;SETS Z ON B
                ;PRESERVES X

        LDA     B,#5            ;CHECK 5 ROWS
        LDA     A,POB_PORT5:    ;READ PORT 5
        ASL     A               ;IGNORE BIT 7
NEXTROW:ASL     A               ;ROTATE KEY INTO CARRY,CLR IF KEY PRESSED
        BCC     GOTKEY          ;BRANCH IF KEY PRESS DETECTED
        DEC     B
        BNE     NEXTROW         ;CHECK NEXT ROW
        RTS                     ;RETURN WITH B=0 IF NO KEY FOUND

GOTKEY:                         ;B IS POSITION IN ROW
        TST     KBB_SHFK        ;TEST SHIFT ENABLE FLAG
        BNE     11$             ;EXIT IF SHIFT DISABLED, B IS NOT ZERO
        PSH     A
        LDA     A,KBB_KNUM:     ;GET CURRENT KEY NUMBER
        CMP     A,#25           ;IS IT 6TH COLUMN, CONTAINING SHIFT KEY?
        PUL     A
        BNE     11$             ;EXIT IF NOT, B IS NOT ZERO
        DEC     B               ;COMPARE B WITH ROW 1 (ROW WITH SHIFT KEY)
        BEQ     10$             ;BRANCH IF THIS IS THE SHIFT KEY (B IS 0)

        PSH     B               ;ELSE CHECK IF SHIFT ALSO PRESSED
1$:     ASL     A
        DEC     B
        BNE     1$
        PUL     B
        INC     B               ;RESTORE B TO REAL KEY PRESSED
        BCS     11$             ;BRANCH IF C SET (FROM ASL A)

10$:    OIM     #KY_SHFT,KBB_STAT:      ;SET SHIFT-PRESSED FLAG
        TST     B                       ;SET Z FLAG ON B
11$:    RTS
```

The auto-repeat of the keys is accomplished using KBB_PREV, KBB_DLAY, KBB_REPT and KBB_CNTR in the following manner:

```
        LDX     BTA_POLL                ;POLL KEYBOARD (SEE ABOVE)
```

```
        JSR     0,X                     ;RETURN KEY PRESS IN A
        CMP     A,KBB_PREV:             ;IS SAME KEY STILL PRESSED ?
        BEQ     1$                      ;BRANCH IF YES
        STA     A,KBB_PREV:             ;SAVE NEW KEY
        LDA     B,KBB_DLAY:             ;DELAY BEFORE AUTO-REPEATING BEGINS
        BRA     2$

1$:     LDA     B,KBB_REPT:             ;DELAY BETWEEN KEYS DURING REPEAT
        TST     KBB_CNTR                ;TEST DELAY COUNTER
        BEQ     2$                      ;BRANCH IF TIME TO RETURN KEY
        LDA     B,KBB_CNTR:
        DEC     B                       ;TO DECREMENT DELAY COUNTER
        CLR     A                       ;NO KEY UNTIL COUNTER IS ZERO

2$:     STA     B,KBB_CNTR:             ;SET DELAY COUNTER
        LDX     BTA_TRAN
        JSR     0,X                     ;TRANSLATE KEY IN A REGISTER

        TIM     #KY_CPNM,KBB_STAT:      ;IS IT CAP OR NUM.
        BNE     DOCLICK                 ;IF SO, JUST EMIT KEY CLICK

        TST     A                       ;TEST IF KEY PRESSED
        BEQ     END                     ;END OF INTERRUPT IF NOT

                ;INSERT KEY INTO BUFFER - SEE BELOW
DOCLICK:
                ;EMIT KEY CLICK         - SEE BELOW

END:    RTI                             ;RETURN FROM INTERRUPT
```

TRANSLATING

The ram vector BTA_TRAN points to the routine used to translate the key returned from BTA_POLL. The function of this routine is to translate the key number passed in the A register into an ASCII character and return it in the A register.

The following code can be used to translate a key number (0 to 36) into the ASCII key it represents on the standard Organiser. It uses KBB_STAT to decide whether 'shifted' characters, or lower-case characters are returned and also to refresh the cursor in the correct state (block or line). If the key pressed is SHIFT CAP or SHIFT NUM, no key is returned (i.e. the A register is 0) but the KY_CPNM flag is set.

```
KB_TRAN:
        LDA     B,KBB_STAT:             ;GET KEYBOARD STATUS FLAGS
        AND     B,#<$FF-KY_CPNM>        ;CLEAR CAP/NUM FLAG
        BPL     3$                      ;BRANCH IF NOT SHIFT,NOT CAP OR NUM
        CMP     A,KBB_CAPK              ;IS IT THE CAP KEY ?
        BNE     1$                      ;BRANCH IF NOT
        EOR     B,#KY_CAPS              ;TOGGLE CAPS FLAG
        BRA     2$
1$:     CMP     A,KBB_NUMK              ;IS IT THE NUM KEY ?
        BNE     3$                      ;BRANCH IF NOT
        EOR     B,#KY_NUMB              ;TOGGLE NUM FLAG
2$:     ORA     B,#KY_CPNM              ;SET CAP/NUM FLAG
        CLR     A                       ;RETURN NO KEY

3$:     PSH     A                       ;SAVE KEY PRESSED
        OS      KB$STAT                 ;SET KEYBOARD STATE
                                        ;PRESERVES B
        TBA                             ;SET A TO KBB_STAT
        PUL     B                       ;B IS KEY PRESSED
        TST     B
        BEQ     9$                      ;BRANCH IF NO KEY PRESSED
        PSH     B
        LDX     BTA_TABL                ;ADDRESS OF KEYBOARD LOOKUP TABLE
        DEX                             ;SO KEY 1 IS FIRST KEY IN TABLE
        LDA     B,KBB_STAT:             ;GET KEYBOARD STATUS
        ASL     B                       ;EXCLUSIVE OR KY_SHFT WITH KY_NUMB
        BVC     4$                      ;BRANCH IF NOT 'SHIFT' MODE
        LDA     B, 36
        ABX                             ;SKIP TO 'SHIFTED' LOOKUP TABLE
4$:     PUL     B
```



```
        ABX                             ;INDEX INTO TABLE
        LDA     B,0,X                   ;GET ASCII KEY FROM TABLE
        LSR     A                       ;TEST CAPS FLAG
        BCS     5$                      ;IF SET LEAVE AS LOWER CASE
        CMP     B,#^a/a/
        BCS     5$
        CMP     B,#^a/z/
        BHI     5$
        ADD     B,#^a/A/-^a/a/          ;CONVERT B TO UPPER CASE
5$:
        CMP     B,#K_DEL                ;IS IT DELETE KEY
        BNE     9$                      ;BRANCH IF NOT
        ASL     A                       ;TEST SHFT FLAG
        BPL     9$                      ;BRANCH IF NOT SHIFT
        DEC     B                       ;SHIFT DEL IS ALWAYS DEL RIGHT
9$:     TBA                             ;RETURN TRANSLATED KEY IN A
        RTS
```

The keyboard table pointer KBA_TABL points to the following table by default:

```
KBT_TABL:
        ASCII  /zvpjd/
        .BYTE   K_EXE
        ASCII  /xr1f/
        ASCII  / wqke/
        ASCII  /yuoic/
        .BYTE   K_DEL
        ASCII  /tnhb/
        ASCII  /?smga/          ;(?=SHIFT - NOT RETURNED EVER !)
        .BYTE   K_MODE
        .BYTE   K_UP,K_DOWN
        .BYTE   K_LEFT,K_RGHT
        .BYTE   K_AC             ;ON/CLEAR KEY

        ;SHIFTED CHARACTERS
        ;=================

        ASCII  /.258)/
        .BYTE   K_EXE
                  $
        ASCII  $+-$
        ASCII  $/$
        ASCII  / 369%/
        ASCII  /0147(/
        .BYTE   K_DEL
        ASCII  /:$">/
        ASCII  /?;,=
```

BUFFERING

When a key press is detected, the ASCII character for that key is placed in a 16 byte wrap-around buffer, KBT_BUFF. An offset into the buffer KBB_BACK and a count of the number of characters in the buffer KBB_NKYS are used to implement the buffering. If the buffer is already full, the character is not stored, but a 'buffer-full' beep is emitted lasting 10ms. The following code can be used to buffer the character in the A register:

```
        LDA     B,KBB_NKYS:             ;GET NUMBER OF KEYS IN BUFFER
        LDX     #10                     ;10MS FOR 'BUFFER-FULL' BEEP
        CMP     B,#16                   ;IS BUFFER FULL ?
        BEQ     DOBUZ                   ;BRANCH IF YES
        LDX     #KBT_BUFF               ;ADDRESS OF KEYBOARD BUFFER
        ADD     B,KBB_BACK:             ;OFFSET INTO BUFFER
        AND     B,#$0F                  ;WRAP AROUND
        ABX
        STA     A,0,X                   ;STORE ASCII KEY IN BUFFER
        INC     KBB_NKYS                ;INCREMENT NUMBER OF KEYS
DOCLICK:
        LDA     B,KBB_CLIK              ;LENGTH OF KEY CLICK
        BEQ     END                     ;NO CLICK IF ZERO
        DEC     B                       ;LENGTH 1 WILL GIVE < 1MS CLICK
```

```
        CLR     A
        XGDX                            ;DURATION IN X
DOBUZ:  LDD     TMW_TCNT                ;SIZE OF SWITCH OFF TIME OUT
        STD     TMW_TOUT:               ;RESET TIMEOUT COUNT IF KEY PRESSED
        LDD     #56                     ;A GOOD NOTE
        OS      BZ$TONE                 ;BEEP FOR X MS
END:
```

The keyboard buffer can be cleared out using system service KB$FLSH and there is a facility for inserting a key into a 1 byte buffer KBB_WAIT by using system service KB$UGET. KBB_WAIT is always tested for a key before looking in the main keyboard buffer KBT_BUFF.

Keys can also be inserted into KBT_BUFF, but keyboard interrupts must be prevented while this is done. The following code will insert the key contained in the A register:

```
PUTA:
        SEI                             ;STOP INTERRUPTS OCCURRING IN HERE
        LDA     B,KBB_NKYS:             ;GET NUMBER OF KEYS IN BUFFER
        CMP     B,#16                   ;IS BUFFER FULL ?
        BEQ     NOPUT                   ;BRANCH IF YES
        LDX     #KBT_BUFF               ;ADDRESS OF KEYBOARD BUFFER
        ADD     B,KBB_BACK:             ;OFFSET INTO BUFFER
        AND     B,#$0F                  ;WRAP AROUND
        ABX
        STA     A,0,X                   ;STORE ASCII KEY IN BUFFER
        INC     KBB_NKYS                ;INCREMENT NUMBER OF KEYS
NOPUT:  CLI                             ;RESTORE INTERRUPTS
```

KEY CLICK

The key click is produced in the keyboard interrupt by using the routine for system service BZ$TONE, using the following code:

```
 KEYCLICK:
        LDA     B,KBB_CLIK              ;LENGTH OF KEY CLICK
        BEQ     END                     ;NO CLICK IF ZERO
        DEC     B                       ;LENGTH 1 WILL GIVE < 1MS CLICK
        CLR     A
        XGDX                            ;DURATION IN X
        LDD     #56                     ;A GOOD NOTE
        JSR     BZ_TONE                 ;BEEP FOR X MS
 END:
```

Note: BZ_TONE is JSR'ed to and not called as an operating system service. Hence, intercepting operating system calls to BZ$TONE (by re-vectoring SWI) will not affect the key click. JSR'ing or JMP'ing to operating system services can **only** be done **within** the operating system and **not** by external routines.

The length of the key click, KBB_CLIK, can be changed. Zero will disable the click and any value up to $FF will specify the length of the click in milliseconds.

Warning: BZ_TONE toggles the alarm line to produce the click (using addresses SCA_ALARMHIGH and SCA_ALARMLOW) but if SOE_B is high, it is possible that 21v will appear on any devices present, see low level pack access. For example if an EPROM device is selected, a byte may be blown on the pack. Since the click occurs on interrupt, care must be taken to ensure interrupts are disabled whenever SOE_B is set high.

ALARM CHECKING

Alarms and diary alarms are checked for only when the flag AMB_DOIT is set (e.g. by the NMI on a minute boundary):

```
        TST     AMB_DOIT                ;TEST DOIT FLAG
        BEQ     1$                      ;BRANCH IF CLEAR
        ;ALARM CHECKING CODE GOES HERE
        CLR     AMB_DOIT                ;CLEAR DOIT FLAG
1$:
```

The method of alarm checking is described in chapter 19. AMB_DOIT is cleared when alarm checking is finished to prevent it being done on the next interrupt.

FRAME-COUNTER

A 16 bit frame-counter, TMW_FRAM (address $20CB,$20CC), is incremented by one by the keyboard interrupt and wraps around every 64k:

```
        LDX     TMW_FRAM
        INX
        STX     TMW_FRAM
```


It can be used as a random number or for timing. The rate that the counter runs at is the keyboard interrupt rate and so can be calculated as follows:

TIME BETWEEN INCREMENTS (IN SECS) = (KBW_TDEL + 35) / 921600

KBW_TDEL is $B3DD by default, giving:

TIME = ($B3DD + 35) / 921600 = 0.05 SECS

KBW_TDEL can be changed to adjust the timing but will also affect everything else controlled by the keyboard interrupts, e.g. key repeat rate.

Note that the rate of the frame-counter is not 100% accurate because interrupts are disabled while running some parts of the operating system.

DISPLAY TIMING

The rate of horizontal and vertical scrolling on the display is timed using DPW_REDY (address $006D,$006E). This variable is decremented by one by each keyboard interrupt until it reaches zero, i.e.:

```
        LDX     DPW_REDY:
        BEQ     NODEX
        DEX
        STX     DPW_REDY:
NODEX:
```

TESTING FOR KEYS

Keys are tested for by looking in the keyboard buffer. While testing for a key, if no key is found, the packs may be switched off, a low battery test is made, and the machine itself may switch off.

KEYTEST AND KEYGET

The system service KB$TEST will return the first key found in the keyboard buffer (zero if no key) but will not remove it. KB$GETK uses the same routine as KB$TEST but will wait for a key to be pressed and then remove it from the buffer.

If the function of key testing is to be changed, both KB$TEST and KB$GETK must be re-vectored by intercepting system calls through SWI, (KB$GETK does not call KB$TEST through the SWI). For example, KB$TEST may be rewritten to carry out a different task when low battery is detected.

PACK SWITCH OFF

If KB$TEST finds no keys in the keyboard buffer, the pack-switch-off flag KBB_PKOF is tested. If it is non-zero, the packs are switched off by calling the system service PK$PKOF (see chapter 9).

LOW BATTERY TEST

If KB$TEST finds no keys in the keyboard buffer, a low battery test is made. Bit 0 of PORT 5 is set when the battery supply voltage drops below approximately 5.2v. If this is the case, the message BATTERY TOO LOW is displayed and the machine will switch off after 4 seconds. When the machine is revived, the program will continue in KB$TEST where it left off.

MACHINE SWITCH OFF

If KB$TEST finds no keys in the keyboard buffer, the auto-switch-off flag, TMB_SWOF, is tested. If it is non-zero, the time remaining in seconds before switch off occurs, TMW_TOUT, is tested. If TMW_TOUT is zero, the machine is switched off by calling system service BT$SWOF. When the machine is revived, the program will continue in KB$TEST where it left off.

KEYBOARD VECTORS AND VARIABLES

The following vectors and variables allow the function of the keyboard to be customised very easily. They can all be read and written to directly, except for KBB_STAT, which should only be written to using system service KB$STAT.

KBB_STAT

KBB_STAT stores the following flags:

FLAG	BIT OF KBB_STAT	DESCRIPTION
KY_SHFT	7	SET IF SHIFT KEY PRESSED
KY_NUMB	6	SET IF NUMERIC LOCK
KY_CPNM	1	SET IF CAP OR NUM KEY
KY_CAPS	0	SET IF LOWER CASE LOCK

KBB_STAT can be read directly, but system service KB$STAT should be used to write to it.

BTA_POLL

This vector points to the routine which polls the keyboard. It must return the number of the key pressed in the A register and can set flags to be used by BTA_TRAN (e.g. shift, cap, num flags in KBB_STAT).

BTA_TRAN

This vector points to the routine which translates the key number supplied from BTA_POLL in the A register into the ASCII character it represents, returned in the A register. The routine will use flags which have been set by BTA_POLL, to decide how to translate the character. For example, if the SHIFT flag is set, a 'shifted' set of characters will be returned, or if a CAP flag is set, the characters will be be returned as lower case. To decode the character, a table of characters pointed to by BTA_TABL is used.

BTA_TABL

This is the vector which points to a table of characters used to translate a key press into an ASCII character. For an example, see above. The vector can be changed to point to a new set of characters and should contain 72 characters (36 'shifted') unless, of course, the translate routine has been changed or SHIFT has been disabled.

KBB_SHFK

This flag is used to disable the SHIFT function. It is tested only in the keyboard poll routine at BTA_POLL. If the flag is set, the SHIFT function is disabled and the SHIFT key will act as a normal key. Hence, it will return the 26th character of the lookup table which is currently a "?" character.

KBB_CAPK

This byte contains the number of the key (1 to 36) required to be the CAP key. It is used only by the keyboard translate routine at BTA_TRAN. By default it is set to 32 (the up-arrow key). To disable the CAP key altogether, a number greater than 36 should be stored in KBB_CAPK.

KBB_NUMK

This byte contains the number of the key required to be the NUM key. It works in exactly the same way as KBB_CAPK.

KBW_TDEL

This word controls the rate of keyboard interrupts. When an interrupt occurs the value in KBW_TDEL is stored in the TIMER 1 OUTPUT COMPARE REGISTER 1, and the FREE RUNNING COUNTER is set to zero, so that the next interrupt will occur after KBW_TDEL clock cycles.

Hence, the time between interrupts = (KBW_TDEL + 35) / 921600 secs

There is an overhead of 35 cycles for each interrupt. The default value for KBW_TDEL is $B3DD, giving a time interval of 0.05 secs. Note that a value of zero in KBW_TDEL, will cause the machine to lock up.

KBB_DLAY

This byte stores the delay before auto-repeat of the keys begins in terms of the number of keyboard interrupts. The default value is 14, so with interrupts running at 20 times per second, the delay is 0.7 secs.

KBB_REPT

This byte stores the delay between keys when auto-repeating in terms of keyboard interrupts. The default value is 0 which is the fastest value. A value of 1 will repeat at half normal speed, 2 at a third normal speed etc.

KBB_CLIK

This byte stores the length of the keyboard click in ms. A value of zero will turn off the key click altogether. The default value is 1 giving the shortest possible click.

KBB_PKOF

This flag controls whether the packs are switched off by KB$TEST. If it is non-zero, which it is by default, the packs will be switched off whenever KB$TEST is called and there is are no keys in the keyboard buffer.

KBB_SPEC (LZ only)

Contains flags to handle the 'special' keys.

Bit 7 - set to disable all special SHIFT keys.
Bit 1 - set to allow SHIFT-SPACE.
Bit 0 - set to allow SHIFT-EXE.

SYSTEM SERVICES

This section describes the operating system calls available for keyboard handling.

KB$INIT

Initialises keyboard interrupts. The following tasks are carried out by this routine:

1. The keyboard buffer is flushed of any characters.
2. The keyboard interrupt rate KBW_TDEL is set to $B3DD.
3. The initial delay and key repeat rate KBB_DLAY, KBB_REPT are set to 14 and 0 respectively.
4. The keyboard status flags, in KBB_STAT are set to 0, i.e. KY_SHFT, KY_CPNM, KY_CAPS and KY_NUMB are cleared.
5. The length of the key click, KBB_CLIK, is set to 1.
6. The interrupts are enabled: FREE RUNNING COUNTER is set to 0, TIMER 1 OUTPUT COMPARE REGISTER 1 is set to $B3DD, bit 3 of TIMER 1 CONTROL STATUS REGISTER 1 is set, to enable the interrupt, and the I mask bit of the condition code register is cleared.

KB$TEST

Looks in keyboard buffer for a key but does not remove it. If the buffer is not empty, the ASCII value of the first key found in the buffer is returned in the B register. The unget buffer KBB_WAIT is tested for a key before looking in KBT_BUFF. If KBB_WAIT is empty and there is a key in KBT_BUFF, it is transferred to KBB_WAIT.

If no keys are found, the following will occur:

1. If KBB_PKOF is non-zero, the pack will switch off using system service PK$PKOF.
2. If TMB_SWOF is non-zero and TMW_TOUT is zero, the machine will switch off using system service BT$SWOF.
3. If low battery is detected, BATTERY TOO LOW will be displayed for 4 secs before the machine switches off.

If the I mask (bit 4) of the condition code register is set (i.e. interrupts are disabled) when KB$TEST is called, it will poll the keyboard itself allowing the operating system to run with interrupts disabled. Every function of the keyboard interrupt routine is carried out except alarm checking. Any keys found are put in the keyboard buffer, the frame-counter is incremented and the display timer is decremented etc. There is a 50ms fixed delay after polling the keyboard so that if KB$TEST is called in a loop, the keyboard is polled approximately every 50ms. After polling the keyboard itself, KB$TEST looks in the keyboard buffer as usual.

KB$GETK

Waits for a key press and returns it in the B register. This routine uses the same code as KB$TEST. If there are no keys in the keyboard buffer, a SLP instruction is executed before next testing for a key, in order to save power. When a key is detected, it is removed from the buffer simply by clearing KBB_WAIT.

The auto-switch-off timeout counter TMW_TOUT is reset to the value in TMW_TCNT at the start of this routine, so the machine will not switch off in this routine for TMW_TCNT seconds.

KB$BREK

Tests if ON/CLEAR key is pressed. The key is tested for directly by reading PORT 5 and then searched for in the keyboard buffer KBT_BUFF. If ON/CLEAR is detected, KB$BREK waits for it to be released, then flushes the keyboard buffer and returns the carry flag set. If no ON/CLEAR key is found, the carry flag is cleared.

KB$FLSH

Flushes keyboard of any characters in type-ahead buffer. KBB_BACK, KBB_NKYS, KBB_PREV and KBB_WAIT are all set to zero.

KB$UGET

Puts the key supplied in the B register into the keyboard 'unget' buffer, KBB_WAIT. The buffer can only hold 1 key. If the buffer is not already full, the key is not placed in the buffer.

KB$STAT

Sets the state of the keyboard. The B register is stored into keyboard status byte KBB_STAT.

Bit 0 of the B register set indicates lower case, bit 6 set indicates numeric lock. Note that some other bits of KBB_STAT are used as flags and should not be affected. Hence KBB_STAT should be read and bits 0 and 6 either cleared or set before calling KB$STAT, see example below.

The state of the cursor (block or line) is determined by the new keyboard status and stored in DPB_CUST. A line cursor is for 'shift' mode, otherwise it's a block cursor. The new cursor type is immediately refreshed on the display.

EXAMPLE:

```
        LDA     B,KBB_STAT:     ;GET CURRENT KEYBOARD STATUS
        ORA     B,#KY_NUMB      ;NUMERIC LOCK. LINE CURSOR.
        OS      KB$STAT         ;SET KEYBOARD STATUS
```

KB$CONK (LZ only)

Exactly the same as KB$GETK but turns the cursor ON before waiting for a key and OFF afterwards.

EXAMPLE

FULL ASCII SET FROM KEYBOARD

The example code enables the whole ASCII character set to be accessed from the keyboard. The LEFT-ARROW and RIGHT-ARROW keys become two more SHIFT keys. Holding down LEFT-ARROW and pressing the P key, for example will return ASCII character 16 which, if displayed on the Organiser, is translated as a bell character. For all the characters returned in each 'shift' mode, see the table at the end of the example.

This example is taken from the code used in the RS232 device in terminal mode, to enable the whole ASCII set to be typed from the keyboard.

There are two routines provided in the file: KB_NEW and KB_OLD. KB_NEW must be called to set up the new keyboard and KB_OLD must be called to restore the normal keyboard.

See Example Appendix B.

INTERFACE SLOTS

GENERAL

The Organiser interfaces to the outside world through three slots:

1. SLOT 1. Side entry, top (nearest display)
2. SLOT 2. Side entry, bottom
3. SLOT 3. Top entry

Slots 1 and 2 are generally used for memory devices such as datapacks and rampacks, and slot 3 is normally used for communications or other interfaces. Electrically all three slots are very similar, and the distinction is for ergonomic reasons. Internally the three slots are connected together as a bus, carrying 8 bit bi-directional data, power and control lines to select and control devices plugged in.

This section describes the bus hardware in general terms, and should be read together with the sections on specific devices and control software to gain a full understanding of its operation.

SLOT CONTROL BUS

All three slots have 16 connections. In general they are connected as a bus, but there are some minor differences in signals particularly to slot 3. The slot signals and their functions are listed below:

power rails	0 volts	all slots	system ground
	Vcc3	all slots	5 volt rail, switched under software control
	Vpp	slots 1,2	21 volt rail for programming datapacks
	Vb	slot 3	system power rail for power in or out (5.5 to 11 volts)
data bus	SD0 - SD7	all slots	8 bit data bus from processor port 2
control bus	SCK	all slots	4 general control lines from processor port 6
	SMR		
	SOE_B		
	SPGM_B	slots 1,2	
slot selection	SS1_B	slot 1	3 control lines from processor port 6, used to select the current active slot
	SS2_B	slot 2	
	SS3_B	slot 3	
other	AC_B	slot 3	input for external switch on

see Datapack Connector for pinout.

POWER RAILS

A detailed description of the power supply circuitry is included in chapter Power Supply Board. The main properties of the externally available power rails are included here for reference.

Vb (slot 3 only)

This is the main Organiser system power rail, and is fed by the Organiser battery via a forward diode. All Organiser regulated power rails are derived from this rail. Vb can be used as a power output or as an external power input. As a power output, the battery voltage minus a diode drop will appear at this pin (5.5 to 8.5 volts dependent on battery condition). As a power input, the voltage applied should be higher than the battery voltage to ensure no current drain from the battery. It is recommended that an external power source also feeds the Vb pin through a forward diode, to ensure no reverse current to the external source when it is powered off. In this configuration power for the system is drawn from either the internal battery or the external power source, whichever is supplying the higher voltage. Vb should be between 5.5 and 11.0 volts under a maximum system load of 175 mA.

The lower limit is determined by the dropout voltage of the 5 volt pass regulators (the low battery indicator is triggered at approx. 5.3 volts on Vb). The upper limit is defined primarily by the Vcc3 pass regulator - see below.

Vcc3 (all slots)

This is the main power rail to the slots, and is regulated to 5 volts +/-5%. It is derived from the Vb rail above. The regulator is a low-dropout type with a PNP pass transistor rated at 1 watt. At a Vb voltage of 11 volts the maximum DC current capacity of Vcc3 is therefore 167 ma (167*(11-5)=1000 mW). In practice 150 ma should be used as the rating of this rail, remembering that all three slots are powered in parallel. The power budget allocated to each slot is 40 ma for an idle device and 70 ma for an active selected device. Only one slot should be active at any one time, giving 40+40+70=150 ma as the peak power drain with three devices present and one active.
Vcc3 is switched on and off by the PACON_B signal from the processor port 6, bit 7. When this bit is defined as input the PACON_B signal is pulled high to leave the regulator in the off state. (When off, only leakage current of a few nA will be supplied to the Vcc3 rail). To switch Vcc3 on, port 6 bit 7 should be defined as output and low (0).

Vpp (slots 1,2)

This rail is designed specifically for programming of datapack EPROMs, and may assume one of three voltages:

1. 0 volts when Vcc3 is off
2. 4.5 volts when Vcc3 is on (diode drop below Vcc3)
3. 21 volts +/-2% when the 21 volt regulator is on

The 21 volt state is normally used in a pulsed mode under software control, for programming EPROMs with defined algorithms. This is discussed further in the Datapack section of the manual.

DATA BUS (PROCESSOR PORT 2)

The data bus SD0-SD7 is an 8 bit bi-directional bus to all three slots, and is controlled from the processor I/O port 2. The notes below summarise port 2 operation in the context of the Organiser system.

The primary use of port 2 is as an eight bit parallel I/O port. Two registers control this function:

- Port 2 data register $0003
- Port 2 DDR $0001

The DDR determines the I/O direction of the port bits (0 for input, 1 for output). Only 2 bits of the DDR are active:

- Bit 0 defines the direction of SD0
- Bit 1 defines the direction of SD1-SD7

The DDR is a write-only register, and read-modify-write instructions should be used with caution.

With the DDR set to input, data present on the bus can be read through the data register. If no slot is active a $00 will be returned, defined by the eight pull-down resistors on the data lines.

When the Organiser is off (processor in standby mode) the DDR is automatically set to input, and remains in this state on system initialisation. In subsequent operation this should be used as the rest state, and in particular should always be set to input whenever Vcc3 is switched off.

With the DDR set to output, data can be set onto the bus by a write to the data register. Data is latched into the register, and will remain on the bus until a further write. Note that data can be written to the data register with the DDR set to input, and this data will be set onto the bus when the DDR is turned round.

Control of the bus and bus direction is entirely under software control. Control of devices in the slots is described further in the next section, but it is important to stress here that control of the port 2 DDR is vital for proper bus operation. A condition where the DDR is set to output and a slot device is also outputting to the bus should not be allowed to occur if bus contention and possible device damage are to be avoided.

In addition to the data bit I/O function, each bit of port 2 has a secondary function which may be selected under software control. When selected, the relevant bits assume their secondary function, overriding the DDR setting where necessary. The secondary functions are described in the processor manual. An example of their use is the Organiser RS232 interface, which uses the internal serial communications interface and the port 2 Tx and Rx bits. Note that in special cases such as this, various bits of the data bus may separately be defined as inputs and outputs simultaneously.

CONTROL LINES (PROCESSOR PORT 6)

Port 6 of the processor is an 8 bit I/O port controlled by two registers:

- Port 6 data register $0017
- Port 6 DDR $0016

The DDR determines the direction of the port bits (0 for input, 1 for output). Each bit of the DDR determines the direction of the corresponding bit of the data register. The DDR is a write only register, and read-modify-write instructions should be used with caution.

When the Organiser is off (processor in standby mode), the DDR is automatically set to input and remains in this state on system initialisation. In this case the lines from the ports will take up states defined by the relevant external pull-up and pull-down resistors.

The port bits are defined as follows:

bit 7	PACON_B	This bit is used to switch the main Vcc3 power rail to the slots as described above.
bit 6 bit 5 bit 4	SS3_B SS2_B SS1_B	These three bits are used to select the current active slot. The rest state should be with all three bits set high i.e. with all slots inactive. Note that when the relevant port bits are set to input, these lines are pulled high by external 6k8 ohm pull up resistors to the Vcc1 voltage rail. Vcc1 is the supply rail to the processor board and is present at all times (including when the Organiser is off). Of particular importance, these lines are pulled high whether or not the Vcc3 rail is on. This has been designed so that a small amount of power - of the order of 10 micro-amps -can be drawn by each of the slots through the slot select line when the slots are otherwise powered down. Rampacks are an example of the use of this facility. To protect against unwanted power drain through this line, a blocking diode or transistor are normally employed in each device between the slot select input and the device circuit. A slot is selected by setting a "0" onto one of the three lines. Only one should be selected at any one time, and the software is responsible for ensuring this. Each device should be designed so that it can only output to the bus when its slot select line is pulled low
bit 3 bit 2 bit 1 bit 0	SOE_B SPGM_B SMR SCK	These are four general purpose control lines used to control devices in the slots. All four are wired to slots 1 and 2, but the SPGM_B signal is not available on slot 3. With the port bits defined as input, the rest state of all except SPGM_B is low. SPGM_B is pulled to the Vcc3 rail via a resistor, and so will be low with Vcc3 off and high with Vcc3 on. The way these lines are used is to some extent dependent on the type of device currently selected. They are normally used as outputs to control devices, but under special circumstances one or more of the lines may be defined as input. The four signal names are related to the functions of the lines when used to control 8 or 16k datapacks: • SOE_B directly controls the datapack EPROM OE_B signal • SPGM_B directly controls the datapack EPROM PGM_B signal • SMR resets the datapack address counters • SCK clocks the datapack address counters These meanings are not fixed and the lines can be used in different ways depending on the particular active device.

AC_B INPUT

The AC_B signal from slot 3 can be used by a top slot device to switch the Organiser on. Its function is the same as pressing the Organiser AC key, but it is only effective when the machine is off.

To activate this function, the AC_B signal should be pulled low by the external device, by an open-collector npn transistor or other means (internally the signal is pulled up to Vcc1 by a 47k ohm resistor). The RS232 interface is an example of the use of this.

If the Organiser is off, pulling the AC_B line low will switch it on. When the Organiser is on the AC_B signal is disconnected and has no effect. Note that if the line is pulled low permanently the Organiser will restart whenever it tries to switch off.

PACKS

GENERAL

Packs are the removable storage medium used in the Organiser. On a Pack the data is stored in either

- EPROM (ultraviolet Erasable and electrically Programmable Read-Only Memory). These packs are called Datapacks.
- Flash-EEPROM (Electrically Erasable and Programmable Read-only Memory). These packs are called Flashpacks.
- Battery buffered RAM (Random Access Memory). These packs are called Rampacks.

Storing Data on Packs has certain advantages and disadvantages when compared to conventional storage such as floppy disks. Packs are much more robust than disks - e.g. you cannot scratch a datapack and they are not susceptible to magnetic fields. They require no special drives - they can be driven from a simple I/O port. This results in higher reliability (no moving parts), much lower power consumption (no motors) and a much smaller package than floppy disk drives.

All packs can be read faster than disks but the write time of Data- and Flashpacks is comparatively slow.

Also once written to, the space on Data- and Flashpacks is permanently used even if the data is deleted. Once a pack gets full, the complete pack has to be erased. While Flashpacks are formatted within the Organiser, Datapacks must be placed under an Ultraviolet lamp.

Selective erasure of one part of a pack other than a Rampack is not possible.

PACK TYPES

DATAPACKS

These are the most common memory packs. They are available in several sizes: 8, 16, 32, 64, 128k, and even made larger ones by 3party manufacturers. Note that the CM model organiser cannot use packs larger than 64k.

Datapacks of 8 and 16K were first produced for the Series One Organiser. The Series II offers compatibility so those old packs can be used with it.

Datapacks contain an EPROM (Erasable Programmable Read-only Memory) chip. Such chips can be written to by using a high voltage, and once that is done the data contained on it is very safe. Writing to a datapack will cost considerable battery power and is very slow.

Information stored on a datapack will last for many years without the need of a power supply.

EPROM's are erased by subjecting them to ultraviolet light through the quartz window on top of the chip. The database cabinet has a hole underneath the label to access the window without having to open it.

The recommended erasure procedure is exposure to UV light which has a wavelength of 253.7nm. The integrated dose (i.e. UV intensity x exposure time) for erasure should be a minimum of 15W-sec/cm2. The erasure time with this dosage is approximately 15 to 20 mins using a UV lamp with a 12mW/cm2 power rating. The EPROM should be placed within 2.5cm of the lamp tubes during erasure.

FLASHPACK OVERVIEW

Detailed Flashpack Information is provided in an extra chapter.

Flashpacks on the Organiser represent a substantial improvement in storage technology. There are four major advantages:

- significantly less power used when writing,
- nonvolatile,
- can be formatted in place,
- bigger than available EPROMs.

Writing is done in a completely different way to EPROMs and a special software driver must be present. The driver software comes as a bootable device on every standard Flash Datapack. Once booted, the flashpack driver is resident, i.e. it does not get removed when On/Clear is pressed, even if the pack is no longer present. If the driver is not present and a write is attempted a "READ ONLY PACK" error is reported.

Flashpacks can be formatted in place by using the main menu option FLASH (automatically established by the driver) or by calling the OPL procedure FLSHFORM:("B",1) where the first parameter is the slot (B or C). FLSHFORM:("B",0) operates the same way without displaying a message.

Formatting does consume a noticeable amount of power and it is suggested that a power supply unit is connected.

As after formatting the driver has to be copied back to the pack, there must be another flashpack (or a flash formatter pack) present, otherwise a DEVICE MISSING error will occur.

The CM operating system does not handle flashpacks.

RAMPACKS

Rampacks are packs that have RAM rather than EPROM as their storage medium. This has advantages and disadvantages:

- RAM can be written to much faster than EPROM (the same speed as reading in fact).
- The RAM used in rampacks is CMOS RAM and uses less power than EPROM.
- RAM can be altered and does not require formatting.
- RAM is volatile and requires a back up battery to retain its contents.
- Data in RAM is not as secure as in EPROM and is more easily corrupted (e.g., by pulling rampack out of Organiser while it is being accessed).
- RAM is much more expensive byte for byte than EPROM.

The CM operating system does not handle rampacks.

WARNING

Rampacks contain a lithium battery that has an estimated lifetime of 5 years.

A Rampack with a **dead battery** is still fully functional within the organiser, but if at any time it is plugged out, **all data is lost!** This may also happen when the organiser is left unpowered (e.g. while battery changes).

There is no way to check the pack battery by software. However older documents list a PACK BATTERY LOW (191) error which was removed later, as the feature was never implemented.

Usually there is no need for formatting a rampack, nevertheless the following program might become useful in case of spurious read errors:

```
FMTRAM:
REM  PROGRAM TO FORMAT RAMPACK
LOCAL a%(11)
CLS :PRINT "Format C: Y/N ?"
IF LOC("Yy0",GET$)
   rem writes a zero byte at the beginning of the rampack
   rem this invalidates the pack and the OS automatically reformats
   a%(1)=$4f37  :rem     CLRA, PSHB
   a%(2)=$3f62  :rem     OS pk$setp
   a%(3)=$3225  :rem     PULA, BCS $1
   a%(4)=$0d36  :rem           PSHA
   a%(5)=$4f36  :rem     CLRA, PSHA
   a%(6)=$30c6  :rem     TSX,  LDAB $01
   a%(7)=$013f  :rem           OS pk$save
   a%(8)=$6131  :rem           INS
   a%(9)=$334f  :rem     PULB, CLRA
   a%(10)=$3f62 :rem $1: OS pk$setp
   a%(11)=$3900 :rem     RTS
   USR(ADDR(a%()),2) :REM 1=Pack B, 2=Pack C
ENDIF
STOP
```

DEVICE A: INTERNAL RAMPACK

Device A: (also known as PACK A:) is the internal RAM of the Organiser which is treated as if it were a rampack. It is accessible via the same operating system services that handle datapacks.

Obviously the amount of memory available to Device A: varies depending on the machine type and on other demands on memory (e.g. the Diary and OPL programs).

DEBUG RAMPACK

A special type of rampack can be produced by setting the pack header ID byte (see below) to $3C and then adjusting the checksum word. The Rampack thus produced will automatically back up all (XP) or parts of the internal memory (LZ) in case of a TRAP error (i.e. a system crash).

A few seconds after displaying the TRAP message, the organiser will switch off. If a debug rampack is is present in one of the slots when the organiser is switched on again, all of the non-paged RAM area (see memory map) is copied onto the pack. Only low level pack access procedures are used in order to preserve as much of the zeropage as possible.

This is intended for debugging purposes, but may also be helpful for rescuing data.

SOFTWARE

As well as the 3 slots through which the Organiser can access datapack devices (or in the case of the top slot datapack like devices) it can also access the internal RAM as a datapack type device. From the top level menu and from OPL the following naming convention is used:

DEVICE A: Internal RAM of the Machine.
DEVICE B: Upper side slot (closest to the top slot).
DEVICE C: Lower side slot.
DEVICE D: Top slot.

VARIABLE USAGE

Described below are the variables used by the pack handling system services.

PKB_CURP	$8B	Current device being looked at by the operating system. It contains the contents of B register on the last call to PK$SETP. If this is zero (i.e. device A: internal RAM pack selected) then the ports may be left selecting device B,C or D. See also PKB_CPAK.	Set by PK$SETP
PKB_CPAK	$8C	Actual current slot. Only set if the current slot is powered up and selected. Set to $FF when packs are turned off by PK$PKOF. If this byte is zero then device A: has been selected but the slots will be still powered up.	Set by PK$SETP and PK$PKOF
PKW_RASI	$8D	Length of internal RAM pack (device A:).	Set by all pack routines when accessing ram file.
PKW_CMAD	$8F	Offset into RAM file.	
PKB_HPAD	$91	High order byte of pack address.	Set by all pack routines when accessing devices B,C,D.
PKW_CPAD	$92	Pack address.	
PKA_PKID	$94	Pointer to current pack identifier in array PKT_ID.	Set by PK$SETP.
PKT_ID	$20D7	An array of 4 elements each 10 bytes long that contain the id string of each of the 4 devices A:,B:,C: and D: respectively.	

PACK ID STRING

Every pack logged on the Organiser has a ten byte string for identification. This string is blown onto a blank pack by the Organiser during "sizing". It consists of an ID byte, a size byte, a 6 byte time string and a 2 byte checksum.

Note that the time string (bits 2-7) is replaced by device information on bootable packs (see details in chapter External Interfacing).

		normal pack	bootable pack
0	ID	Id byte. This byte describes the pack type and function. The bits of this byte signify the following: 0 This is clear for a valid MKII Organiser pack. 1 This is cleared if the pack is a ram pack. 2 This is set if the pack is paged. 3 This is cleared if the pack is write protected. 4 This is cleared if the pack is bootable. 5 This is set if the pack is copyable. 6 This is cleared if the pack is a flashpack or a debug ram pack. 7 This is set if the pack is a MK1 Organiser datapack.	
1	SZ	This contains the size of the pack in 8k (8096) units (e.g. =1 for an 8k pack, 2 for a 16k pack etc.).	
2	YR	The year the pack was sized.	0 for a software application 1 for a hardware device (for descriptive purposes only)
3	MNTH	The month the pack was sized.	device identification number
4	DAY	The day of the month the pack was sized.	Device version number
5	HR	The hour the pack was sized.	boot priority (by convention same as device identification number)
6 7	FRH FRL	A unique two byte number, generated by reading the contents of the free running counter at the time of sizing.	Device code pack address
8 9	CHKH CHKL	A word checksum of the first 4 words of the ID string.	

PK$SETP sets PKA_PKID to point to the relevant ID string in PKT_ID for the requested slot. When PK$SETP is called and a valid pack is in the requested slot then the first 10 bytes of the pack are compared to the 10 bytes stored in RAM for that particular slot. If they are not the same, then the 10 bytes from the pack are copied into RAM and, if requested, a "PACK CHANGED" error is reported.

The word checksum is used in rampacks to determine if they have been corrupted. If this checksum is not correct in a rampack it is assumed that they have been corrupted and they are filled with $FF and then resized (all data in them is lost). This does not apply to the CM operating system as it does not handle rampacks.

The first 2 bytes of the ID string are useful in determining the pack type and size.

Note that if bit 7 of the ID byte is set (MK1 Organiser datapack), the id string on the datapack will be different to that stored in the id table. The operating system converts the MK1 id string on the datapack to a valid MKII id string in the id table.

SYSTEM SERVICES

This section describes the operating system calls for pack handling.

These routines provide all that the programmer should need for datapack accessing. They automatically take care of reading and writing to the different types and sizes of packs giving the programmer a consistent interface irrespective of the pack which is plugged in.

However the operating system also offers more comfortable file handling services.

These routines will access the internal RAM of the Organiser (referred to as PACK A) as if it were an external rampack.

PK$SETP

Sets up the operating system to access the current pack.

It powers up the slots (if they are powered down) and selects the required slot. It then detects if a pack is plugged into that slot and if so "logs on" that pack.

This routine must be run before calling any further pack routines and must be called again if the packs have been turned off or the slots have been modified in any way (i.e. by a user machine code program).

PK$SAVE

Save a given number of bytes at the current position in the current pack.

PK$READ

Reads a given number of bytes from the current position in the current pack into a buffer.

PK$RBYT

Reads a byte from the current position in the current pack.

PK$RWRD

Reads a word from the current position in the current pack.

PK$SKIP

Skips the pack's address up by a given number.

PK$QADD

Returns the current pack address.

PK$SADD

Sets the current pack address.

PK$PKOF

Turns off all slots.

PK$SETP or PK$SADD must be called before the packs can be accessed again.

PACK ERRORS

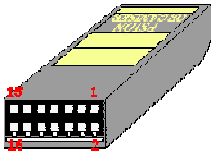
The following errors can occur when accessing the datapack slots.

NAME	VALUE	DESCRIPTION
ER_PK_BR	200	READ PACK ERROR
ER_PK_NP	246	NO PACK IN SLOT
ER_PK_DE	245	WRITE PACK ERROR
ER_PK_RO	244	ATTEMPTED WRITE TO READ ONLY PACK
ER_PK_DV	243	BAD DEVICE NAME
ER_PK_CH	242	PACK CHANGED
ER_PK_NB	241	PACK NOT BLANK
ER_PK_IV	240	UNKNOWN PACK TYPE
ER_AL_NR	254	RAM PACK FULL (NO MORE ROOM IN RAM)
ER_GN_BL	194	BATTERY TOO LOW

CONNECTOR PINOUT

There are sixteen lines on the datapack connector, as follows:

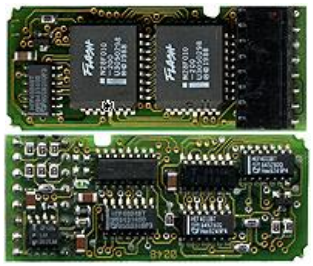
SD0-7	8 data lines
SMR	Master Reset
SCLK	ClockOutput
SOE_B	EnableSlot
SS_B	Select
Gnd	Ground (0v)
SPGM_B	Program
SVCC	5v
SVPP	Program Voltage



15	13	11	9	7	5	3	1
SVCC	Gnd	SOE_B	SMR	SD7	SD5	SD3	SD1
SVPP	SPGM_B	SS_B	SCLK	SD6	SD4	SD2	SD0
16	14	12	10	8	6	4	2

Note that lines which are active low (inverted logic) are followed by '_B'.

FLASHPACKS



THE FLASH DRIVER

VERSION CONFLICTS

The driver software comes as a bootable device on every standard Flash Datapack and on the Flash Formatter packs.

If you have more than one Flash Datapack, they may have different driver versions.

You must install whichever version is the latest.

You see the version number when you select the 'Flash' option. (On XPs, the number is on the end of the menu.)

Note that the original flash pack driver did not display a version number.

- If you do have different driver versions
  - install the most recent version,
  - format any Flash Datapacks which have the old version.
  - Otherwise, when you come to format the newer Datapack, the formatting process would copy the older version of the software from the older Datapack onto the newer one.
  - Be sure to backup any data from the older Datapack - perhaps to the newer one - before formatting it.
- If you are installing a new pack
  - first note the version of your current software, then remove the software.
  - Install the software from the new pack, and note its version number.
  - It will probably be the same version, or a later one. You can now use all of your packs.
  - In the unlikely event that the new driver is an earlier version that the one from your original pack, remove the new pack, re-install previous driver and format the new pack.

STANDARD DRIVER

The flashpack driver is is permanent, i.e. it does not get removed when On/Clear is pressed, even if the pack is no longer present.

In order to achieve this it has priority $F8 and assumes that it is the first item booted in. Anything else already booted in will be therefore removed from the device list.

To make the device permanent two alternative strategies are followed depending on whether the code is booted in the low 7K or in the PERMCELL. If it is booted in the low 7K then 'dva_bot' is set to the same as 'dva_top'. If it is booted in the PERMCELL then the base value (at $2000) is made equal to the value at $2002. To ensure that the flash driver is not booted in more than once bit 2 in 'dvt_spar' is set.

Every time the Organiser is rebooted 'FLASH' is re- inserted in the menu. This is to avoid the situation where FLASH has been deleted from the menu.

The flash driver re-vectors the following OS routines:

- fl$renm
- fl$copy
- dv$vect
- pk$setp
- fl$writ
- fl$deln
- dv$lkup
- pk$save
- fl$eras
- fl$bdel
- dv$clsr

MULTIBOOT DRIVER

The multi-boot Flash driver allows .BIN files following the driver code to be booted in.

This is an extension to the standard Flash Driver. It requires about 500 bytes of extra code some of which will be taken out of the PERMANENT cell. For this reason it is not included in the standard driver.

The first time it boots as well as making itself permanent, it boots all the long records (02 80 type that are assumed to be .BINs) immediately after the bootable code on the pack. The second time (ie when the driver is already present) it will be boot into the PERMCELL (the existing flash driver will already take up most of the lower 7K). Once it has determined that the flash driver is already present it minimises its size and boots the following long records.

You must use 3.20 or above of BLDPACK and MAKE to utilise this feature.

WARNING: BLDPACK is unable to adjust any absolute pack addressing (used for loading overlays) in the Bins that it strings together.

DRIVER REMOVAL

Before removing the driver all other devices and their drivers must be removed by pulling out all bootable packs and devices like commslink, printer and then pressing On/Clear at the top level. This is to avoid re-revectoring problems.

Now the driver can be removed by option REMOVE in the FLASH menu.

If a device is still present the REMOVE fails with the message "BAD DEVICE CALL".

HARDWARE

Flash Datapacks consist of either one or two Flash chips on a board with associated regulator, counters and logic. All circuitry is CMOS so reading requires very little power.

Flash Datapacks are paged and segmented. Segments control address lines A14 to A18 (16K byte blocks) and pages control lines A8 to A13 (256 byte blocks). So to get to the last byte requires the segment to be set, 63 toggles of the page and 255 toggles of the address. This will take in the order of 1 millisecond.

All Organiser models can read data from Flash Datapacks except CMs which are not able to recognise segmented packs.

Writing

Writing is done in a completely different way to EPROMs and is implemented by the driver. If the Flash driver is not present any write attempted results in a "READ ONLY PACK" error.

All Flash Datapacks have their write protect bit set to zero so that unless the Flash driver is present no writing can take place. In order that Flash Datapacks can be write protected the most significant bit of byte 8 of the header is used as the alternate write

protect bit. When a pack is freshly formatted it is always 1 (ie write enabled).

Formatting

Flash Datapacks can be formatted in place. To do this each byte must be written to zero and then for the one or two chips present they must commanded to erase back to FF. When doing this each byte must be checked to ensure that is has been erased properly. If not it tries another erase until either all bytes are FF or the chip is judged to have failed.

As after formatting the driver has to be copied back to the pack, there must be another flashpack (or a flash formatter pack) present otherwise DEVICE MISSING error will occur.

In Flash driver Version 1.3 and greater packs can be formatted by calling the OPL procedure FLSHFORM:("B",0) where the first parameter is the slot (B or C) and the second is whether a display is required. If the second is non-zero then the standard display is given otherwise the screen is not altered.

Formatting does consume a noticeable amount of power and it is suggested that a power supply unit is connected.

FORMATTER PACKS

After formatting the driver has to be copied back to the pack, therefor a pack containing the driver must reside in the other slot. If no other flashpack is at hand, a ram- or datapack must be prepared to store the driver.

Flash formatter packs also provide a reliable means to update the flash driver software.

Important: After having installed the new driver by removing the old one and booting the formatter pack, make sure you format ALL (!!) of your flashpacks to avoid version conflicts!

Never use a Formatter pack if the Flashpack itself has a later version of the software!

Developer Pack's BLDPACK and MAKE are used to create the pack.

This sample .BLD file will build a formatter pack.

```
FLSHFORM   16              !packname, size in kb
FDRIVE19   BIN             !FLASH driver
```

The drivers size is about 7.4k, the rest of the Pack is free and may be used as usually.

Note that datapacks must be blank and unsized to receive the device boot code.

Latest known driver versions:

- Standard Driver V1.9
- Multiboot Driver V1.8

Download

LOW LEVEL PACK ACCESS

Although a deep understanding of the hardware of datapacks and their interface to the Organiser is useful it is not necessary. The operating system software provides all the routines necessary for accessing datapacks while hiding all their complexities from the programmer.

NAMING CONVENTIONS

The following conventions are used when referring to lines on the EPROM in the datapack and the external datapack connector.

1. Lines on the EPROM are preceded by 'E'.
2. Lines on the datapack and Organiser connectors are preceded by 'S'.

3. Lines which are active low (inverted logic) are followed by '_B'.

Hence the chip select line on the EPROM is abbreviated ECS_B and the slot select line on the Organiser connector is abbreviated SS_B.

HARDWARE

The following is a selection of the packs available for use on the MKII Organiser:

- 8k bytes with linear addressing
- 16k bytes with linear addressing
- 32k bytes with linear addressing
- 64k bytes with 256 byte paged addressing
- 128k bytes and more with 256 byte paged and 16k byte segment addressing

Packs above 64k are not supported in the CM version of the MKII Organiser.

The addressing modes are explained more fully below. Although the same physical connector is used to connect the different types of datapack, the way the individual lines are used varies.

A datapack consists of 2 major components, an EPROM and a counter. The EPROM is the device where the data is stored and the counter is used to select where in the EPROM data is stored and retrieved.

EPROM (Erasable Programmable Read Only Memory)

EPROMs are read only memory devices but they can be written to (programmed) and erased under special circumstances. It is worth noting the following important facts about EPROMs.

1. In a blank (erased) EPROM all the bytes are equal to $FF (all bits set).
2. EPROMs are erased by subjecting them to ultraviolet light through the quartz window on top of the chip.
3. Bytes are written by applying a special set of voltages and signals to the EPROM.
4. When writing to an EPROM, bits can only be modified from a 1 to a 0.

The EPROMs used in datapacks have an 8-bit data bus. The size of their address bus varies from 13 bits for an 8K EPROM to 16 bits for a 64K EPROM. Other lines that can be present on an EPROM are as follows:

- Chip Select (abbreviated ECS_B): when this line is low the EPROM is active. When this line is high the EPROM will ignore any other signals present on any other lines.
- Output Enable (abbreviated EOE_B): when this line and ECS_B are low the EPROMs outputs are active and it is possible to read from the EPROM.
- Programming Supply Voltage (abbreviated EVPP): this line is normally at 5V but is raised to between 12V and 21V (depending on size and type of EPROM) when writing to the EPROM. On 32k and larger EPROMs, EVPP is combined with EOE_B.
- Program (abbreviated EPGM_B): only on some EPROMs (8k,16k). It is used when writing to the EPROM. On 32k and larger EPROMs EPGM_B is combined with EOE_B.

NB. Although the names of these lines are similar to those present on the datapack connector there is not necessarily any direct connection between similarly named lines.

Reading from EPROM

In general to read from an EPROM the following procedure must be followed.

1. Set up the required address on the EPROM
2. Set EOE_B low
3. Set ECS_B low
4. Read data on the data bus
5. Set ECS_B high
6. Set EOE_B high

The configuration of EVPP and EPGM_B is dependent on the type of EPROM but does not change during the read cycle.

Writing to EPROM

In an erased or blank EPROM all the bytes are $FF, i.e. all bits are 1. To write to an EPROM the relevant bits are "blown" from ones to zeroes.

So to write a $3 to the EPROM a $3 is latched on the data bus and the relevant programming pulses applied to the EPROM. This causes the EPROM to AND the current contents with $3 and store the result back into the EPROM.

If the EPROM was blank then the byte blown will be ($3 AND $FF) = $3. If, however, the EPROM was not blank and, say, the byte $81 was previously stored in the EPROM then the byte blown will be ($3 AND $81) = $1.

This can be used to advantage for writing headers on packs. If a valid record header is $81 then at a later date this can be "blown" down to $1 to indicate an erased record.

The algorithm for writing to EPROMs varies depending on the type of the EPROM. The following is an example of one algorithm to write to an 8k or 16k EPROM. The algorithm used by the operating system is not described here because of its complexity in handling the many different types of EPROM.

1. Set up the required address on the EPROM

2. Set EOE_B high
3. Raise EVPP to the required voltage (21V for 8k or 16k EPROM type)
4. Latch data onto the data bus
5. Set ECS_B low
6. Set EPGM_B low
7. Wait for 50 ms
8. Set EPGM_B high
9. Set the data bus to input
10. Set EOE_B low
11. Read back data and verify. If it is not the same as the programmed data then programming has failed (possibly due to non erased or faulty EPROM).
12. Set EOE_B high
13. Set ECS_B high
14. Lower EVPP to 5v

ADDRESSING MODES

To address an 8k byte EPROM 13 address lines are required, while a 16k byte EPROM requires 14 a 32k byte EPROM needs 15 address lines.

Hence to connect an 8k byte EPROM to be directly addressable a 27-way connector would be required consisting of the following:

1. 13 address lines
2. 8 data lines
3. 4 control lines (CS,OE,VPP,PGM)
4. 2 power supply lines (5V and 0V)

To connect a 32k byte EPROM to be directly addressable a 28-way connector would be required consisting of the following:

1. 15 address lines
2. 8 data lines
3. 3 control lines (CS,OE,VPP)
4. 2 power supply lines (5V and 0V)

Linearly Addressed Datapacks

To cut down the size of connector and to provide a more consistent interface a counter has been connected to the address lines (A1 upwards) of the EPROM. The counter is then interfaced to via only 2 or 3 control lines (depending on the EPROM type). On an 8k datapack there are 2 control lines connected to the counter:

1. Master Reset (abbreviated SMR). When this line is high the counters are reset. Also if the clock line is low (see below) this results in an address of zero on the EPROM.
2. Clock (abbreviated SCLK). This line has a dual function
    - It acts as the least significant address bit (A0). If SCLK is low then the address on the EPROM will be even. If clock is high then address on the EPROM will be odd.
    - When SCLK goes from high to low the counter is incremented resulting in the address on the EPROM being incremented by 2, but since A0 will also go low and decrement the address. The result is the address on the EPROM is incremented by 1.

Thus to set the address on the EPROM to zero SCLK should be set low and MR should be pulsed high. To then set address $1 SCLK should be set high. To then set the address to $2 SCLK should be set low again, and so on. To set the address $100 on the EPROM SMR should be pulsed high and then SCLK toggled 256 times. The counter is set up to "wrap around" the size of the pack- i.e. counting to address 8192 on an 8k pack will leave it set at address 0. This sort of pack is addressed linearly and is referred to as a linearly address datapack.

Page Counted Datapacks

Obviously accessing higher addresses in the EPROM this way increases the access time. To access the last byte of a 8k datapack SCLK must be toggled 8191 times. To decrease the access time on larger packs (32k bytes and larger) a 256 byte page counter has been introduced. This line is called SPGM_B. SPGM_B is used when writing to (programming) 8k and 16k datapacks.

On these page counted packs there are 2 counters:

1. This counter is connected to address lines A1 to A7 on the EPROM and is incremented by SCLK as described above.
2. This counter is connected to address lines A8 to A15 (depending on the EPROM size) and is incremented by bringing SPGM_B from a high state to a low state. (Going from low to high has no effect).

Both counters are reset to zero by pulsing SMR high.

To set an address on the EPROM to $100 on a page counted pack SCLK is set low, SMR is pulsed high and then SPGM_B is pulsed high.

To access the last byte of a 32k byte datapack SPGM_B must be pulsed 127 times and then SCLK must be toggled 255 times.

Both counters on a paged pack "wrap around". The counter incremented by SCLK wraps around $100 bytes, i.e. toggling SCLK $100 times will not change the address on the EPROM. The page counter wraps around the size of the pack, i.e. pulsing SPGM_B 128 times will not change the address on a 32k EPROM.

This sort of pack is referred to as a "page counted" or "paged" pack. Both linear and paged 32k datapacks have been produced.

Segmented Datapacks

A 128k byte EPROM would normally require 17 address lines which would make it physically too large to fit inside a datapack. However a version of a 128k byte EPROM is made which consists of 8 x 16k byte segments. The EPROM is addressed as a normal

16k byte EPROM with 14 address lines. Inside the EPROM there is a segment register which you can write to via the data bus by setting up a special condition on the control lines.

To write to the segment register the EPROM must be selected and its outputs disabled. To do this, the lines SOE_B (output enable) and SS_B (slot select) must be used.

The following procedure must be followed to write to the segment register:

1. Set SMR high
2. Set SOE_B high
3. Set SPGM_B Low
4. Make sure SVPP is set to 5V
5. Output segment number on data bus
6. Set SS_B low
7. Set SS_B high
8. Set SMR low

To access the last byte of a 128k byte datapack 7 must be written to the segment register, SPGM_B must be pulsed 63 times and then SCLK must be toggled 255 times. On a 128k pack only the bottom 3 bits of the segment address are used; the top 5 bits are ignored. Writing an 8 to the segment register of a 128k EPROM datapack is the same as writing a 0.

RAMPACKS

Rampacks are packs that have RAM rather than EPROM as their storage medium. This has advantages and disadvantages:

- RAM can be written to much faster than EPROM (the same speed as reading in fact).
- The RAM used in rampacks is CMOS RAM and uses less power than EPROM.
- RAM does not require 21V when writing to it and hence has lower power consumption.
- RAM can be altered and does not require formatting with ultraviolet light.
- RAM is volatile and requires a back up battery to retain its contents.
- Data in RAM is not as secure as in EPROM and is more easily corrupted (e.g., by pulling rampack out of Organiser while it is being accessed).
- RAM is much more expensive byte for byte than EPROM.
- Rampacks contain a lithium battery that has an estimated lifetime of 5 years.

The 32k rampack is a paged datapack and has a hardware ID byte of 1.

128k datapacks also have page counters so they are "page counted, segmented packs".

All datapack handling routines in the operating system will handle 32k and 64k paged rampacks and 128k (and more) segmented paged rampacks. The CM operating system does not handle rampacks.

USAGE

8K AND 16K DATAPACKS

All 8K and 16k datapacks are linearly addressed datapacks.

The following conditions are required to access 8k and 16k datapacks:

READING	WRITING
SCLK Don't care (either high or low)	SCLK Don't care (either high or low)
SMR low	SMR low
SVPP 5v	SVPP 21v
SOE_B low	SOE_B high
SPGM_B high	SPGM_B low
SS_B low	SS_B low

32K DATAPACKS

Both linearly addressable and paged 32k datapacks are produced.

The following conditions are required to access 32k linearly addressed datapacks:

READING	WRITING
SCLK Don't care (either high or low)	SCLK Don't care (either high or low)
SMR low	SMR low
SVPP 5v	SVPP 21v
SOE_B low	SOE_B high

- SS_B low

The following conditions are required to access 32k paged datapacks:

READING	WRITING
SCLK Don't care (either high or low).	SCLK Don't care
SMR low	SMR low
SVPP 5v	SVPP 21v
SOE_B low	SOE_B high
SS_B low	SS_B low

64K DATAPACKS

The majority of 64k datapacks produced have been paged however a small number of linearly addressed 64k packs were produced.

The following conditions are required to access 64k paged datapacks:

READING	WRITING
SCLK Don't care	SCLK Don't care
SPGM_B high	SPGM_B low
SMR low	SMR low
SVPP 5v	SVPP 21v
SOE_B low	SOE_B high
SS_B low	SS_B low

128K AND BIGGER DATAPACKS

All 128k datapacks are segmented and paged.

The following conditions are required to access 128k datapacks:

READING	WRITING
SCLK Don't care	SCLK Don't care
SPGM_B Don't care	SPGM_B Don't care
SMR low	SMR low
SVPP 5v	SVPP 21v
SOE_B low	SOE_B high
SS_B low	SS_B low

32k, 64k, 128k AND BIGGER RAMPACKS

All 32k and 64k rampacks are paged.

All 128k and bigger rampacks are segmented and paged.

The following conditions are required to access rampacks:

READING	WRITING
SCLK Don't care	SCLK Don't care. (either high or low)
SPGM_B Don't care	SPGM_B Don't care
SVPP Don't care (5v recommended)	SVPP Don't care (5V recommended)
SMR low	SMR low
SOE_B low	SOE_B high
SS_B low	SS_B low

ORGANISER INTERFACE

The Organiser has three interface slots; one top slot and two datapack slots. This section describes the two datapack slots.

The top slot is very similar to the two datapack slots but has some unique features. It is described separately in chapter External Interfacing.

ORGANISER SIDE SLOT CONNECTOR

There are sixteen lines on the Organiser side slot connector, as follows:

SD0 - SD7	Data bus
SMR	Master Reset
SCLK	Clock
SOE_B	Output Enable
SS_B	Slot Select
Gnd	0v
SPGM_B	Program
SVCC	5v
SVPP	Program Voltage

These are the same as the datapack connector.

PROCESSOR INTERFACE

The ports on the 6303 and the I/O addresses that are associated with the datapack interface are as follows:

- PORT 2 (POB_PORT2, address $3, data direction register $1). This 8-bit bi-directional port is used as the data bus to all the datapack slots (including the Top slot). When no pack is plugged in this data bus will read zero.
- PORT 6 (POB_PORT6, address $17, data direction register $16). This 8-bit bi-directional port is used for the control lines. All lines are defined as outputs when in use and inputs when not in use. When this port is set to all inputs the slots are powered down and deselected.

bit 0	SCLK	Clock line to the datapack slots
bit 1	SMR	Master reset line
bit 2	SPGM_B	Program line
bit 3	SOE_B	Output Enable line. When this line and SS_B are low the EPROMs outputs are active and it is possible to read from the EPROM. SOE_B set high enables SVPP to be switched to 21V (see also SCA_ALARMHIGH).
bit 4	SS1_B	Chip select line for slot 1. Selects slot 1 when low.
Bit 5	SS2_B	Chip select line for slot 2. Selects slot 2 when low.
Bit 6	SS3_B	Chip select line for the top slot. Selects top slot when low.
Bit 7	PACON_B	When this line is low the slots are powered up by setting VCC to 5V. When this line is high all the slots are powered down.

- SCA_PULSEENABLE ($200): Enables generation of 21V
- SCA_PULSEDISABLE ($240): Disables generation of 21V
- SCA_ALARMHIGH ($280): Enables SVPP to be switched to 21V. Also used for sound generation (key click, beep etc.).
- SCA_ALARMLOW ($2C0): Disables SVPP being switched to 21V. Also used for sound generation with SCA_ALARMLOW.
- PORT 5 (POB_PORT5, address $15): Only bit 1 (ACOUT) of this port is associated with the datapack interface. When this bit is high this indicates the 21V is ready to be switched on to the packs.

NB. Port 2 can also be used for serial communications. Bit 2 of this port is used for external clock input. To enable this bit to be used as an output $4 must be stored in the Rate/Mode Control Register (RMCR, address $10). If this is not done Bit 2 of port 2 will remain an input irrespective of what is in the data direction register. For further details see page 50 of Hitachi's HD6301X-HD6303X Family Users Manual. On reset zero is stored in the RMCR.

The following equalities are assumed in the machine code examples:

```
SCLK    =0
SMR     =1
SPGM    =2
SOE_B   =3
SS1_B   =4
SS2_B   =5
SS3_B   =6
PACON_B =7
```

Powering up the Slots

The 5v supply (SVCC) and program voltage supply (SVPP) to the slots are switchable by software. These supplies are common to all the slots.

The slots are powered up by setting PACON_B low. This sets SVCC and SVPP to 5V.

After powering up the slots there should be a delay of at least 50ms to allow the power to settle. During this delay all lines of port 6, except PACON_B, should be set to inputs. The slots should be powered down whenever possible to reduce power consumption. Setting SVPP to 21V is described below.

The following example powers up the slots.

```
; Make port 6 an input. This powers down the slots and deselects them.
; This should have been done immediately after the slots were last used.
        CLR     POB_DDR6
;
; Make port 2 inputs. Port 2 (pack data bus) should always be left inputs.
        CLR     POB_DDR2
; Initialize port 6 so it has the correct data when it is made an output.
; Make SS1_B=SS2_B=SS3_B=1,SMR=SCLK=PACON_B=SOE_B=0,SPGM_B=1
        LDA     A,#$74
        STA     A,POB_PORT6:
; Make PACON_B of port 6 an output, powers up slots
        LDA     A,#$80
        STA     A,POB_DDR6:
; Wait for power to settle
        LDX     #11520                  ; 50 ms delay
1$:
        DEX
        BNE     1$
; Make rest of port 6 outputs now the slots are on
        LDA     A,#$FF
        STA     A,POB_DDR6:
; The slots are now powered up, but none are yet selected.
```

Selecting a Datapack Slot

As can been seen from the above diagram the datapack data bus (SD0-SD7 on port 2), SMR, SOE_B and SCLK (on port 6) are common to all three slots. SPGM_B is common to the two side slots. To select which slot is in use the three chip select lines (SS1_B,SS2_B,SS3_B) are used.

The chip select lines use inverse logic so when the lines are high the slot is deselected and when a line is low the slot is selected.

WARNING: Only one slot must be selected at a time. Selecting two or more slots simultaneously can result in permanent damage to the hardware!

The following rules must be adhered to when accessing the datapack slots:

1. When not in use all slots should be left deselected and powered off where possible.
2. When not in use the data bus (port 2) should be left defined as an input.
3. In general, with the exception of toggling the SCLK line, all control line manipulation should be done with all slots deselected.

An example of selecting a slot is in the next section.

Reading from a Datapack Slot

If the slots have been powered up then the following program will read a byte from the datapack in slot 1 at it's current address.

```
; Make port2 inputs.
        CLR     POB_DDR2
; set SPGM_B high and SOE_B low
        BSET    SPGM_B,POB_PORT6:
        BCLR    SOE_B,POB_PORT6:
; select slot 1 (this is always done last)
        BCLR    SS1_B,POB_PORT6:
        LDA     B,POB_PORT2:            ; read data into B register
; deselect slot 1
        BSET    SS1_B,POB_PORT6:
```

NOTE: If there is no pack plugged into the current slot then there is always a zero on the data bus (all lines on the pack data bus are pulled low). This fact is used to detect if a pack is present. If the first byte of a pack is read as a zero then it is assumed there is no pack in the slot. Hence it is important to make sure the first byte of a pack is not zero.

Hardware ID Byte

On certain packs and devices there is an ID byte hardwired into the pack. This byte is read in the normal way except that SMR is held high during the read cycle. This byte does not exist on standard datapacks and attempting to read it will return the first byte on the pack. At present the only pack which has a hardware ID byte is the external rampack, which has an ID byte of 1. However no attempt should be made to read the hardware ID from the top slot.

If the slots have been powered up then the following program will read the hardwired ID byte from the datapack in slot 1.


```
; Make port2 inputs.
        CLR     POB_DDR2
; set SMR high
        BSET    SMR,POB_PORT6:
; set SPGM_B high and SOE_B low
        BSET    SPGM_B,POB_PORT6:
        BCLR    SOE_B,POB_PORT6:
; select slot 1
        BCLR    SS1_B,POB_PORT6:
; read hardwired ID byte into B register
        LDA     B,POB_PORT2:
; deselect slot 1
        BSET    SS1_B,POB_PORT6:
```

Setting the Program Voltage (SVPP)

It is not recommended that users attempt to write their own code to write to datapacks. The following information is only given for completeness. The 21V required for programming EPROMs is generated using a hardware "pump" similar to those found in electronic flash guns. After the pump is turned on you must wait for the voltage to reach the required level (like waiting for the ready light to appear on a flash gun).

The following lines are used to control SVPP:

- PORT 6 (POB_PORT6, address $17)
  When Bit 3 of this port (SOE_B) is set high together with SCA_ALARMHIGH, 21V (if generated) is switched to SVPP.
- Location $200 SCA_PULSEENABLE
  Enables generation of 21V (starts hardware pump)
- Location $240 SCA_PULSEDISABLE
  Disables generation of 21V (stops hardware pump)
- Location $280 SCA_ALARMHIGH
  Enables SVPP to be switched to 21V with SOE_B
- Location $2C0 SCA_ALARMLOW
  Disables SVPP being switched to 21V
- PORT 5 (POB_PORT5, address $15).
  Only one bit of this port is associated with the datapack interface:
  ACOUT- When this bit (bit 1) is high this indicates the 21V is ready to be switched on to the packs.

SCA_PULSEENABLE, SCA_PULSEDISABLE etc., are activated by accessing the relevant memory location with either a read or write. It is recommended the TST instruction is used, i.e.

```
        TST SCA_PULSEENABLE     ; starts hardware pump
        TST SCA_PULSEDISABLE    ; stops hardware pump
```

NOTE. The packs must be turned on (PACON_B low) when switching 21V to SVPP. Damage may result to the packs or Organiser if this is not done. The following sequence will switch SVPP to 21V:

1. Set SOE_B low, TST SCA_ALARMLOW
   make sure initially switched off.
2. TST SCA_PULSEENABLE
   start pump.
3. Wait for ACOUT to come high-wait until voltage is high enough.
4. TST SCA_PULSEDISABLE
   stop pump.
5. TST SCA_ALARMHIGH and set SOE_B high
   21V is now switched to SVPP.
6. Wait 5ms to allow the voltage to settle.

TST SCA_ALARMLOW or setting SOE_B low will switch SVPP back to 5V.

WARNING: The 21V should not be switched to SVPP while the pump is on as this can result in damage to the hardware.

After pumping, the voltage is stored in a capacitor. This means there is only a limited amount of energy available for blowing EPROMs before repumping is required. The operating system allows for 24ms of blowing time for each pump cycle. This will only be available after ACOUT goes high as the capacitor will discharge itself through leakage.

The pump charges a standard tolerance 100uF 40V electrolytic capacitor to 35V which is then regulated to 21V before being switched to SVPP.

NB.

1. If the 21V is switched to SVPP before the ready condition is detected, or it is left switched on and not repumped, then a voltage somewhere between 5V and 21V will be switched to SVPP. This can result in a variety of disastrous effects (e.g. destruction of data) depending on the conditions under which it occurs.
2. The operating system keyboard interrupt routine can TST SCA_ALARMHIGH under certain circumstances, so it is important that in any routine where SOE_B may be high interrupts must be disabled.

Writing to a Datapack

The EPROMs are written to using a variation of the INTEL quick pulse programming algorithm. Users are advised not to attempt to write their own EPROM programming code, but to use the supplied operating system calls.

EXAMPLE

The following code powers up the slots and then reads the first 20 bytes of a pack in slot 1. It contains no checking to see if a pack is in the slot or if it is, what sort of pack it might be.

It is recommended that the operating system be used to power up and down the slots and the second example shows how this is done. An explanation of the operating system calls is given in the next section.

```
 ; Save interrupt mask and disable interrupts
        TPA
        PSH     A
        SEI
; Initialise ports
; 4 must be stored in RMCR to allow bit 2 port 2 to be an output.
        LDA     A,#$4
        STA     A,POB_RMCR:
; Make port 6 an input. This powers down the slots and deselects them.
        CLR     POB_DDR6
; Make port 2 inputs. Port 2 (pack data bus) should always be left inputs.
        CLR     POB_DDR2
; Initialise port 6 so it has the correct data when it is made an output.
; Make SS1_B=SS2_B=SS3_B=1,SMR=SCLK=PACON_B=SOE_B=0,SPGM_B=1
        LDA     A,#$74
        STA     A,POB_PORT6:
; Make PACON_B of port 6 an output, powers up slots
        LDA     A,#$80
        STA     A,POB_DDR6:
; Wait for power to settle
        LDX     #11520                  ; 50 ms delay
1$:
        DEX
        BNE     1$
; Make rest of port 6 outputs now the slots are on
        LDA     A,#$FF
        STA     A,POB_DDR6:
; Reset pack counters with master reset
        BSET    SMR,POB_PORT6:          ; Master reset high
; Store 0 in segment register in case we have 128k pack
        BSET    SOE_B,POB_PORT6:
        LDA     A,#$FF
        STA     A,POB_DDR2:             ; make port 2 output
        CLR     A
        STA     A,POB_PORT2:            ; 0 on data bus
        BCLR    SPGM_B,POB_PORT6:
        BCLR    SS1_B,POB_PORT6:         ; select slot 1 and latch data
        BSET    SS1_B,POB_PORT6:         ; deselect slot
        BCLR    SOE_B,POB_PORT6:
        BCLR    SMR,POB_PORT6:          ; Master reset low
        BSET    SPGM_B,POB_PORT6:
        CLR     POB_DDR2                ; make port 2 inputs again
        BCLR    SS1_B,POB_PORT6:        ; select slot 1
        LDX     #DATA                   ; where to save data
LOOP:
        LDA     B,POB_PORT2:            ; read data
        STA     B,0,X                   ; store data
        BTGL    SCLK,POB_PORT6:         ; increment counter on EPROM
        INX
        CPX     #DATA+20                ; Have we reached the end yet ?
        BNE     LOOP                    ; If not loop
        BSET    SS1_B,POB_PORT6:         ; deselect slot
        CLR     POB_DDR6                ; Turn off all slots
        PUL     A
        TAP                             ; restore interrupt mask
```

The following example does the same as previous one but uses operating system calls for powering up and down the slots. If the user has to write their own pack reading routine then this is the preferred method.

```
; Save interrupt mask and disable interrupts
        TPA
        PSH     A
        SEI
; Initialize ports, power up, select required slot and reset pack counters
        CLR     A
        LDA     B,#PAKB
        OS      PK$SETP
        BCS     ERROR                   ; optional error checking
        LDX     #DATA                   ; where to save data
LOOP:
        LDA     B,POB_PORT2:            ; read data
        STA     B,0,X                   ; store data
        BTGL    SCLK,POB_PORT6:         ; increment counter on EPROM
        INX
        CPX     #DATA+20                ; Have we reached the end yet
        BNE     LOOP                    ; If not loop
; turn off slots
        OS      PK$PKOF
        PUL     A
        TAP                             ; restore interrupt mask
```


EXTERNAL INTERFACING

GENERAL

The Organiser II has been designed to be extended in a number of ways

1. Adding extra software services.
2. Adding extra hardware interfaces.

In both cases an extension to the operating system is known as a "DEVICE". As far as the operating system is concerned a "DEVICE" can be just an extra software service, or both an extra software service and an hardware interface.

In general if an extra device is added, it will probably be designed to interface to the top slot. However there is no reason why it need not be designed to run in one of the side slots. In fact it is perfectly possible to design an adapter board which will allow an interfaces to run in one of the side slots.

SOFTWARE INTERFACING

Built into the operating system is a service DV$BOOT which will load devices into the operating system. This service will scan each slot in the machine for a normal PACK of any kind which has the NOBOOT bit clear in the first byte on the PACK. Packs with this bit clear are called "BOOTABLE" packs.

Bootable packs have a special header which contains the information the DV$BOOT service requires to load the device into RAM.

Hardware interfaces like the RS232 adapter have an 8K PACK built into them as well as the interface hardware. Together they form the device.

BOOTABLE PACK DESCRIPTION

A pack which is BOOTABLE must have the following header information in the first six bytes of the pack:

ADDRESS

BYTE DESCRIPTION

0 PACK_CONTROL_BYTE

1 PACK_SIZE_BYTE

2 DEVICE_OR_CODE_BYTE

3 DEVICE_NUMBER_BYTE

4 DEVICE_VERSION_BYTE

5 DEVICE_PRIORITY_BYTE

6 DEVICE_CODE_ADDRESS_WORD

PACK_CONTROL_BYTE

This byte is used by the PACK handling services to hold various bits of information about the PACK.

Each bit in the byte has a particular function as follows:

0 This is clear for a valid MKII Organiser pack.

1 This is cleared if the pack is a ram pack.

2 This is set if the pack is paged.

3 This is cleared if the pack is write protected.

4 This is cleared if the pack is bootable.

5 This is set if the pack is copyable.

6 This is cleared if the pack is a flashpack or a debug ram pack.

7 This is set if the pack is a MK1 Organiser datapack.

Hence BIT 4 of the PACK_CONTROL_BYTE must be cleared to indicate that the PACK contains a device to be loaded into the machine. Some examples of valid PACK_CONTROL_BYTEs are as follows:

- 8K and 16K DATAPACKS: $6A i.e. EPROM device, NOT page counted, writeable, copyable and bootable.
- 32K, 64K and 128K DATAPACKS: $6E i.e. EPROM device, page counted, writeable, copyable and bootable.
- 32K RAMPACK: $6C i.e. RAM device, page counted, writeable, copyable and bootable.

PACK_SIZE_BYTE

This byte contains the number of 8K blocks in the PACK.

DEVICE_OR_CODE_BYTE

This byte is used for descriptive purposes only. It should be set to 0 the device is a software application with no additional hardware, otherwise it should be set to 1.

DEVICE_NUMBER_BYTE

This byte contains the device number of the code extension or hardware device.

As more than one device can be "BOOTED" into the operating system, it is necessary to have a mechanism to identify each of the devices currently booted. This is accomplished by having a unique device number for each device.

The device number is a value in the range $01 to $FF. However a number of these are reserved by Psion and should not be used. The reserved numbers are in the range $80 to $C0 and $01 to $40. In 1986 (when the technical manual was published) Psion already supplied a number of devices whose device numbers are as follows:

- $C0: RS232 INTERFACE
- $BF: BAR CODE INTERFACE
- $BE: SWIPE READER INTERFACE
- $0A: CONCISE OXFORD SPELLING CHECKER

By convention devices which do not have an hardware interface are allocated device numbers in the range $01 to $40 and devices with a hardware interface are allocated device numbers in the range $80 to $C0.

DEVICE_VERSION_BYTE

This byte contains the release version number of the device. This byte is not used by the operating system and is only for documentary purposes.

By convention version numbers are N.M and the byte is formed by the following:

VERSION_BYTE = N*16+M

Thus for a version number of 2.3 the byte will have the value 35 ($23).

DEVICE_PRIORITY_BYTE

This byte determines the order in which devices are booted into memory.

The priority byte may have any value in the range $1 to $FF. The higher the value the higher the priority of the device. Thus a device with priority $FF will be booted before a device with priority $FE.

The DV$BOOT service scans all the slots and builds a table of priorities from all the bootable packs. The priorities are then sorted and each device is loaded in turn. In the event of a tie in priorities the devices will be loaded in the following order:

1. SIDE SLOT B
2. SIDE SLOT C
3. TOP SLOT - SLOT 3

By convention priorities are the same as the device number. By this convention hardware devices will always be booted before software-only applications such as the concise oxford spelling checker etc.

DEVICE_CODE_ADDRESS_WORD

This word contains the address on the PACK of the device code to be booted into the operating system.

If the device code immediately follows the 8 byte header on the PACK then this address will be 8. However it is often desirable to have other information on the PACK before the device code and as such this word allows the device code to be anywhere on the PACK.

RELOCATABLE OBJECT CODE DESCRIPTION

As device code can be loaded anywhere in memory, depending on the machine type (CM, XP or LA etc.), and on how many devices are loaded, it is mandatory that the code to be loaded is in a relocatable form. The operating system provides a service DV$LOAD which will load the relocatable code into the machine and apply the relocation fix-ups.

The code pointed to by the DEVICE_CODE_ADDRESS_WORD must be in Psion's relocatable object code format. The relocatable object code format is as follows:

1. A word containing the number of bytes of code to be loaded.
2. The block of code to be loaded.
3. A word containing the checksum of the preceding block of code.
4. A word containing the number of fix-up addresses.
5. One word for each fix-up address.
6. A word containing the checksum of the preceding fix-up table.

Psion was supplying an assembler which would automatically generate object code in the relocatable format. However to illustrate the relocatable object code format, the following simulates the relocatable object code format using a normal assembler.

```
                ORG     0
0000 0011               DW      CEND-CBASE      ; SIZE OF CODE
                ORG     0100h
0100            CBASE:
0100 CE 2188            LDX     #RTT_BF         ; RUN TIME BUFFER
0103 86 20              LDAA    #20h            ; SPACE CHARACTER
0105 C6 14              LDAB    #10             ; DO 10 TIMES
0107            LOOP:
0107 A7 00              STAA    0,X             ; STORE SPACE IN BUFFER
0109 08                 INX                     ; GO ON TO NEXT BYTE
010A 5A                 DECB                    ; DECREASE COUNT
010B 26 03              BNE     LEND            ; FINISHED ?
010D 7E 0007    FIX1:   JMP     LOOP            ; NO - SO LOOP
0110            LEND:
0110 39                 RTS                     ; EXIT ROUTINE
0111            CEND:
0111 04E9               DW      04E9h           ; CHECKSUM OF CODE BLOCK
0113 0001               DW      (FIXEN-FIXST)/2 ; NUMBER OF FIXUPS
0115            FIXST:
0115 000E               DW      FIX1-CBASE+1    ; ADDRESS IN CODE BLOCK
0117            FIXEN:
0117 000E               DW      0Eh             ; CHECKSUM OF FIXUPS
```

The code must be assembled at an address above the zero page (here $100) to avoid zero page adressing.

All checksums are calculated by accumulating each byte in a word. Overflow is ignored. The following code fragment will checksum the code between CBASE and CEND.

```
        LDX     #CBASE          ; START ADDRESS
        CLRA
        CLRB
        STD     UTW_S0          ; START CHECKSUM AS 0
LOOP:
        CLRA
        LDAB    0,X             ; GET BYTE
        ADDD    UTW_S0          ; ADD CHECKSUM
        STD     UTW_S0          ; SAVE IT
```

```
        INX
        CPX     #CEND           ; ALL DONE
        BNE     LOOP            ; NO - SO DO MORE
        ; UTW_S0 NOW HAS THE CHECKSUM
```

Each entry in the fix-up table is the word-offset of a word in the code which requires relocation. After the code block is loaded, the DV$LOAD service adds the load address to all relative addresses in the code which are mentioned in the fix-up table.

If for example the above code segment was loaded at address $2000 then the fix-up of $E in the fix-up table, would result in $2000 being added to the code block at address $2000+$E in memory. This would change $7E $00 $07 to being $7E $20 $07, the correct absolute address of LOOP.

DEVICE CODE DESCRIPTION

Just loading the code from a device into memory is insufficient as the code itself must interface to the operating system. The following describes the component parts of the interface between the device code and the operating system:

```
CODE_START:
        DW      0               ; FILLED IN BY DV$BOOT
BOOT_DEVICE:
        DB      0               ; FILLED IN BY DV$BOOT
DEVICE_NUMBER:
        DB      12              ; AS IN THE PACK HEADER
VERSION_NUMBER:
        DB      $10             ; AS IN THE PACK HEADER
MAX_VECTOR_NUMBER:
        DB      3               ; NUMBER OF VECTORS PROVIDED
VECTABLE:
        DW      INSTALL         ; INSTALL VECTOR
        DW      REMOVE          ; REMOVE VECTOR
        DW      LANG            ; LANGUAGE VECTOR
INSTALL:
        ; THE INSTALL CODE
REMOVE:
        ; THE REMOVE CODE
LANG:
        ; THE LANG CODE
```

When the device is booted into memory DV$BOOT will load the size of the code into the word at CODE_START. This is necessary for DV$VECT to be able to walk the list of device drivers in memory. At the same time the slot that the code was loaded from will be placed in BOOT_DEVICE. This is to allow the device to know which slot it has been booted from. Thus if a device wants to access the PACK from which it has been booted the following code fragment can be used:

```
        CLRA                    ; REPORT PACK CHANGED ERROR
        LDAB    BOOT_DEVICE     ; SLOT DEVICE WAS BOOTED FROM
        OS      PK$SETP         ; SELECT THE SLOT
```

The DEVICE_NUMBER byte and the VERSION_NUMBER byte are the same as those on the PACK header. DV$VECT uses the DEVICE_NUMBER byte to select the right device from the list in memory. The VERSION_NUMBER byte is purely for documentary purposes.

The MAX_VECTOR_NUMBER is to allow DV$VECT to check that the device service exists. Thus if DV$VECT were used to call the device in the example above to perform vector number 3, it would fail, as it only provides vectors 0,1 and 2.

There follows a list of vectors which are used to jump to appropriate parts of the code in the device. DV$VECT is called with the device number in the A register and the vector number to be called in the B register. DV$VECT will scan the devices in memory to see if the device is present and if it is, it will then jump to the appropriate vector. All devices must provide vectors 0,1 and 2, and may provide up to 255.

DEVICE VECTORS

Every device which is loaded into the operating system has a vector table which directs the operating system to the appropriate code to handle that "VECTOR SERVICE". The first 3 "VECTOR SERVICES" have a defined meaning as follows:

VECTOR SERVICE 0 - INSTALL VECTOR

Whenever the DV$BOOT service loads a device it will call the INSTALL vector immediately after loading the device. This will give the device the opportunity to initialize itself.

For example the RS232 interface installs the COMMS menu item into the top level menu at this stage by calling the system service TL$ADDI.

On completing the required INSTALL code the device should clear the carry flag and then do an RTS instruction.

If for any reason the INSTALL code decides the device cannot run properly (i.e. not enough memory), then the carry flag should be set and the device will not be installed (and no RAM will be allocated for the device).

A device need not return from this vector, in which case it will have effectively taken control of the machine.

NOTE: The DV$BOOT service uses the first 4 bytes of RTT_FF to build the priorities table and as such the install vector should preserve these 4 bytes if there is any danger of them being corrupted. See the description of DV$BOOT for more details.

VECTOR SERVICE 1 - REMOVE VECTOR

When DV$BOOT is called, it first calls DV$CLER, which calls the REMOVE vector for each device. DV$BOOT then zeroes the permanent cell (and also resets DVA_TOP in LA/OS) effectively discarding all devices. The REMOVE code can then tidy up anything that needs to be done before the device is thrown out of memory. For example the RS232 interface removes the COMMS menu item from the main menu in its REMOVE code. The REMOVE code should terminate with the carry clear and an RTS instruction, even though any errors are ignored.

VECTOR SERVICE 2 - LANGUAGE VECTOR

This vector service is required by the OPL language and provides the mechanism by which the language can be extended.

For any procedure called in an OPL program, the language will first call the DV$LKUP service to see if any devices are prepared to handle the procedure. To do this DV$LKUP calls each device's LANGUAGE vector with the X register pointing to a leading count-byte string containing the name of the procedure. If none of the devices are prepared to handle the procedure then the language will search packs A,B,C and D for an OPL procedure of that name.

If a device is prepared to handle the procedure then it will return its device number in the A register and the vector service number which will handle the procedure in the B register. The language will then immediately call the DV$VECT service which will call the vector service.

For example the RS232 interface provides a language procedure LINPUT$: and as such the LANGUAGE vector code can be coded as follows:

```
LANGUAGE_VECTOR:
        LDD     0,X             ; GET SIZE AND FIRST LETTER
        SUBD    #7*256+'L'      ; COMPARE
        BNE     NOT_LINPUT      ; NOT A MATCH
        LDD     2,X             ; GET NEXT TWO LETTERS
        SUBD    #'IN'           ; COMPARE
        BNE     NOT_LINPUT
        LDD     4,X             ; GET NEXT TWO LETTERS
        SUBD    #'PU'           ; COMPARE
        BNE     NOT_LINPUT
        LDD     6,X             ; GET NEXT TWO LETTERS
        SUBD    #'T$'           ; COMPARE
        BNE     NOT_LINPUT
        LDAA    #C0h            ; RS232 DEVICE NUMBER
        LDAB    #4              ; VECTOR NUMBER TO HANDLE LINPUT$
        CLC
        RTS
NOT_LINPUT:                     ; NOT LINPUT$ - SO NOT PREPARED TO
        SEC                     ; HANDLE THE PROCEDURE
        RTS
```

NOTE: As the LANGUAGE vector code is called every time an OPL procedure is executed, it is important that the LANGUAGE vector code be as fast as possible. Clearly a device can handle as many procedures as necessary by just chaining the name matches or by searching a list of procedures. See the chapter on the language as to how OPL passes parameters to device procedure handlers and how devices pass back their results.

BOOTING

The system service DV$BOOT is responsible for loading and removing devices. The area in which devices are loaded in the operating system is one of the preallocated cells, known as the PERMANENT cell. This cell is the first allocated cell and has the unique property that it will never be moved by the allocator (i.e. it always has the same base address). This is obviously crucial as once the code has been relocated in memory it can no longer be moved, as all addresses have been converted from relative addresses to absolute addresses.

Organisers with 32kb or more RAM use the RAM from $0400 to $2000 ('low' RAM) for loading device code, in preference to the permanent cell. This means that while devices occupy less than 7k in total, the full 24k will be available for the user. Note that INFO will only show devices which are occupying high memory - i.e. any devices which have overflowed the 7k. For example with an RS232, a barcode reader, and a concise oxford spelling checker booted, only the spelling checker will be loaded into the permanent cell in high memory, so INFO will report DEVICES xxx%. The addresses DVA_BOT and DVA_TOP mark the lower and upper limits of low memory. DVA_TOP is increased as devices are loaded into low memory. DVA_BOT is initialised to $0400 at cold startup.

The operations performed by DV$BOOT are as follows:

1. Run the DV$CLER service, which will call the REMOVE service of each device currently in memory. DV$BOOT will then zero the PERMANENT cell. On Organisers with 32kb or more RAM it will also clear the low memory by setting DVA_TOP to DVA_BOT.
2. Scan the three slots for packs which are bootable.
3. Build a table of the priorities of each device in RTT_FF.
4. Scan the table for the highest priority device.
5. Grow the PERMANENT cell by the size of the device at the end of the PERMANENT cell (see below for LA/OS).
6. Load the code into the area just allocated.
7. Store the size of the code into the first word.
8. Store the slot number that the code was loaded from in the third byte of the device code.
9. Call DV$VECT to execute the INSTALL vector of the device.
10. Zero the priority in the table in RTT_FF.

11. Repeat the above until all priorities in the table are zero.

Organisers with 32kb or more RAM:

The device code is loaded into 'low' memory from $0400 to $2000, and the address of the byte after the code is stored in DVA_TOP. If there is not enough room in low memory, the device code and any further devices will be loaded into the permanent cell.

The DV$BOOT service is called by the operating system when a cold start is required. (On multi-lingual machines like the LZ, devices are booted on cold start before asking for a language and again afterwards when the language has been selected to enable the device to switch languages. This feature can be disabled - see foreign languages.)

Thereafter DV$BOOT is only ever called when the ON/CLEAR key is pressed in the main menu (i.e. at the top level). Thus to load a device into the operating system the device must be plugged into the machine and the ON/CLEAR key pressed until the main menu is reached. Then one further press will execute the DV$BOOT service and load the device driver. There is no problem in booting the device drivers any number of times.

To remove a device from memory, simply remove the device from the machine and repeat the above procedure (i.e. perform a boot). As the device is no longer present and as DV$BOOT always throws out all devices before performing the boot procedure the device will be effectively removed from memory.

NOTE: As DV$BOOT always throws all devices out of memory, no device can call the DV$BOOT service. In order for a device to call the DV$BOOT service it must copy a routine to call DV$BOOT into some safe portion of memory and then jump to that code. The code can then either get back to device, which will have been reloaded, through a known vector service or just simply return from where it came.

OPL programs, because they are running in a different area of memory can easily call the DV$BOOT service as follows:

```
REBOOT:
REM BOOTS ANY DEVICES INTO MEMORY
LOCAL I%,CODE$(4)
I%=ADDR(CODE$)+1 : REM SKIP THE SIZE BYTE OF THE STRING
POKEB I%,$3F     : REM THE SWI INSTRUCTION
POKEB I%+1,23    : REM THE DV$BOOT VECTOR NUMBER
POKEB I%+2,$39   : REM A RETURN INSTRUCTION
USR(I%,0)        : REM CALL THE MACHINE CODE
```

This procedure when called will execute the DV$BOOT service.

The following procedure will remove all devices from memory:

```
        OS      DV$CLER         ; run 'remove' vector of all devices
        LDX     DVA_BOT         ; reset low memory (harmless on non-LA/OS)
        STX     DVA_TOP
        LDX     #2000h          ; zero the permanent cell
        OS      AL$ZERO
```

From model LA onward, the user can reserve an area of low memory from $400 upwards as follows:

```
        LDA     A,$FFE8         ; TEST OS/VERSION

        ANDA    #7
        CMPA    #2              ; must be >= 2
        BLT     NOT_OK
        OS      DV$CLER         ; RUN 'REMOVE' VECTOR OF ALL DEVICES
        LDX     #2000h          ; AND KILL ANY DEVICES IN HIGH MEMORY
        OS      AL$ZERO         ; SO THAT THEY CANNOT BE ACCESSED AFTER THEIR
                                ; REMOVE VECTORS HAVE BEEN CALLED
        LDX     #SPACE_REQUIRED+0400h
        STX     DVA_BOT         ; SET BASE ADDRESS FOR FUTURE DEVICE LOADING
        STX     DVA_TOP         ; SET TOP= NEW BOTTOM
        ; RAM NOW AVAILABLE from $0400 - DVA_BOT - 1
```

2 AND 4 LINE MODELS

All machine code applications that run on the LZ will automatically be put into "2-line compatibility mode" as soon as anything is printed to the screen. The LZ will however be switched back to 4-line mode after returning. The device itself must put the machine into 4-line mode if required, by calling system service DP$MSET. Machine code applications which print to the screen should use the following method to implement 2-line and LZ compatible software. Whenever the application is entered the following should be carried out:

- Before calling any system services, check the value of DPB_MODE. If DPB_MODE = 0, take no further action and run in 2-line mode.
- If DPB_MODE = 1, call DP$MSET prevent the operating system switching to 2-line mode.

Note that machine code applications can simply check DPB_MODE rather than looking at the ROM id bytes because the mode will not change until anything is printed to the screen. However, OPL applications will have already been put into 2-line mode by the time they are running, so they must check the ROM id.

For example the following procedure will print "HELLO" on the 2nd line of a 2-line machine and on the 4th line of an LZ:


```
        LDAA    DPB_MODE
        BEQ     1$          ;branch to run in 2-line mode
        OS      DP$MSET     ;set to 4-line mode A=1
        OS      UT$DISP
        DB      12,23       ;clear screen and goto 4th line
        ASCIZ   "HELLO"
        RTS

1$:     OS      UT$DISP
        DB      12,10       ;clear screen and goto 2nd line
        ASCIZ   "HELLO"
        RTS
```

HARDWARE INTERFACES

The Organiser has three slots available in which hardware interfaces can be fitted. The TOP slot (slot 3) is the preferred slot for interfacing to external hardware as it has been especially designed for this purpose. However there is no reason why one of the two SIDE slots (slots 1 and 2) cannot be used. There are however a number of fundamental differences between the top slot and the side slots.

In general though the 16 way connectors from the Organiser to the outside world constitute a proprietary BUS and as such there are a number of rules from a hardware and software point of view with regard to interfacing on this BUS.

In the following sections the BUS components which are common to all slots are described followed by a section on the differences between the TOP slot and the SIDE slots.

BUS SIGNALS

SIGNAL NAME	DESCRIPTION
SD0 - SD7	Data Bus
SOE_B	Output Enable (active low)
SMR	Master Reset (active high)
SCLK	CLocK
SSS_B	Slot Select (active low)
SVCC	+ 5 Volts
SGND	0 Volts

This common set of signals comprise the Psion proprietary BUS. This BUS was designed to allow Psion's removable PACKS to be interfaced to the Organiser. In order to understand the following description of the BUS it is necessary to have read and understood the chapter on low level access to packs. All Psion's hardware interfaces consist of both a PACK and the hardware interface itself. The PACK is used to provide the DEVICE code which is loaded into the operating system, thereby providing the software interfacing to the OPL language and the operating system itself.

While it is possible to build hardware interfaces without a PACK on board, it is strongly recommended that this is not done. The PACK not only provides a mechanism for adding extra code associated with the hardware, it also provides a means whereby the code can tell that the interface is still plugged into the machine by reading the on board PACK.

SSS_B

This signal is the SLOT select signal and is used to select the hardware plugged into that slot. At no time should an external interface try to control any of the signals on the BUS until SSS_B goes low. There is obviously one SSS_B signal for each slot in the machine.

It is also mandatory that none of the control signals on the BUS should be changed while SSS_B is active. The control signals are SOE_B,SMR and SCLK. Thus to change a signal, SSS_B should be raised high, then the signal changed and finally SSS_B can be lowered again.

This signal should always be pulled up through a resistor to SVCC.

SOE_B

The SOE_B signal is used to select between the PACK on the interface and the external hardware. With SOE_B low (i.e. active) the PACK should output its contents onto the DATABUS. With SOE_B high then the hardware interface circuitry should control the DATABUS.

This signal should always be pulled down through a resistor to 0 volts

SMR

The SMR signal is used to reset the counters on the PACK to 0. It is also used in a number of combinations with SOE_B high. These are explained in a table later on.

This signal should always be pulled down through a resistor to 0 volts

SCLK

The SCLK signal is used to clock the counters on the PACK. It can also be used in a number of combinations with SMR high. These are explained in a table later.

This signal should always be pulled down through a resistor to 0 volts

SVCC

This is the +5 Volt rail to external hardware interfaces. It can be used to supply power to the hardware interface. Note that when the Organiser is switched off, this voltage will no longer be supplied. Instead a small amount of current can be drawn from the SSS_B signal as it is supplied from a different power rail. See Chapter 3 for more details.

SGND

This is the 0 Volt signal to the interface.

SD0-SD7

These 8 signals represent an 8 bit DATABUS between the hardware and the Organiser. Since it is possible to both read from and write to hardware interfaces, this bus is BI-DIRECTIONAL. However the rest state when not in use is always INPUT to the Organiser. i.e. The PORT2 direction register should be set to 0 so that all bits of PORT2 are input. If the direction of these bits are ever changed to OUTPUT they must be reset to INPUT before control is passed back to the operating system, as the operating system assumes that PORT2 is always configured as all INPUT.

All these signals are pulled down internally to the Organiser to 0 Volts. Thus if no interface is present in a slot the Organiser will always read a 0. This is used by the operating system to determine that a pack is not present in the slot.

BUS STATES

The following table represents the rules by which the interface should respond to various states of the BUS control signals. In the table, HIGH stands for +5 Volts, LOW stands for 0 Volts and X stands for don't care.

STATE	SSS_B	SOE_B	SMR	SCLK	DESCRIPTION
0	HIGH	X	X	X	All interfaces deselected.
1	LOW	HIGH	HIGH	X	128K PACK device segment select.
2	LOW	HIGH	LOW	X	Hardware interface select.
3	LOW	LOW	HIGH	HIGH	Ram PACK ID 1 select.
4	LOW	LOW	HIGH	LOW	Ram PACK ID 0 select.
5	LOW	LOW	LOW	X	PACK read select.

When the operating system selects a slot using the PK$SETP service, it will place the control signals in a number of these states. However the top slot is treated slightly differently, in that the operating system will not support 128K packs or RAM packs. Thus the following states can occur for the various slots:

STATE	SLOT 3	SLOTS 1 & 2
0	Yes	Yes
1	No	Yes
2	No	No
3	No	Yes
4	No	Yes
5	Yes	Yes

STATE 0

SSS_B - HIGH
SOE_B - don't care
SMR - don't care
SCLK - don't care

This state is the rest state for all interfaces. When SSS_B is high no interface should be trying to control the databus or any of the control signals.

STATE 1

SSS_B - LOW
SOE_B - HIGH
SMR - HIGH
SCLK - don't care

This state is used by the operating system to select the 16K segment in 128K PACK devices. Note that PK$SETP selects segment 0 every time it is called regardless of the device fitted in the slot. As the operating system does not support 128K PACK in the top slot this state can be used to select devices in a hardware interface to be used in the top slot.

STATE 2

SSS_B - LOW
SOE_B - HIGH
SMR - LOW
SCLK - don't care

This state is used to write data to either packs or RAM packs in the slots. However if the RDWRT bit in the ID byte of the pack is clear then the operating system will never place state 2 on the control lines.

This state is the only state that can be used safely to select other devices than the PACK on interfaces intended for use in the side slots.

STATE 3

SSS_B - LOW
SOE_B - LOW
SMR - HIGH
SCLK - HIGH

This state is used by the operating system to select the RAM pack hardware ID 1. Note that PK$SETP uses this state to determine if a RAM pack is plugged into a slot. If so, it will output an ID byte of 1 in response to this state. Note that packs output byte 1 of the EPROM in response to this state.

As the operating system does not support RAM packs in the top slot this state can be used to select devices in a hardware interface to be used in the top slot.

STATE 4

SSS_B - HIGH
SOE_B - don't care
SMR - don't care
SCLK - don't care

This state is used by the operating system to select the RAM pack hardware ID 0. Note that PK$SETP uses this state to determine if a RAM pack is plugged into a slot. If so, it will output an ID byte of 1 in response to this state. Note that Packs output byte 0 of the EPROM in response to this state.

As the operating system does not support RAM packs in the top slot this state can be used to select devices in a hardware interface to be used in the top slot.

STATE 5

SSS_B - HIGH
SOE_B - LOW
SMR - LOW
SCLK - don't care

This state is used by the operating system to read data from the PACK circuitry on interfaces. The byte, corresponding to the currently selected address on the PACK counters, is output on the databus in response to this state.

EXAMPLE

Take as an example the BARCODE interface for the top slot, as supplied by Psion.

The interface has two discrete interface circuits:

1. The PACK circuitry.
2. The BARCODE input signal buffer circuitry.

State 5 is used to enable the EPROM in the PACK circuitry to output its contents on the databus.

States 1 and 2 are used to enable the BARCODE input signal buffer circuitry. I.e.

SSS_B - low,
SOE_B - high,
SMR - don't care,
SCLK - don't care.

Thus to select the BARCODE PACK circuitry the following code fragment can be used:

```
SELECT_PACK:
        CLRA
        LDAB    #PAKD
        OS      PK$SETP         ; THE PACK IS NOW SELECTED
        ; THE CONTROL LINES ARE IN STATE 5
```

To select the BARCODE input signal buffer circuitry the following code fragment can be used:

```
SELECT_BARCODE:
        CLRA
        LDAB    #PAKD
        OS      PK$SETP           ; SELECT PACK AS NORMAL
```

```
        OIM     #CS3,POB_PORT6      ; DESELECT THE SLOT
        OIM     #OE,POB_PORT6       ; SET SOE_B HIGH
        AIM     #~CS3,POB_PORT6     ; SELECT THE SLOT AGAIN
        ; BARCODE HARDWARE NOW SELECTED
        ; THE CONTROL LINES ARE IN STATE 1 OR 2
```

NOTE: Before control can be returned to the operating system, STATE 5 must be reestablished on the control lines. This can be achieved as follows:

```
DESELECT_BARCODE:
        OIM     #CS3,POB_PORT6      ; DESELECT THE SLOT
        AIM     #~OE,POB_PORT6      ; SET SOE_B LOW
        OIM     #~CS3,POB_PORT6     ; SELECT THE SLOT AGAIN
```

NOTE: Control lines should only ever change state when the control lines are in STATE 0, i.e. the interface is deselected.

SYSTEM SERVICES

DV$BOOT

Will boot all devices plugged into the Organiser.

Any errors occurring during the boot are reported directly. However the screen is saved before the message is displayed and restored immediately afterwards.

DV$LOAD

This service will load code from a pack which is in the Psion relocatable object format. It loads the code into memory and then applies any fix-ups that are required, thus relocating the code to the address as specified.

The routine assumes that the correct slot has already been selected with the PK$SETP service.

DV$VECT

This service will search the devices in the PERMANENT cell for a device whose device number matches the given number.

Then the service checks that the passed vector number is not greater than the maximum vector number supported by the device. If it is, then an error is returned. Otherwise the appropriate vector is loaded from the device vector table and a JMP is done to the vector.

DV$VECT passes the X register and the scratch register UTW_S0 through to the vectored routine, so that these may be used to pass parameters to the device vector routine. It is up to the device to specify what is passed and what is returned. DV$VECT returns the same things as the vectored routine.

DV$LKUP

This service will call the LANGUAGE vector of all devices in memory. If a device signals that it is prepared to handle the given procedure name then the service will return the device number and the vector number of the code to handle the procedure.

DV$CLER

This service calls the REMOVE vector for all devices currently in memory.

It is usually called by DV$BOOT just before zeroing the permanent cell in preparation to booting.

EXAMPLE

This is an example of a device to allow the operating system services to be called from OPL procedures. The device is a procedure called SWI%: and it takes as its argument an integer specifying which service should be invoked.

Most functions also require values in the A,B(D) or X register as well and so the function requires 2 global variables X% and D% to be declared by the OPL procedure. Any values to be passed in UTW_S0 or UTW_S1 can be set with POKEW before calling SWI%:.

If carry is clear after the call to the operating system SWI%: will return 0 and if it is set, SWI%: will return -1 and D% will have the error number.

If SWI%: signals success D% and X% will be set directly from the machine registers D and X.

NOTE: The D register is actually made up of the A register and the B register so that D = A*256 + B. The global variable D% mimics this. So if a routine requires A = 1 and B = 3 this can be passed as D = 1*256+3. On return the values of A and B can be determined from D% as follows:

A% = (D%/256)
B% = (D% AND 255)

See the full code listing for the routine.

COMMS LINK

INTRODUCTION

COMMS LINK driver code is interrupt driven and handles XON/XOFF and hardware handshaking at the lowest level. It also parity checks all incoming data if parity is enabled.

As all serial I/O is interrupt driven, interrupts CANNOT be disabled (as with the keyboard). All calls to the operating system via the SWI instruction are intercepted by the driver code once the port has been opened, to make data pack access transparent (and possible!) while the port is open. It also ensures that the port is active and not affected by various power saving techniques in the operating system while it is open.

ALTERED SYSTEM SERVICES

The function of the following operating system services are altered when the port is opened; PK$SETP, PK$PKOF, BZ$TONE, BZ$ALRM and BZ$BELL. As all SWI's and interrupts are intercepted directly, and the old handlers are not chained to, any other application that re-vectors the same services will not function correctly while the port is open. The old handlers are saved only when the comms pack is booted during the install code but are restored every time the port is closed. Any application booted in after the comms pack, which re-vectors any service used by the comms pack, will not function correctly even when the port has been closed as the original handler will be restored.

- PK$SETP - This function will first wait for the correct time to elapse from the last character being sent and then switch off the port. From this point the port will no longer be able to receive characters. The operating system function will then select the pack in the normal way, and the next call made to a COMMS LINK service will switch the port on again.
- PK$PKOF - This function will do nothing while the port is open.
- BZ$TONE, BZ$ALRM, BZ$BELL - These three functions will all just do a beep regardless of any parameters passed while the port is open because interrupts cannot be disabled.

MACHINE CODE INTERFACE

The interface to the COMMS LINK driver code is via a block of fixed address memory of 40 bytes long starting at dvt_spar ($214F). This memory is divided into two areas: a table of variables which set various I/O parameters and an entry point for calls to the driver code. All these variables and a macro for calling the driver code via this entry point are defined below.

VARIABLES

The following variables can be set set to control the function of the COMMS LINK driver code. However, these variables are also set by the SETUP option in the COMMS menu and by the LSET OPL function, so generally their values must be preserved.

After any of these variables have been altered a call to rs$setvars must be made to set up various derived variables.

None of these variables should be altered while the port is open.

SINGLE BYTE VARIABLES

Address	Name	Range (default values bold)		
$2150	rsb_baud	0-9	50,75,110,150,300,600, 1200,2400,4800,**9600**	Baud rate (transfer speed)
$2151	rsb_parity	0-4	**NONE**, ODD, EVEN, MARK, SPACE	Data, Parity, Stop Bit settings: transfer frame setup
$2152	rsb_bits	0-1	0 - 7 BIT DATA 1 - **8 BIT DATA**	

$2153	rsb_stop	0-1	0 - **1 STOP BIT** 1 - 2 STOP BITS	
$2154	rsb_hand	0-7	NONE,**XON**,RTS,XON+RTS, DTR,DTR+XON,DTR+RTS,ALL	Handshake mode
$2155	rsb_proto	0-2	**NONE**, XMODEM, PSION	File transfer protocol (off by default)
$2156	rsb_echo	0-1	LOCAL, **HOST**	Echo (Terminal emulation only)
$2157	rsb_width	0-250	**NONE**, 1-250	buffer width (forced line length)
$2158	rsb_timout	0-255	**NONE**, 1-255 seconds	LPRINT timeout

3 BYTE VARIABLES:

All contain a length byte (0-2) and two bytes of data

Address	Name	default	
$2159	rst_reol	13,10 <CR><LF>	Receive End Of Line character(s)
$215C	rst_reof	26 <SUB>	Receive End Of File character(s)
$215F	rst_rtrn	empty	Receive Translate character(s): 1 - remove, 2 - replace first by second
$2162	rst_teol	13,10 <CR><LF>	Transmit End Of Line character(s)
$2165	rst_teof	26 <SUB>	Transmit End Of Files character(s)
$2168	rst_ttrn	empty	Transmit Translate characters: 1 - remove, 2 - replace first by second

SYSTEM VARIABLES

It is strongly suggested to leave these unchanged.

Address	Name	Range	
$216B	rsb_off_del	1-255 default=3	Time to off delay in characters
$216C	rsb_xoff_del	1-255 default=7	Time to off delay with XON/XOFF
$216D	rsb_tcon_val	0-255	Time constant value for timer 2 baud rate generation
$216E	rsb_off_ticks	0-255	No. of ticks for baud rate dependent Tx off delay
$216F	rsw_off_tcon	0-$FFFF	Time constant for single tick TX off delay (word)
$2171	rsb_sec_timer	0-255	General purpose decrement to zero second timer (decremented once every second until zero)
$2172	RESERVED WORD		
$2174	rst_entry_point		Entry point for assembler interface, contains a jump instruction to the driver (3 bytes).

The general purpose timer rsb_sec_timer is used by the higher level functions such as LPRINT to implement time outs so be warned!

CALLS TO THE DRIVER

All calls to the COMMS LINK driver code are made via the macro "RS" which is defined below:

```
    ;
    ; RS - Macro for assembler interface to COMMS LINK drivers
    ;
.macro    RS  function
          jsr rst_entry_point
.byte     function
.endm
```

Before using this macro a check must be made that the COMMS LINK code has been booted as follows:

```
          ldx  #opl_name          ; choose an OPL function unique to COMMS-LINK
          os   dv$lkup            ; ask O/S if it's booted
          bcs  not_booted         ;ok to call RS$ routines now

opl_name: .ascic"XFEOF"
```

DRIVER FUNCTIONS

The following functions are available:

0	RS$open	Open the COMMS LINK channel
1	RS$close	Close the COMMS LINK channel
2	RS$putchar	Put a character to the RS232 port
3	RS$getchar	Get a character from the RS232 port
4	RS$flush	Flush the receive buffer
5	RS$setvars	Set the COMMS LINK variables up after a change
6	RS$lprint	Print a string
7	RS$linput	Input a string
8	RS$licon	Link layer connect call
9	RS$lidis	Link layer disconnect call
10	RS$liput	Link layer put a frame call
11	RS$liget	Link layer get a frame call

Example to open the RS232 port for reading and writing:

```
        clrb              ; Open for reading and writing
        RS   RS$open
        bcs  error        ; Deal with error
        ...               ; Rest of code
```

In general all functions indicate an error condition by returning with the carry flag set and the appropriate error code in the B register.

RS232 CONTROL SIGNALS

The COMMSLINK hardware provides the following RS232 control signals on the port POB_PORT2 (address $03).
All bits are readable, writing to input bits is to be avoided.

Name	Top slot pin	Direction	PORT2 bit no	Set if:
RTS	5	input	2	correspondent busy
CTS	6	output	0	organiser busy
DSR	4	input	1	correspondent busy
DTR	-	linked to DSR	-	DSR set

Note that DTR (normally an output) is merely linked to DSR and can not be driven by a program.

PROGRAMMING

CONSIDERATIONS

In between a call to RS$OPEN and a call to RS$CLOSE, programmers should look out for the following operating system routines:
Routines which access packs:

- all FL$ routines
- all PK$ routines except for PK$PKOF
- DV$BOOT, DV$LOAD
- any other routines which indirectly access packs, such as calls to OPL, DV$VECT, TL$XXMD etc.

These turn off the RS232 port, until the next call to COMMS LINK. This is normally no problem, unless a program tries to access POB_PORT2 after a call to one of the above routines but before another COMMS LINK call. The RS232 port should then be turned on by calling RS$OPEN again, as follows:

Example: no problem here

```
        os FL$OPEN
        ...                     ; RS232 now off
        jsr rst_entry_point
        Byte RS$getchar        ; or similar COMMS LINK function
        ...
        tim #cts,POB_PORT2      ; test the CTS line
```

Example: this is a BUG

```
        Os PK$SETP
        ...                     ; RS232 now off
        Tim #CTS,POB_PORT2      ; test the CTS line
```

Example: corrected version of above

```
        Os PK$SETP              ; Re-open the RS232 port, to switch it on
        ldab #opened_with       ; use value originally supplied to RS$OPEN
        andb #^XFE              ; but with bit 0 clear to indicate that
        jsr rst_entry_point     ; the handshaking state is not to be reset
        Byte RS$open
        ...
        Tim #CTS,POB_PORT2      ; now OK to access port
```

Other routines whose behavior is altered while the RS232 port open:

- PK$PKOF - has no effect
- BZ$ routines - these all give a strange beep
- KB$ routines do not switch the packs off to save power

Do not call BT$SWOF without first closing the RS232 port with RS$CLOSE

The keyboard interrupt service routine is altered so that interrupts remain enabled while the keyboard is being scanned. This gives RS232 interrupts a higher priority than keyboard interrupts.

PROGRAMMING EXAMPLES

Checking that COMMS LINK is installed

First check that the COMMS LINK software has been loaded - i.e. COMMS is in the top-level menu. If COMMS LINK is not installed, any program which calls the machine code interface routines provided by COMMS LINK will crash.

This check can be performed in OPL as follows:

```
        commsin%:
        rem return true if COMMS LINK installed
        onerr iserr::
        xfeof:          :rem try a harmless COMMS LINK function
        iserr::
        return err<>203 : rem not installed if "missing proc" error
```

or in machine code:

```
                bsr check_present
                bcs not_present
                bra is_present

check_present:  bsr 1$          ; PC relative LDX #XFEOF
                .ascii "XFEOF"  ; leading count byte string
           1$:  pulx            ; PC = address of "XFEOF"
                Os dv$lkup      ; if XFEOF not there, no COMMS LINK
                rts
```

Opening and closing the RS232 port

Before accessing the RS232 port, the program must first call the COMMS LINK machine code interface routine RS$OPEN. This indicates that the RS232 port is now in use.

Example:

```
                clrb            ; Open for reading and writing
                jsr rst_entry_point
                Byte RS$open
                bcs error       ; Deal with error
                ...             ; Now free to access the port
```

RS$CLOSE should be called when the RS232 port is no longer needed.

```
        ; bit masks for RS$close
        turn_off_immed= 1       ; switch off immediately after close
                                ; else leave port on
        fail_busy= 2            ; fail with carry set if paused by XOFF
                                ; else ignore XOFF state, and close immediately

        ; Example where port not closed while XOFF handshake

        close_type = fail_busy  ; dont close if busy handshaking,
                                ; leave port switched on

        wait_busy:      ldab #close_type
                        jsr  rst_entry_point
                        Byte RS$close
                        bcs wait_busy   ; wait until XON

        ; Straight close

        close_type = 0  ; ignore XOFF state,
                        ; leave port switched on

                        ldab #close_type
                        jsr rst_entry_point
                        Byte RS$close
```

Using handshaking lines

```
        ; Bit masks
                CTS= 1
                dsr= 2
                rts= 4
        ;Set CTS to indicate ORGANISER II busy

                oim #CTS,POB_PORT2


        ;Clear CTS to indicate ORGANISER II ready for input

                aim #CTS,POB_PORT2      ; not (1) = $FE


        ;Wait on both RTS and DSR

        wait:   Tim #rts!dsr,POB_PORT2
                bne wait
```

Example: OPL/Machine code for controlling the RTS line directly:

```
        global rtshigh%(2),rtslow%(2),a%
        rtshigh%(1)=$71fe
        rtshigh%(2)=$0339
        rtslow%(1)=$7201
        rtslow%(2)=$0339

        rem to assert (raise) the rts line
        a%=usr%(addr(rtshigh%()),0)

        rem to de-assert (lower) the rts line
        a%=usr%(addr(rtslow%()),0)
```

FUNCTION REFERENCE

All all function calls are made via the macro "RS", see above.

RS$open

Function Number: 0

Input parameters: B register: Mode to open.

- Bit 0 - Set if port in TX only mode.
- Bit 1 - Set to enable break key error inhibiting.
- Bit 2 - Set if the TX paused state is to be cleared.

Output values: Carry set if error, error number in B

Registers corrupted: All

Opens the RS232 channel and initialises the hardware, also sets various modes of operation for the port depending on the value passed in B.

If a pack access is made while RS232 is open this access will be delayed until any stray characters have been delt with. Any subsequent calls to RS_putchar or RS_getchar will switch the port on again. If a call to RS$open is made while the port is already open the call is ignored.

Errors:

- ER_DV_NP - Device not present
- ER_GN_BL - Battery too low

RS$close

Function Number: 1

Input parameters: B register: Mode to close.

- Bit 0 CLEAR port closed and is turned off later by pack access or key scan
- Bit 0 SET port closed and turned off
- Bit 1 CLEAR host paused state is ignored
- Bit 1 SET port fails to close if the host is busy (paused by an XOFF)

Output values: Carry set if can not close port because host is busy

Registers corrupted: All

Closes the RS232 port. Carry is set if failed to close due to host busy

Errors: none

RS$putchar

Function Number: 2

Input parameters: A register: Character to be put to buffer

Output values: Carry set if error or busy
B clear or error number

Registers corrupted: B, CCR

Transmits the passed character. Returns with the carry set if an error or the port was busy, else the carry is clear. B is clear if no error.

Errors:

- ER_DV_CA - Invalid device call - Port not open
- ER_RT_BK - Break Key
- ER_GN_BL - Battery too low

RS$getchar

Function Number: 3

Input parameters: None

Output values: A register - Next character from receive buffer
B clear or error number
Carry set if error or busy

Registers corrupted: A, B, CCR

Gets the next character from the receive buffer. Returns with carry set if error or no characters in the buffer, else the next character from the buffer in the A register B is clear.

Errors:

- ER_DV_CA - Invalid device call - Port not open
- ER_GN_RF - Device read fail - Parity or overrun
- ER_RT_BK - Break Key
- ER_GN_BL - Battery too low

RS$flush

Function Number: 4

Input parameters: None

Output values: None

Registers corrupted: CCR

Flushes the receive buffer.

Errors: none

RS$setvars

Function Number: 5

Input parameters: None

Output values: None

Registers corrupted: All

Sets the derived COMMS LINK variables after a change to the setable COMMS LINK variables.

Errors: none

RS$lprint

Function Number: 6

Input parameters: X register - Pointer to text to print
B register - Length of string to print

Output values: B register - Error code if any

Registers corrupted: All

Opens the RS232 port for output only, then writes the passed string to the port applying all the translates, timeouts etc. specified.

Errors:

- ER_GN_BL - Battery too low
- ER_GN_WF - Device write fail (Timeout)
- ER_RT_BK - Break Key
- ER_DV_NP - Device not present

RS$linput

Function Number: 7

Input parameters: A register - Timeout in seconds (0=no timeout)
B register - Number of characters to receive
X register - Address of buffer to place characters

Output values: B register - Error code if any
A register - Number of characters received

Registers corrupted: All

Opens the RS232 port for output and input and then reads the passed number of bytes into the passed buffer applying all the translates, timeouts etc. specified.

Errors:

- ER_GN_BL - Battery too low
- ER_GN_RF - Device read fail (Timeout)
- ER_RT_BK - Break Key
- ER_DV_NP - Device not present

RS$licon

Function Number: 8

Input parameters: None

Output values: B register - Error code if any

Registers corrupted: All

Attempts to establish a logical link with the correspondent link entity. Wait for a suitable acknowledgment.

Errors:

- ER_RT_F0 - a link already exists (local error)
- ER_GN_RF - timeout trying to get a connection
- ER_DV_NP - Device not present
- ER_GN_BL - Battery too low

RS$lidis

Function Number: 9

Input parameters: None

Output values: B register - Error code if any

Registers corrupted: All

Disconnect the logical link with the correspondent link layer. Harmless if the link is already disconnected.

Errors:

- ER_RT_FC - no link in existence
- ER_GN_BL - Battery too low

RS$liput

Function Number: 10

Input parameters: D register - Length of buffer to send
X register - Address of buffer to send

Output values: B register - Error code if any

Registers corrupted: All

Send data in buffer to the correspondent. The data length must be >=0 and <=MAXILEN. Waits for a suitable acknowledgment. (MAXILEN currently 260)

Errors:

- ER_RT_FC - a link does not exist (local error)
- ER_GN_RF - a re-transmission threshold expired
- ER_LX_ST - len exceeds MAXILEN (no data transferred)
- A server disconnection reason code, e.g. "no disk space"
- ER_GN_BL - Battery too low

RS$liget

Function Number: 11

Input parameters: D register - Number of bytes to get
X register - Address of buffer to put bytes

Output values: B register - Error code if any
D register - Number of bytes placed in buffer if OK

Registers corrupted: All

Wait for a frame to arrive from the physical layer. Returns number of bytes placed in buffer if all OK

Errors:

- ER_RT_FC - a link does not exist (local error)
- ER_GN_RF - a timer expired
- ER_LX_ST - data length exceeds Len (1st Len bytes in buffer)
- A file server disconnection reason code, e.g. drive door open.
- ER_GN_BL - Battery too low

Psion Link Protocol

Link Layer

In its simplest form data is sent over the link using the 'link layer', an error correcting protocol which uses data packets. Each packet has the following format (in hex):

16 10 02 <CHAN> <TYPE> <DATA> 10 03 <CRC>

The first three bytes are always 16 10 02.

The next byte, <CHAN>, is the channel number. It is not used and must always equal 1. It is compared to the value stored at $2173, which is set to 1 when the comms link is loaded and not changed afterwards. It may be that some other communications devices use a different channel number.

The next byte, <TYPE>, contains the type of packet in the top 5 bits, and also a sequence number in the bottom 3 bits. The sequence number is a number that each data packet has to make sure that they are not mixed up. Since the sequence number increases with each new data packet, missing packets can be detected.

There are 4 types of packet:

<TYPE>=0	ACKNOWLEDGE	Signals proper reception of a data packet. The sequence number of the acknowledge is the same as that of the data packet received. If a packet is received with the wrong sequence number, an Acknowledge is sent of the last correct data packet, so this is also used as a request for the next packet. Contains no data.
<TYPE>=1	DISCONNECT	Signal that communication is to end. When sent to the Psion, the data should contain the error number. If the Psion sends one then it has no data. The sequence number is 0.
<TYPE>=2	LINK REQUEST	Requests a connection. Used at the start of a session to test two-way communication. Sequence number is 0 and it contains no data.
<TYPE>=3	DATA	A data packet. The sequence number is one more than that of the previous data packet (modulo 8) or 1 if it is the first data packet of the session.

The bytes 10 03 follow the data to signal the end of the packet.

The <CRC> is a 16-bit checksum for all the bytes between the header 16 10 02 and the footer 10 03. See below.

To make sure that sending the bytes 10 03 in the data will not prematurely end the packet, an escape mechanism is used. Every 10 byte in the data (or chan/type bytes) will be sent twice over the link. The receiving end then replaces each 10 10 by a single 10, and will not end the packet if it is followed by a 03. The extra 10 is not used in generating the CRC however.

The maximum length of a packet is 260 bytes.

CRC

Explaining CRC theory is beyond this manual. However CRCs can be computed by knowing the following values:

```
CRC(01)=C1C0      CRC(10)=01CC
CRC(02)=81C1      CRC(20)=01D8
CRC(04)=01C3      CRC(40)=01F0
CRC(08)=01C6      CRC(80)=01A0
```

The CRC of any single byte can be computed by EOR-ing the values above together for each set bit in the byte. If ChCl is the CRC for a sequence of one or more bytes, then the CRC of that sequence followed by a byte B will be CRC(B eor Ch) eor (Cl*0100).

For example, let's calculate CRC(01 10):

```
CRC(01) = C1C0.
CRC(01 10) = CRC(10 eor C1) eor C000
           = CRC(D1) eor C000
           = CRC(80) eor CRC(40) eor CRC(10) eor CRC(01) eor C000
           = 01A0 eor 01F0 eor 01CC eor C1C0 eor C000
           = 005C
```

See also:

- machine code used in the CommsLink driver
- Visual Basic example

The Link Protocol

Connecting

When CL is running on the PC, it continuously sends out Link Request packets. When the Psion is ready to start communicating it first starts up a link session in the following manner:

1. Waits to receive a Link Request. Times out if none received.
2. Sends a Link Request.
3. Waits to receive an Acknowledgment (seq.nr. 0). Times out if none.

After that the link is considered to be active and working, and the sequence numbers are reset.

Specifically, the packets involved are:

```
Link Request: 16 10 02 01 10 10 10 03 00 5C
 Acknowledge: 16 10 02 01 00 10 03 01 90
```

If in step 3 a disconnect request is received, it should contain an error byte as data. Receiving a data packet will cause the Psion to return a data acknowledgement, and then it also considers the link to be active.

The start-up sequence above is performed before every data transfer. It can also be called specifically by using the XLCON: command, or calling rs$licon in machine code.

Sending

When sending a sequence of data packets, each packet is numbered 1,2,..,6,7,0,1,... etc. After each packet is sent, a reply is expected. What happens next depends on the type of reply:

- An acknowledgement with the correct sequence number: The packet has been successfully sent, so now further packets can be sent if necessary.
- An acknowledgement with the wrong sequence number, or
- A packet with an incorrect CRC, or
- A data packet: The same data packet will be re-sent. If this happens too often, a 'Device Read' error occurs.
- A link request: A 'Device read' error occurs.
- A disconnect packet: The error in the packet is raised.

A single packet can be sent using the XLPUT:(a$) command, or RS$liput in machine code.

Receiving

When receiving a sequence of data packets, each packet should numbered 1,2,..,6,7,0,1,... etc. After each packet is received, a reply is expected. When the Psion is waiting for a data packet, what happens depends on the type of packet it receives:

- A data packet with the correct sequence number: An acknowledgement is sent for this packet. The data is stored.
- A data packet with the wrong sequence number, or
- A packet with an incorrect CRC, or too long: An acknowledgement is sent with the previous sequence number, i.e. the sequence number of the last received packet.
- A link request, or
- A disconnect packet: The error in the packet is raised.
- An acknowledgement: It is simply ignored.

A single packet can be received using the XLGET$: command, or RS$liput in machine code.

Note that the sequence numbers used when receiving packets are independent from those used when sending packets.

Disconnecting

When the link session is finished, the link should be deactivated. This is done by calling the XLDIS: command, or RS$lidis in machine code. This will send a disconnect packet, and close the link session. The disconnect packet has no data when sent from the Psion, but those sent by CL will contain a single byte which is the error number.

Higher layers

The link layer is simply an error correcting protocol for sending small snippets of data. Built on top of this is a protocol for sending and receiving files, file access commands etc.

The Psion Series 3 and 5 first have an NCP protocol layer above the link layer. NCP is a method of sending data over different channels. This is not implemented on the series II, though there seems to be a simple channel byte in every packet which is not used at all.

The file protocol layer generally works like this:

1. The startup sequence is performed first.
2. The Psion sends a word like 'FTRAN' in a data packet (which is acknowledged in the link layer as usual).
3. The CL program loads the FTRAN overlay, the program in S_FTRAN.EXE
4. The PC sends back an empty data packet as reply (which is again acknowledged in the link layer as usual).
5. The Psion sends a command to the FTRAN program in a data packet (+ack).
6. The PC sends data packet back in reply (+ack).
7. Repeat 5 and 6 till everything accomplished.
8. The Psion sends a disconnect packet.

It now remains to be explained what overlays there are, what commands they accept and what data they return.

FILE overlay

This overlay handles all remote file access (i.e. XFOPEN, XFCLOSE etc). There are five types of data packet that the Psion sends to CL/FILE. The data packet starts with a byte between 00 and 04 to indicate the call type and is followed by the parameters.

CALL TYPE		PARAMETERS	DESCRIPTION
XFOPEN	00	mode type name	Opens a remote file where mode/type are bytes and name is a string, exactly like the XFOPEN: command.
XFCLOSE	01		Closes current remote file, and exits the FILE overlay.
XFPUT	02	datastring	Puts the data at the current file position.
XFGET$	03	length	Gets (at most) length bytes from the current file position which are returned in the reply data packet from CL/FILE.
XFPOS	04	mode position	Sets the current file position. Mode is a byte but position is a string representation of the new file position. The reply data packet contains a string representation of the old file position.

For example, suppose we run the following short program:

```
TEST:
XFOPEN:("HOMER.TXT",1,1)
XFPUT$:("Doh"+CHR$(33))
XFCLOSE:
```

then the following list is the complete exchange between the Psion and the PC, assuming no errors occur:

```
Start link layer session:
      PC:  Link Request:  16 10 02 01 10(10)10 03 00 5C
      Ps:  Link Request:  16 10 02 01 10(10)10 03 00 5C
      PC:  Acknowledge:   16 10 02 01 00 10 03 01 90
Then the FILE overlay is loaded:
      Ps:  FILE overlay:  16 10 02 01 19 46 49 4C 45 10 03 2D BE
```

```
      PC:  Acknowledge:   16 10 02 01 01 10 03 C0 50
Reply:
      PC:  No Data:       16 10 02 01 19 10 03 C0 5A
      Ps:  Acknowledge:   16 10 02 01 01 10 03 C0 50
File is opened:
      Ps:  Open command:  16 10 02 01 1A 00 01 01 48 4F 4D 45 52 2E 54 58 54
                          10 03 39 8B              H  O  M  E  R  .  T  X  T
      PC:  Acknowledge:   16 10 02 01 02 10 03 80 51
Reply:
      PC:  No Data:       16 10 02 01 1A 10 03 80 5B
      Ps:  Acknowledge:   16 10 02 01 02 10 03 80 51
Send data:
      Ps:  Data Doh!:     16 10 02 01 1B 02 44 6F 68 21 10 03 A1 DE
                                        D  o  h  !
      PC:  Acknowledge:   16 10 02 01 03 10 03 41 91
Reply:
      PC:  No Data:       16 10 02 01 1B 10 03 41 9B
      Ps:  Acknowledge:   16 10 02 01 03 10 03 41 91
End signal:
      Ps:  End packet:    16 10 02 01 1C 01 10 03 98 C0
      PC:  Acknowledge:   16 10 02 01 04 10 03 00 53
Reply:
      PC:  No Data:       16 10 02 01 1C 10 03 00 59
      Ps:  Acknowledge:   16 10 02 01 04 10 03 00 53
End link layer session:
      Ps:  disconnect:    16 10 02 01 08 10 03 00 56
```

FTRAN overlay

This is the most commonly used one. It handles all standard file transfer. There are four types of data packet that the Psion sends to FTRAN. The data packet starts with a byte between 00 and 03 to indicate the call type and is followed by the parameters.

CALL TYPE		PARAMETERS	DESCRIPTION
Command	00	command filetype filename	Tells FTRAN to perform a command on the remote file. The filetype is 00 for ODB files, 01 for OPL files, 02-0F for block files. It is 00 if the type is irrelevant. The available commands are listed below.
End	01		Exits the FTRAN overlay.
PutData	02	datastring	Passes data to FTRAN for saving on disk.
GetData	03	length	Gets (at most) length bytes from the current file position which are returned in the reply data packet from CL/FILE. The length requested is usually FE.

The commands available are:

Recv	00	Warns CL/FTRAN that the Psion is about to request data from the file of the given name. This returns the length and type of the PC file. Further details below.
Send	01	Warns CL/FTRAN that the Psion is about to send data to the file of the given name. Any existing file of that name will be overwritten without warning.
Exists	04	Gives an error if the file does not exist.

When sending/receiving ODB files, each record is sent in a separate packet. No data is returned after the recv command.

When sending/receiving block files, the data is split in a separate packets, generally about FE bytes long. The data returned after the recv command is the length of the block in the file, followed by the Psion file type (between 82 and 8F).

When sending/receiving OPL files, the data is split in a separate packets just like block files. The line ends are indicated by zero bytes, and it is FTRAN that translates it to/from CR/LF when writing to disk. The data returned after the recv command is the length of the block in the file, the Psion file type (81 - although the file is stuctured like an OB3), followed by 0000 and the length of the OPL part of the procedure block file. Note that using file type 83 works fine if used with the COMMS menu, but produces pack corruption when used with the OPL XTRECV: command.

BOOT overlay

This overlay handles the loading of BOOT code, and the XMLOAD: command. After the BOOT overlay is loaded, it expects to be sent a data packet containing the name of the boot file to load. The reply packet contains the length of the code.

The Psion then sends a packet containing the address to load the code. In reply the first data packet contains the first part of the code, ready to be stored in memory. When the Psion next sends a packet containing just a zero byte, the next code packet is sent. This is repeated as often as necessary.

When the end of the code has been reached, a disconnect packet with a zero error is returned by CL/BOOT.

Note that fixing up the relocatable addresses in the code is done by CL/BOOT.

Instead of using the BOOT option in the menu, the XMLOAD: command can be used. XMLOAD:(addr%,len%,name$) loads the boot code from remote file name$ to address addr%, but rejects it if the code exceeds length Len%.

EXIT overlay

This overlay exits the CL program completely (by stuffing q and y in the keyboard buffer of the PC). It has no other functions. It can therefore be used by using the following sequence of commands:

```
XLCON:
XLPUT:("EXIT")
XLDIS:
```

PLAN and PLANM overlays

These overlays are used by the spreadsheet program. When you choose the FILE/IMPORT or FILE/EXPORT menu options, the Psion requests the PLANM overlay which returns the file type menu, and the Psion disconnects again. The file type menu returned is 19h bytes long and is as follows:

```
03 "WKS" 0001
03 "WK1" 0002
03 "WR1" 0003
03 "DIF" 0004
00
```

It is in the correct format for the MN$DISP (SWI 50h) system service.

The PLAN overlay handles the transfer and conversion of the spreadsheet files. To export a file, the Psion first sends it a packet of which the first byte is between 81h and 84h depending on the file type chosen, and the rest contains the file name. The spreadsheet data follows; first a packet containing the dimensions of the sheet, then a packet for each non-empty cell, and finally a packet containing just FE. The reply that PLAN gives each time is a packet containing 00, except that the FE is answered with FE 00.

A cell is encoded in this format:

Column	Byte containing column of cell (0 is column A)
Row	Byte containing row of cell (0 is row 1)
Type	Contains 22h (") if cell contains text. Contains 87h if cell contains a number or formula
Contents	ASCII string with contents of cell.

If a cell contains a formula and its result then one packet contains the formula, and the cell is resent in the next packet but this time containing just the value.

For example, here are the packets sent to PLAN to export a simple sheet:

```
84 54 45 53 54                  Export TEST in DIF format.
02 04                           Columns A-B, Rows 1-4 used.
01 00 22 43 4F 4C 31            B1, contains "COL1"
00 01 22 52 4F 57 31            A2, contains "ROW1"
01 01 87 31                     B2, contains 1
00 02 22 52 4F 57 32            A4, contains "ROW2"
01 02 87 35                     B3, contains 5
01 03 87 3D 42 32 2B 42 33      B4, contains =B2+B3 (formula)
01 03 87 36                     B4, contains 6 (value)
FE                              End of sheet
```

Import is very similar. First the Psion sends CL/PLAN a packet of which the first byte is between 01 and 04 depending on the file type chosen, and the rest contains the file name. It just returns 00. Then the Psion sends a 00 and PLAN returns the first non-empty cell, in the same format as before. This is repeated until PLAN sends FE00, signifying the end of the sheet.

UTILITY SYSTEM SERVICES

The system services are explained in detail at the system services page.
They are included here only for completeness!

BUFFER HANDLING

UT$CPYB

Copies one buffer to another buffer. The two buffers can overlap in any way required.

UT$ICPB

Case independent buffer compare.

UT$CMPB (LZ only)

A case dependent buffer compare. Works in exactly the same way as UT$ICPB but is case dependent.

UT$FILL

Fills a buffer of the given length with the specified byte, filling two bytes at a time for efficiency.

T$ISBF

Finds the occurrence of the minor buffer within a major buffer.

UT$WILD (LZ only)

Works in the same way as UT$ISBF except that the wild characters '*' and '+' apply.

UT$SPLT

Finds the address and length of a field in a buffer where the fields are separated by the character specified.

UT$UTOB

Converts an unsigned binary number to ASCII decimal.

UT$XTOB

Converts an unsigned binary number to ASCII hexadecimal in the buffer at X.

DISPLAY HANDLING

These services provide complex display handling, displaying any required parameters according to the format control string parameter. The LCD display is not cleared and the text is displayed from the current cursor position and vertical scrolling will begin after the text fills the whole LCD display.

UT$DISP

Displays literal text and variables according to the format control string which is inserted in-line directly after the operating system call to UT$DISP.

UT$DDSP

Displays literal text and variables according to a given format control string.

This is identical to UT$DISP except that the format control string is not inserted in-line.

UT$CDSP (OS Version 2.5 and later only)

Like UT$DISP but screen is cleared first.

THE FORMAT CONTROL STRING

Format descriptors interspersed between literal text in the format control string, allow the display of variables formatted in the specified way.

Optionally, the programmer can justify the text within a field of chosen width, where the field is filled with a specified character.

All display is done through the service routine DP$EMIT, so that control characters (such as 16 for beep) can be used in the format control string.

PASSING THE PARAMETERS

Variable parameters must be pushed onto the machine stack in reverse order to that in which they will be displayed. For a buffer parameter, the length byte must be pushed before the address.

WARNINGS

The variable parameters must correspond in number and in type to the specification of the format control string.

EDITOR

There are two distinct editors in the operating system. The operating system itself uses the LANGUAGE EDITOR is for editing OPL programs, and the LINE EDITOR is used in all other editing situations; the editors can however each be used in a wide range of circumstances. Both can be called by the programmer, so a basic outline of some of the differences is given here. For a detailed description please refer to the system services page.

The line-editor is used for editing a maximum of 255 characters in the editor buffer, RTT_BF. Any text on the display prior to calling the editor will be treated as a prompt, with the text that is editable displayed directly after it. If required, a string can be set up for editing by copying it into the editor buffer before calling the editor.

The language editor can be used for editing any text files. The amount of text that can be edited is limited only by memory availability. Free RAM is allocated to the editor as required. An existing file can be loaded for editing in this memory. A file can be saved with a block file type between $81 and $8F inclusive, as specified by the programmer.

When exiting the editor, the user has the choice of

1. translating an OPL program,
2. saving the file,
3. quitting, in which case everything typed in the editing session is lost.

Since any type of text file can be edited (not only OPL procedures), the programmer can choose to prevent the option to translate the edited file.

LZ machines provide an extended language editor.

LINE-EDITOR SYSTEM SERVICES

ED$EPOS

This is the standard input line-editor.

A maximum characters may be specified. There are two modes of operation, multi-line and single-line editing. With multi-line editing the down arrow key starts a new line of text, while with single-line editing the up and down arrow keys send the cursor to the beginning and end of the editable text respectively. A new line is represented in the editor buffer by a TAB character (ASCII value 9).

The editor is exited when the EXE key is pressed, or when the ON/CLEAR key is pressed with an empty editor buffer. When the editor buffer is not empty, the ON/CLEAR key clears it and does not cause an exit. The programmer can also choose whether to allow exit from the editor when the user presses the MODE key.

ED$EDIT

Calls the line-editor service routine ED$EPOS with the cursor initially positioned on the first editable character.

TL$XXMD

Typically used to edit a filename, with the MODE key being used to select a PACK as in "FIND A:".

TL$XXMD clears the screen, prints a string followed by a space, a device letter 'A' to 'C' and a colon. The system service ED$EDIT is then called to edit a line of text, typically a file name. If the MODE key is pressed, the device letter displays the next available pack. TL$XXMD returns when either EXE or ON/CLEAR is pressed.

TL$ZZMD (LZ only)

Works like TL$XXMD except that the screen is not cleared.

LANGUAGE EDITOR

LG$RLED

Handles RUN, LIST, EDIT and DELETE commands in the top level PROG menu. LIST, EDIT and DELETE can operate on any text file, while RUN operates on OPL procedures only. The system variable TLB_MODE must be set by the programmer to control which of the top level commands is performed.

The standard input language editor can be used for editing any text files (with size limited only by memory availability). The file is loaded into RAM specially allocated to the language editor. An edited file can be saved with a block file type between $82 and $8F inclusive.

When exiting the editor, the user has the choice of

1. translating an OPL program,
2. saving the file,
3. quitting, in which case everything typed in the editing session is lost.

Since any type of text file can be edited (not only OPL files), the programmer can choose to prevent the option to translate the edited file. Attempting to translate a non-OPL file will simply produce an error message.

The language editor differs from the line-editor (ED$EPOS) in the use of cursor and control keys.

1. The ON/CLEAR key deletes only the current line (not the whole record).
2. The down arrow key moves the cursor one line down the file. (It does not create a new line).
3. The EXE key creates a new line. (It does not exit the editor).
4. A line can be split in two using the EXE key.

Note that on the LZ the PROG menu utilities may also be accessed by LG$ENTR.

LG$NEWP

Calls the language editor to create a new file.

This is identical to LG$RLED when it is operating in editing mode except that the file to be edited is created.

LINE VIEWER

ED$VIEW

Allows the user to view the string that is in the run-time buffer.

New lines are specified by TABs, and the up and down arrow keys control the line being scrolled. Two spaces are displayed between the end and beginning of any scrolling text. The left and right arrow keys are used to stop, start and change direction of scrolling. Any other key pressed ends the service.

MENUS

A menu consists of a list of item names (with leading byte-count), each followed by the address of an associated subroutine, or by 0. The maximum name length is 16. The items are displayed (and numbered) in the same order as in the menu-list. The first item is displayed at the top left-hand corner (HOME position) of the LCD display. The display is filled from left to right until an item is too long to fit, in which case a new line is started

On the LZ all menus (except 1 line menus) are aligned in 3 columns. However if there is any item greater than 6 characters long, 2 columns will be used. Aligning in columns uses more space in RTT_BF where the menu is constructed, so to maintain compatibility, if an aligned menu will not fit, the menu reverts back to a single spaced menu automatically. All menu items appear on the screen capitalized (but not in 2-line mode - for compatibility) but this can be overridden by writing 1 to MNB_CPTZ ($209C). Note that any spaces within menu items are converted to character 254 to save confusion.

USER MENU

MN$DISP

Displays a menu according to the specified menu-list, allowing the user to scroll through the menu using the arrow keys or by pressing the first letter of the item to be selected.

MN$DISP exits when the user either

1. presses any of the special keys specified, or
2. selects an item with a unique first letter by pressing its first letter.

MN$XDSP (LZ only)

Exactly the same as MN$DISP except that text already on the screen is treated as a title with the menu underneath (starting on the next clear line). If more the 3 lines of text are on the screen, the text will be truncated, and the last line used for the menu.

MN$1DSP (LZ only)

Exactly the same as MN$DISP except that the menu is displayed in one line only and scrolls horizontally. The remaining 3 lines of the screen are left intact. The menu is displayed in the line containing the cursor.

MN$TITL (LZ only)

Exactly the same as MN$DISP except that the top line is used to display an icon on the left and a running digital clock on the right just like the top-level menu.

UDG 0 is always displayed in the top left hand corner of the display.

TOP LEVEL MENU

The top level menu is held in RAM in the second allocator cell. Either OPL procedures or machine code programs may be called from the top-level menu.

On the LZ the top level menu has been extended to allow the names of notes and files to be inserted as well as the names of OPL procedures. This is done by using 0001 and 0002 as the execution addresses for Notes and Files respectively (0000 is still for OPL).

TL$ADDI

Inserts a given menu item into the top-level menu at given position.

TL$DELI

Deletes an item from the top level menu.

TL$RSTR (OS versions 2.7 and above only)

Restores several top-level functions to their initial state. This service first boots all devices, asks which language is to be set (on multi-lingual machines), initialises World and Notes (on LZ machines), and resets the top- level menu. The system time, the alarms, the diary and all files are left unaffected.

MATHS FUNCTIONS

INTEGER ARITHMETIC

UT$SDIV

2-byte by 2-byte signed integer division routine.

WARNING: Does not check for division by zero, which will cause an infinite loop.

UT$SMUL

2-byte by 2-byte signed integer multiply routine.

UT$UDIV

2-byte by 2-byte unsigned integer division routine.

WARNING: Does not check for division by zero, which will cause an infinite loop.

UT$UMUL

2-byte by 2-byte unsigned integer multiply routine.

FLOATING-POINT NUMBERS

These consist of a 12-digit precision BCD-packed mantissa with one byte for the sign and a one byte signed binary exponent. They are structured as follows: (low memory -> high memory)

mantissa_1	low-order byte
mantissa_2	.
mantissa_3	.
mantissa_4	.
mantissa_5	.
mantissa_6	high-order byte
exponent	1 byte
sign	1 byte

Each byte of the mantissa contains two decimal digits, the high-order digit in the top 4 bits and the low-order digit in the bottom 4 bits. The sign byte has the normal sign convention i.e. negative if bit 7 set, positive otherwise. Standard practice in the Organiser operating system is $80 for negative and zero for positive. An exponent of zero (with a non-zero mantissa) implies that the value of the number lies between 1 and 9.99999999999 inclusive.

The mathematical routines operate on two floating-point registers, which are identical in structure to the floating-point numbers described above except for the addition of a guard byte to the mantissa. These bytes are used for rounding purposes.

The registers are arranged in zero-page memory as follows:

Address	Variable name	Function
$C5	MTT_AMAN	accumulator: guard byte
$C6	-	start of accumulator mantissa
$CC	MTB_AEXP	accumulator exponent
$CD	MTB_ASGN	accumulator sign
$CE	MTT_OMAN	operand: guard byte
$CF	-	start of operand mantissa
$D5	MTB_OEXP	operand exponent
$D6	MTB_OSGN	operand sign

Thus the floating-point number -12345.0006789 can be declared as follows:

```
FP_CONST:
        .BYTE   $89,$67,$00,$50,$34,$12         ;mantissa
        .BYTE   4                               ;exponent
        .BYTE   $80                             ;negative sign
```

and copied into the accumulator by the following code:

```
        LDD     #MTT_AMAN+1
        LDX     #8
        STX     UTW_S0:
        LDX     #FP_CONST
        OS      UT$CPYB
```

FLOATING-POINT ARITHMETIC

The four binary operators add, subtract, multiply and divide operate directly on the floating-point registers in the order

accumulator (operator) operand = accumulator

e.g. to do the operation 6/3 = 2, 6 is placed in the accumulator, 3 in the operand and the result after calling mt_fdiv will be in the accumulator.

Any number used as an argument to these routines must be normalised, i.e. the most significant digit of the mantissa must be non-zero for all non-zero numbers. Any number having zero in the most significant byte is treated as zero.

For example declaring the number 0.00009999 as follows

```
        .BYTE   0,0,$99,$99,0,0         ;most significant digit is zero
        .BYTE   -1
        .BYTE   0
```

is incorrect. It should read:

```
        .BYTE   0,0,0,0,$99,$99         ;non-zero digits shifted to
                                        ;most significant position
        .BYTE   -5                      ;and exponent adjusted accordingly
        .BYTE   0
```

These routines also require the exponent to be in the range -99 to 99. The result is always returned as a normalised floating-point number in the accumulator, the operand remains unchanged, and both guard digits are cleared to zero. No validation is made of the contents of the registers on input, and the arithmetic routines will attempt to operate on invalid floating-point numbers with unpredictable results.

A description of the system services to handle floating-point arithmetic follows.

MT$FADD

Does a floating-point add on the accumulator and operand registers.

MT$FSUB

Subtracts the floating-point operand from the accumulator.

MT$FNGT

Toggles the sign of the register pointed to by X.

MT$FMUL

Performs a floating-point multiply on the BCD registers.

MT$FDIV

Performs a floating-point divide on the BCD registers.

CONVERTING NUMERIC TEXT TO FLOATING POINT

MT$BTOF

Converts a numeric ASCII string to a floating-point number. The string may be terminated by any character which is not numeric (and not by ".", "E" or "e" unless these have previously occurred in the input string).

An Error is returned if:

- The exponent is greater than 99 or less than -99 (irrespective of the value of the mantissa)
- The input string contains more than 12 significant digits (not including leading or trailing zeros)
- The total of significant digits + trailing zeros exceeds 44
- There are no valid numeric digits in the mantissa (e.g. .E5)
- There are no valid numeric digits in the exponent (e.g. 6E)

CONVERTING FLOATING-POINT TO TEXT STRING

NT$FBDC

Converts floating-point accumulator to numeric ASCII text in decimal format. This is the format nnnn.nnn, i.e. a decimal number with no exponent. If the number of significant digits after the decimal point is greater than the decimal places specified, the number is rounded, otherwise it is zero-filled on the right to make up the necessary decimal places.

MT$FBEX

Converts floating-point accumulator to numeric ASCII text in exponential (scientific) format. This is in the form n.nnnnnE+nn, a decimal mantissa with one digit before the decimal point followed by the exponent. The exponent is always expressed as a + or - sign followed by 2 numeric digits.

MT$FBIN

Converts floating-point accumulator to numeric ASCII text in integer format. If the value in the floating-point accumulator is not an integer, then the string output is the accumulator value rounded to the nearest integer.

MT$FBGN

Converts floating-point accumulator to numeric ASCII text in general format, i.e. in whichever of the above three formats is the most suitable for the given number and field width.

The output string will be in integer format if the number is a floating-point integer, otherwise decimal format. If the field width is not sufficient for the above, the number is output in scientific format rounded to as many decimal places as can be accommodated. A field width of 7 is sufficient to convert any floating-point number without returning an error, though severe truncation of the mantissa may result.

THE SCIENTIFIC FUNCTIONS

With a few exceptions, the system services for these functions require a single floating-point number on the run-time stack (described in the chapter on the language pointers) as input. On exit this number is replaced by the floating-point result, with the stack pointer unchanged. The result is also left in the floating-point accumulator.

FN$ATAN

Returns the arctangent in radians of a floating-point number.

FN$COS

Returns the cosine of a floating-point number (assumed to be in radians).

FN$EXP

Returns the result of the arithmetic constant e (2.71828...) to the power of a floating-point number.

FN$LN

Returns the logarithm to the base e of a floating-point number.

FN$LOG

Returns the logarithm to the base 10 of a floating-point number.

FN$POWR

Returns the result of X to the power of Y where X and Y are both floating-point numbers.

FN$RND

Pushes a random floating-point number onto the run-time stack. The number will always lie between 0 and 0.999999999999 inclusive.

The seed for the random number generator is held in the 7 bytes at FNT_SEED. If these bytes are set to a particular value then the sequence of random numbers following this will always be the same.

FN$SIN

Returns the sine of a floating-point number (assumed to be radians).

FN$SQRT

Returns the square root of a floating-point number.

FN$TAN

Returns the tangent of a floating-point number. The argument is assumed to be in radians.

FN$ASIN (LZ only)

Returns the arcsine in radians of a floating-point number.

FN$ACOS (LZ only)

Returns the arccosine in radians of the floating-point number.

SORTING

XF$SORT (LZ only)

Sorts a file. An optional subroutine may be specified to be called during the sort (usually used to print to the screen). The method used is a Quick Sort algorithm with passes as follows:

1. Generate the tag list
2. Sort the tag list
3. Reconstruct the file from the sorted tag list

The total number of bytes in RAM required to sort a file of size S bytes and containing N records is approximately given by the following formula:

Space required = 2 * (S + N)

If there is insufficient memory for the sort, the file will left alone and an error will be returned.

UT$SORT (LZ only)

Provides the core utility for a Quick Sort. The routine assigns space for a Tag for each of the items to be sorted. Then a caller supplied routine is called in three different ways:

1. to obtain a tag for each item
2. to compare any two items
3. to receive the ordered tags

LIST FUNCTIONS

FN$SUM (LZ only)

Returns the sum of a list floating-point numbers.

FN$MEAN (LZ only)

Returns the arithmetic mean of a list of floating-point.

FN$VAR (LZ only)

Returns the sample variance of a list of floating-point.

The sample variance differs from the population variance in assuming that the data set provided constitutes a sample of the data rather than the complete population.

If the complete data set IS provided, the population variance canbe calculated by multiplying the sample variance by (N-1)/N.

FN$STD (LZ only)

Returns the sample standard deviation of a list of floating-point numbers.

The sample standard deviation differs from the population standard deviation in assuming that the data set provided constitutes a sample of the data rather than the complete population.

If the complete data set IS provided, the population standard deviation can be calculated by multiplying the sample standard deviation by sqrt((N-1)/N).

FN$MIN (LZ only)

Returns the minimum of a list of floating-point numbers.

FN$MAX (LZ only)

Returns the maximum of a list of floating-point numbers.

BUZZER

This chapter describes the system services which drive the buzzer. The buzzer is driven by software, so programs calling these routines are paused for the duration of the sound. The buzzer volume is not controllable via the operating system services. Note that all of the following services run with interrupts DISABLED in order to prevent the keyboard scanning interrupts from introducing gaps into the middle of the note.

BZ$ALRM

Gives the noise used by ALARM.

BZ$BELL

Makes a standard beep, of fixed frequency and duration. This is the sound produced by PRINT CHR$(16) in OPL, or by a call to DP$EMIT with A=16.

BZ$TONE

Makes a beep specified of duration and frequency.

ERROR HANDLING

ER$LKUP

Given an error code, returns the address of the appropriate error message. If there is no error message for this error code, the default message "*** ERROR ***" is returned.

ER$MESS

Displays the error message associated with the given error code and waits for the user to press SPACE or ON/CLEAR. If there is no error message for this error code, the default message "*** ERROR ***" is displayed.

NOTE: In the case of error code 194, the machine displays the message "BATTERY TOO LOW" for 4 seconds, then switches off.

UT$ENTR

Allows a routine to be called in such a way that it may be exited from any of the routines nested within it.

The most important use for UT$ENTR (together with UT$LEAV) is for error handling: they can mimic the ON ERROR and RAISE facilities in OPL.

UT$LEAV

Causes an exit from the control region established by the most recent UT$ENTR call.

ER$PRNT (LZ only)

Prints a given string in the same format as an error message.

MISCELLANEOUS SYSTEM SERVICES

UT$XCAT

Displays all filenames of the given record type on the current device like DIR in PROG.

XT$DIRM (LZ only)

Produces a directory of files with a prompt like EDIT in PROG.

XT$BAR (LZ only)

Sets up UDGs and draws a bar graph as in the INFO application.

UT$YSNO

Waits for 'Y', 'N' or ON/CLEAR to be pressed, returning the key pressed.

BUILT-IN APPLICATIONS

All Organisers:

ALARMS

- ALARM CHECKING INTERRUPTS
- WAKING UP FOR AN ALARM
- SYSTEM VARIABLES

DIARY

Model LZ only:

NOTEPAD

CALLING THE BUILT-IN APPLICATIONS

- ALARM
- CALC
- NOTES
- PROG
- TIME
- UTILS
- WORLD
- XFILES

ALARMS

The eight alarms are stored in a fixed length 48 byte area AMT_TAB ($22F9).

Each entry contains a date-time in the usual format, with a flag indicating the type of alarm.

BYTE	MODELS CM & XP (LA)		MODEL LZ		
	RANGE		RANGE		
0	0 - 99	year (1900 - **1999**)	0 - 255	year (1900 - 2155)	
1	0 - 11	month	0 - 11	month	
2	0 - 30	day	0 - 30	day	
3	0 - 23	hour	0 - 23	hour	
4	0 - 59	minutes	0 - 59	minutes	
5	0 - 4	0 Alarm not Set 1 Once 2 Weekly 3 Daily 4 Hourly	0 - 133	0 Alarm not Set 1 Hourly 2 Daily 3 Workdays 4 Weekly 5 Once	**plus** sound flag 64 Siren 128 Chimes (optional)

A system variable AMB_WRKD is provided on the LZ machines to indicated which days are workdays.

An alarm entry is canceled by setting byte 5 to zero. Before setting or modifying any alarms, byte 5 should be cleared and then set last of all. This is to prevent interrupts from checking that entry.

Note that although there is no way of manually setting an alarm outside the current week, this limitation need not apply to user programs which manipulate AMT_TAB directly. You can set an alarm to ring at any time between 1900 and end of 1999 (LZ: 2155).

The date-time of a repeating alarm is updated each time it rings; an alarm entry does not contain the original date-time.

ALARM CHECKING INTERRUPTS

Both the diary and the alarms are scanned approximately every minute by the 50ms maskable interrupts which scan the keyboard. Users wishing to alter the alarm or diary alarm action, see vector BTA_OCI, or the BZ$ALRM service.

Every minute an NMI makes a request for alarm checking by setting the flag AMB_DOIT provided the following conditions are met:

- BTB_IGNM <> 0 (else NMI does nothing)
- AMB_EI <> 0
- TMB_SECS = 0 (we are on a minute boundary)

The flag AMB_EI is provided specifically so that user programs can disable alarm checking.

If all these conditions are met, the alarm is not actually checked immediately: this is left to the next maskable interrupt which rings any pending alarms whenever AMB_DOIT is set. This means that alarms are checked as soon as possible after each minute boundary, but any time-critical activities such as writing to datapacks and other operations can delay alarms by using the SEI instruction.

Alarms will never occur while the interrupt mask is set. Also certain activities such as device booting (DV$ calls), storage management (AL$ calls), or modification of the diary or alarms can cause an ALARM to ring late. If interrupts are not required, then an SEI instruction is all that is required to disable alarm checking. If interrupts are required the following code must be used to maintain compatibility with all OS versions:

```
        LDA     A,AMB_EI
        PSH     A
        TPA                     ; preserve interrupt mask
        SEI
        CLR     AMB_EI          ; prevent NMI setting AMB_DOIT
        CLR     AMB_DOIT        ; in case AMB_DOIT already set
        TAP                     ; restore interrupt mask
        ... ; user program alarm checking now off
;       ...
; the next two lines are optional
        INC     AMB_EI          ; set AMB_DOIT if check required **
        INC     AMB_DOIT        ; on next interrupt             **
        PUL     A
        STA     A,AMB_EI        ; restart normal checking
```

The two lines ** will cause the next 50 ms interrupt to perform a check without waiting for the next minute boundary. This will minimize any late running of the alarms. The AMB_DOIT flag can also be set to request more frequent checking than once each minute. However, AMB_DOIT must at no time be set non-zero when AMB_EI is zero as this may cause problems with some early OS versions. Note also that AMB_EI should not be set to $FF.

The 50 ms interrupt first checks the DIARY then the eight alarms. All alarms are sounded even if they are overdue. The earliest DIARY alarm will sound, then the lowest numbered ALARM alarm. If more than one DIARY or ALARM alarm are due, they will ring in pairs (DIARY,ALARM) each minute.

Before a DIARY alarm sounds, the alarm flag (byte 6) in that entry is cleared. Before an ALARM alarm sounds, the repeat time is added on for repeating alarms, and byte 5 is cleared for non-repeating alarms.

The system service DP$SAVE is called to save the screen, and the time and diary text or "ALARM" is displayed for one minute or until ON/CLEAR is pressed. The screen is then restored with a call to DP$REST. The ON/CLEAR key is polled directly, so the keyboard buffer is not affected.

On CM/XPs, BZ$ALRM makes the ALARM sound, while the DIARY beep is a call to BZ$TONE with D = 200 (proportional to 1/frequency), X=50 (note length).

WAKING UP FOR AN ALARM

The Organiser II maintains the system time with a 12 bit external counter while switched off. The machine switches on when the counter overflows every 2048 seconds (34 minutes 8 seconds), updates the system time, and switches off again.

The system service BT$SWOF rings any alarms pending, then checks if an alarm is due in the next 34 mins 8 secs. If necessary, BT$SWOF sets the counter to a value greater than zero to switch the machine on early.

When the Organiser II switches on, it rings the alarm then remains on until normal switch-off. Users wishing to alter this behavior see vector BTA_SOF and for vector BTA_WRM.

SYSTEM VARIABLES

AMB_WRKD ($20A7 - LZ only)

AMB_WRKD is used to determine which days of the week are workdays. With bit 0 representing Monday, bit 1 Tuesday etc. bits are set for workdays and cleared otherwise. Thus a value of $4f in AMB_WRKD moves the weekend to Friday and Saturday. Setting AMB_WRKD to 0 prevents checking for Workday alarms and removes the option from alarm setting.

Note: AMB_WRKD should not be set to $80. Trying to set a workday alarm, or having an existing one go off will cause the machine to lock-up.

DIARY

The information in the diary is stored in RAM in an allocated CELL, separate from the RAM device A:, so the diary can not be accessed as a file from OPL or by the top level functions FIND and SAVE.

Entries are kept sorted by date and time in order to allow alarm checking interrupts to scan the diary efficiently. This also enables the SAVE and RESTORE functions to save the whole diary as a block easily.

The diary is held in the third cell, base pointer $2004. Below the diary are the DEVICE and MENU cells, so any operation which grows or shrinks these cells will cause the diary to move. This includes calls to DV$ and TL$ services, device booting at the top level and any changes in the top level menu.

The diary is terminated by a zero byte, not by the end of the cell and so at initialisation time the diary cell just contains this zero byte. It is illegal to shrink the cell to nothing using AL$ZERO or AL$SHNK.

Within the diary cell the entries are stored as follows:

BYTE	MODELS CM & XP (LA)		MODEL LZ	
	RANGE		RANGE	
0	1 - 64	length of text	2 - 65	length of text + 1
1	0 - 99	year (1900 - **1999**)	0 - 255	year (1900 - 2155)
2	0 - 11	month	0 - 11	month
3	0 - 30	day	0 - 30	day
4	0 - 23	hour	0 - 23	hour
5	0 or 30	minutes	0,15,30,45	minutes
6	0 - 60	if 0 no alarm set else (number of minutes early+1)	0 - 60	if 0 no alarm set else (number of minutes early+1)
7...	string	text of diary entry (1-64 chars)	string	text of diary entry (1-64 chars)
			1-96	duration (in quarter hours)

Example (CM/XP):

```
JAN:12:SUN:17:00
ABCDEF
```

(in 1986)

(alarm set 15 minutes early)

- $06 (length of "ABCDEF")
- $56 (=dec 86, year 1986)
- $00 (month JAN)
- $0B (day 12)
- $11 (hour 17)
- $00 (minutes 0)
- $10 (alarm 15 minutes early)
- 6 bytes "ABCDEF"

and then the next entry or $00 as terminator.

Example (LZ):

```
Mon 8 May 1989 Wk19
6:00a
5:15p ABCDEF
5:45p
```

(alarm set for 15 minutes early)

- $07 (length of "ABCDEF" plus one)
- $59 (= DEC 89, year 1989)
- $04 (month MAY)
- $07 (day 8)
- $11 (hour 17)
- $0F (minutes 15)
- $10 (alarm 15 minutes early)
- 6 bytes "ABCDEF"
- $02 (duration 30 minutes)

and then the next entry or $00 as terminator.

Bear in mind the following restrictions when manipulating the diary:

1. Use SEI to prevent alarm checking interrupts while the diary is being modified or during loading from a datapack. Also see the section on interrupts.
2. All entries must be inserted in chronological order.
3. No two entries must have the same time.
4. Entries may not overlap because of their durations. They may, however, meet exactly. No entry must pass midnight owing to its duration.
5. Text must be less than 65 chars.
6. The allocator system services must be used to allocate space in the diary.

The following examples scan through each diary entry:

```
        LDX     $2004   ; X points to 1st byte
        BRA     2$
1$:     ADD     B,#7    ; skip length byte date alarm flag
        ABX             ; and skip over text
2$:     LDA     B,0,X   ; get length byte
        BNE     1$      ; until 0 terminator found
```

```
 ADDR% = PEEKW($2004)
 LEN%  = PEEKB(ADDR%)
 WHILE LEN% <> 0
   ADDR% = ADDR% + 7 + LEN%
   LEN%  = PEEKB(ADDR%)
 ENDWH
```


SAVING THE DIARY

- CM/XP machines save the diary as a block file of type $82. The file structure is described in the Files chapter. As these files are not human readable, the example OPL program may be used to write a formatted listing of a CM/XP diary to either a file or the commslink.
- On the LZ the diary is no longer saved as an ordinary data file. Each entry is saved as a separate record - the previous example would be:
- ```
  1989050817150216
  ABCDEF
  ```

  The first field specifies the date of the entry (1989, 05, 08, for 8th May 1989), the time (17 15, for 5:15pm), the duration (02) and the alarm byte (16, for 15 minutes before - again, 00 would mean "no alarm"). The second field is the text of the entry. The order of things in the first field, and the use of zeroes where necessary to keep things the same length (e.g. "02" for a duration of 2) make the file easy to sort in OPL, or otherwise. Because a saved record can be edited or even created by a user, the year, month and day sections start from 1, not 0 - e.g., the 1st of the month is saved as "01", to the file, not "00" as in the diarycell. Note, though, that the "Restore" option in the diary will restore entries in any order from a file. The "Xrestore" option can be used to load a CM/XP diary from a pack, in which case each entry is given a duration of 2 (=30 minutes).

NOTEPAD (LZ only)

The Notepad uses the same editor as is used to edit OPL programs. This editor can be invoked directly by calling the system service LG$EDIT (see chapter Editor).

Notepad Structure

The Notepad text is stored in allocator cell 16 (defined as NOTECELL) in exactly the same format as OPL source procedures. On cold boot, the notepad cell is initialized to contain the name "Notepad:" and system variables (see below) are initialized.

The cell is structured as follows:

- a leading word specifying the length of the rest of the cell
- 1-8 byte ASCII name of notepad
- a colon to terminate the name
- an optional binary 0 byte to terminate the name
- ASCII lines of notepad text, each line terminated by binary 0 byte

Each line has a maximum of 255 bytes including the 0 (these bytes are encrypted when not in the editor if a password has been set).

Note that although the notepad name and colon are not editable, if the optional 0 is not included then the first text line including the name has a maximum of 255 bytes. The bytes after the colon can be edited. A line consisting only of the terminating 0 represents a blank line. The notepad can have any number of lines up to the capacity of memory.

If the cell does not have this structure, the editor may display the "PACK READ" error as if a corrupted notepad has been loaded from a datapack or may produce unexpected results.

The notepad is saved onto a pack as a block file of type 87 (defined as BNOTTYP), and has the same structure as a saved OPL procedure with object but instead of the Object Code (OCODE) bytes, notepad header information is stored. Following the header, the file has the same structure as the notepad cell as described above. The OCODE cell is in fact used to hold the header information prior to saving to pack and is zeroed after saving.

The header structure is:

- 1 word length of Header information
- 1 byte flags for notepad
  - bit7 - true if numbered
  - bit3 - true if on/clear exits editor
  - bit2 - true if prompt to be capitalized
  - bit1 - true if no title required
  - bit0 - true if changed (used in editor)
- 1 byte password flag
  - 0 for no password set
  - 9 for password set
- if password flag is 9: 9 bytes password code (see Passwords)

Note that if this structure is generated by the programmer, the password flag must be set to 0 to signify no password or the program will try to decrypt the file when it is loaded.
  ```

System Variables

Four system variables that are permanently maintained for the current notepad can be referenced from outside the notepad editor.

NTB_FLGS ($7FEA - LZ only)

This byte contains flags for the editor. The following bits are currently used:

7	true if numbered	may be poked
3	true if on/clear exits editor	do not poke
2	true if prompt to be capitalized	may be poked
1	true if no title required	may be poked
0	true if changed (used in editor)	may be poked

The default value for a new notepad is $08 (bit3 set).

Note that if bit3 is cleared then the notepad cannot be exited. This bit is 0 for editing OPL procedures where the menu includes the item EXIT.

NTB_PSSW ($7FEB - LZ only)

This byte is 0 if the current notepad in memory has no password or any non-zero value if a password exists. It must not be poked or the editor may attempt to decrypt unecrypted data or not to decrypt encrypted data etc. with unknown results.

NTW_CLIN ($7FFD - LZ only)

This word contains the current line number in the current notepad. Line 0 is the line containing the notepad name. This defines which line the cursor is on.

This variable may be poked to change the line number but should not be given a value greater than the last line number in the notepad. The maximum line number is 1 less to the number of zero line-delimiters in an unecrypted notepad.

NTB_CPOS ($7FFF - LZ only)

This byte contains the cursor position in the current line in the current notepad.

This variable may be poked to change the cursor position but should not be given a value greater than the length of the current line.

CALLING THE BUILT-IN APPLICATIONS (LZ only)

On the LZ entry points to the top-level applications have been included as system services.

They are named as XX$ENTR where XX is the name of the application (e.g. DI$ENTR provides access to the diary functions). Most of these system services take a "function number" to identify which function of the application to run.

Note: function numbers 0, 1 and 2 are standard across the interfaces to the built-in applications ("initialise", "normal entry" and "search"). Values above 2 are application-dependent.

Please note: Using the following calls on a 2-line machine will cause a crash!

The only way to access the built-in applications on CM/XPs is to look up the entry points in the menucell.

Alarm: AM$ENTR

Provides an entry point into the ALARM routines.

The available functions are:

- Function 0 - Initialise. This clears all alarms and enables alarm checking by setting AMB_EI.
- Function 1 - Calls the ALARM application as from the top level.
- Function 2 - Checks for Diary and ordinary alarms due now and in the next 34mins and 8 secs. If an alarm is due it will go off, if one is due in the next 2048 seconds, the number of seconds before the next alarm is returned.
- Function 3 - Tests for Diary and ordinary alarm due now. If one is due it will go off.
- Function 4 - Checks for unacknowledged diary alarms. This does the 'Review missed alarms' screen on switch-on. It checks for any DIARY alarms that have gone off but were not acknowledged by pressing ON/CLEAR. If there are any it displays a screen showing how many were missed and will review them.
- Function 5 - Turns alarms off, preserving the settings.
- Function 6 - Restores alarms (opposite to function 5).

Calculator: CA$ENTR

Provides an entry point to the calculator exactly like selecting **CALC** in the top-level menu.

Do not call from OPL, since OPL is not re-entrant unless all its variables are preserved.

Prog: LG$ENTR

Provides an entry point to the PROG application in the top-level menu.

- Function 1 - Call the PROG application as from the top-level menu.
- Function 2 - Search block body files on packs for a given string.

Notepad: NT$ENTR

Provides an entrypoint to the NOTEPAD application.

- Function 0 - Initialise notepad. This is called on cold booting the LZ.
- Function 1 - Call the notepad as from the top-level menu option "Notes".
- Function 2 - Search first the current notepad and then all saved notepads for a given string. Note that password protected notepads are not searched.
- Function 3 - Edit a named notepad as from top-level menu. If the name is not the same as that in the notepad cell, the file is searched for on the packs.
- Function 4 - Checks whether a notepad file is password protected.

Time: TI$ENTR

Provides an entry point to the TIME application.

There are two functions available:

- Function 0 - Initialise. Actually does nothing!
- Function 1 - Call the TIME application as from the top-level menu.

Utils: XT$ENTR

Provides an entry point to the UTILS application in the top-level menu.

- Function 0 - Initialises system password (as no password).
- Function 1 - Call the UTILS application as from the top-level menu.
- Function 2 - Call the SEARCH option in Utils.
- Function 3 - Call the INFO option in Utils.
- Function 4 - Call the PASSW option in Utils.
- Function 5 - Call the LANG option in Utils.
- Function 6 - Check the system password if it is set on (called on warm start).
- Function 7 - Actually reset the machine (called from RESET in Utils).
- Function 8 - Actually format a rampack (called from FORMAT in Utils).
- Function 9 - Call the DIR option in Utils.
- Function 10 - Call the COPY option in Utils.
- Function 11 - Call the DELETE option in Utils.

World: WL$ENTR

The world application gives the current time (ignoring day light time saving) and dialing code from the base city for 400 cities and 150 countries.

- Function 0 - Initialises the world application. It sets the base to be London, Paris or Bonn depending on the language byte. The initial city is always set to New York Manhattan.
- Function 1 - Runs the world application exactly as from top level menu.
- Function 2 - Returns the base city and country names.

xFiles: XF$ENTR

Provides an entry point to the XFILES application in the top-level menu.

- Function 0 - Initialises the current XFILES file to be MAIN.
- Function 1 - Enter the XFILES application as if from the top-level menu.
- Function 2 - Search files on all devices for a given string.

- Function 3 - Find a file as from top-level menu. The file is searched for on all packs in the order A;, B:, C: and if found made the current file in XFILES.
- Function 4 - Runs the top-level FIND function
- Function 5 - Runs the top-level SAVE function

LZ PASSWORDS

GENERAL

The LZ has a built in security system. It allows you to set a general password to lock up the organiser completely, and also to set a password on a notepad file which is then encrypted to keep out prying eyes.

Passwords are not stored directly in memory, otherwise a simple peek would allow someone immediate access to them. After a password has been typed in, it is encoded in a rather complicated way to give a 9 byte string, the passcode. This encoding is a little complicated to describe (see Key Generation), but not very difficult to do. It is however impossible to undo directly, so the originally typed password can't be recovered from the passcode.

Furthermore, the resulting passcode probably is unique to each password. This means that passwords can be checked by comparing their codes only.

SYSTEM PASSWORD

When a system password is given, it is encoded and the resulting 9 byte passcode is stored at $7FD7-$7FDF. It is difficult to recover the original password from these bytes directly. The only way to do this is to try all possible passwords until you find one that has the right code.

In practice however, it is either impossible or not necessary to do such a search for a system password. On a Psion with a system password set and active it is impossible to read the encoded password. The only way to deactivate an active system password if it is unknown is to remove the battery which of course destroys all data on A:. Note also that since external devices are not loaded during a warm boot it is not possible to intercept the password routine that way.

On the other hand, on a Psion with a password set but not active it is not necessary to do any calculating at all as it is easy to erase the password completely.

The byte at $7FD6 contains the system password state. It can contain the following values:

- 0 - There is no system password (this is the default).
- 1 - There is a system password but it is not active.
- 3 - There is a system password and it is active.

Note: Actually bit 0 is set when a system password has been given, bit 1 is set when it is active.

Poking 0 to $7FD6 will erase the system password. This allows you to set a new system password without knowing the old one. Note that there is no need to clear the passcode at $7FD7-7FDF as it is simply ignored until it is overwritten at such time when a new password is typed in.

NOTEPAD PASSWORDS

When a notepad is given a password, the password is encoded in exactly the same way as the system password. The 9 byte passcode is then also stored in the notepad file (see chapter Notepad). Thus when it is accessed using the notepad editor, it will be password verified before viewing/editing is allowed.

To further secure the notepad text from peeking, it is also encrypted. This encryption is done in a very simple way using an 8 byte key. The same algorithm that produces the passcode from the password is used (with slight modifications) to generate the 8 byte key.

Recovering the notepad text without the password seems to be very difficult. It is impossible to go directly from the passcode to the key without finding the password first. Trying to find out the password from the passcode, and then using that to get the key and decode the text is possible but involves quite an exhaustive search.

However the method used to encrypt the text is so simple that it is possible to decode it without knowing the password or key at all, provided the notepad contains at least about 20 characters of text. No further information is given on that subject, just be warned!

NOTEPAD ENCRYPTION

There now follows a description of the encryption method. Suppose we have the following notepad file:

```
Testing:
This is a
test.
```

and the password ABCDEFGH.

```
Txt: Testing:#  T   h   i   s   _   i   s   _   a  #   t   e   s   t   .  #
Asc:          84  104 105 115 32  105 115 32  97  0  116 101 115 116 46  0
```

Note that the end of a line is indicated by a 0 byte (denoted here by a #). The title of the notepad 'Testing:' will not be encrypted, and if like here there is a zero byte immediately following it, that byte is not encrypted either.

The encryption key is calculated from the password (see Key Generation), and turns out to be 82,47,5,181,247,92,30,165. On of the simplest types of encryption with a key is adding the key letter by letter (or rather number by number) to the text. This is the so-called Vigin$re cypher. The notepads are encrypted in this way, but to further disguise the encryption another sequence is added. The sequence is generated by consecutive multiples of 163, i.e.. 163, 70 (=163+163-256), 233, 140 (233+163-256), etc. This sequence is added to the key before being added to the plain text.

This is of course best illustrated with our example. For the first eight characters we have:

```
Text:          T   h   i   s   _   i   s   _
Ascii:        84 104 105 115  32 105 115  32

OldKey:       82  47   5 181 247  92  30 165
Seq:         163  70 233 140  47 210 117  24
NewKey:      245 117 238  65  38  46 147 189

Asc + NewKey: 73 221  87 180  70 151   6 221
```

Note that if a number exceeds 255, 256 is subtracted to keep it within range of a single byte.

The key that was used here is now used to calculate the key to use on the next 8 letters:

```
Txt:          a   #   t   e   s   t   .   #
Asc:         97   0 116 101 115 116  46   0

OldKey:     245 117 238  65  38  46 147 189
Seq:        187  94   1 164  71 234 141  48
Newkey:     176 211 239 229 109  24  32 237

Total:       17 211  99  74 224 140  78 237
```

The complete encryption is now:

```
T   h   i   s   _   i   s   _   a   #   t   e   s   t   .   #
73 221  87 180  70 151   6 221  17 211  99  74 224 140  78 237
```

KEY GENERATION

Here is a detailed description of the algorithm used to transform the password into the 9 byte code or 8 byte key.

1. Capitalize the password. If its length is less than 8 characters, then repeat it as often as necessary to get an 8 character string. For example, 'abc' becomes 'ABCABCAB'. Let p0 to p7 denote the ASCII characters of this string. For the final few steps let l be the length of the password and let t=(p0+p1+p2+p3+p4+p5+p6+p7) MOD 8. (Note that MOD 8 means the remainder on division by 8, so it is the same as taking the logical AND with 7.)
2. Compute g0 to g3 using the following formulae:

```
g0= (p0+3D)*(p7+25)*CB and ignore the lowest byte
g1= (p1  "   p6  "  "   "    "     "     "     "
g2= (p2  "   p5  "  "   "    "     "     "     "
g3= (p3  "   p4  "  "   "    "     "     "     "
```

   The numbers are obviously given in hexadecimal notation. Note that the multiplications result in 3-byte integer values, and only the 2 high bytes are taken to give the 2-byte unsigned integers g0..g3.

6. Compute f0, f1 using the following formulae:

```
f0= (g1+03)*(g2+0B)*A1 and ignore the lowest byte
f1= (g0+03)*(g3+0B)*AD  "    "     "     "     "
```

   The multiplications result in 5-byte integer values, and only the 4 high bytes are taken to give the 4-byte long integers f0,f1. (Note that the addition operations will never cause a carry, so the factors really are two-byte integers.)

8. Swap the two high bytes with the two low bytes of f0. Similarly f1.
9. Compute e0, e1 using the following formulae:

```
e0=f0*CB and ignore the lowest byte
e1=f1*CB  "    "     "     "     "
```

   Let d0 to d3 denote the 4 bytes of e0, from low to high. Similarly d4 to d7 denote the bytes of e1.

11. Compute the first 8 bytes of the by the following formulae:

```
c0=p0+d7
c1=p1+d6
.. .. ..
```

15. c6=p6+d1
c7=p7+d0

Of course these are restricted to byte values, so ignore any carry that might result.

16. Compute the final code byte by the formula
c8=d(t)+l

where t and l are remembered from step 1.

Now c0 to c8 is the 9 byte passcode.

To compute the 8 byte notepad encryption key, nearly the same algorithm is used. The only difference is that the eight bytes d0 to d7 are used, so steps 6 and 7 are omitted.

Also, during the calculation different numbers are used:

- Step 2: 25, 3D, C5 instead of 3D, 25, CB
- Step 3: 7, 3, A9, 97 instead of 3, B, A1, AD
- Step 5: C5 instead of CB

Example

Password is 'abcdefgh'. All numbers below are in hex.

```
1.  p0..p7=  41,42,43,44,45,46,47,48
    l = 8
    t = (224 mod 8) = 4

2.  g0= (41+3D)*(48+25)*CB = 2A8A[A2]
    g1= ... = 2A7C
    g2= ... = 2A6C
    g3= ... = 2A5B

      Similarly, for an encryption key we get:
      g0= (41+25)*(48+3D)*C5 = 28C7[76]
      g1= ... = 28DE
      g2= ... = 28F4
      g3= ... = 2908

3.  f0= (2A7C+3)*(2A6C+B)*A1 =046EEAFC[A9]
    f1= (2A8A+3)*(2A5B+B)*AD =04C32AFD[16]

      And for encryption key:
      f0= (28DE+7)*(28F4+3)*A9 = 0451EB3C[6B]
      f1= (28C7+7)*(2908+3)*97 = 03DBD692[96]

4.  f0= EAFC046E
    f1= 2AFD04C3

      And for encryption key:
      f0= EB3C0451
      f1= D69203DB

5.  e0= EAFC046E*CB = BA55D783[3A]
    e1= 2AFD04C3*CB = 2216A2C6[A1]

      And for encryption key:
      e0 = EB3C0451*C5 = B5052F52[55]
      e1 = D69203DB*C5 = A51E5CF7[87]

    d0..d7=  83,D7,55,BA,C6,A2,16,22

       And for encryption key:
       d0..d7=  52,2F,05,B5,F7,5C,1E,A5
            or 82,47,5,181,247,92,30,165 in decimal
            The encryption key is now done.

6.  p0..p7=  41, 42, 43, 44, 45, 46, 47, 48
    d7..d0=  22, 16, A2, C6, BA, 55, D7, 83
    c0..c7=  63, 58, E5, 0A, FF, 9B, 1E, CB
7.  c8= d(t)+l = d4+8 = C6+8 =CE
    so the nine byte passcode is
       63, 58,  E5, 0A,  FF,  9B, 1E,  CB,  CE
    or 99, 88, 229, 10, 255, 155, 30, 203, 206 in decimal.
```


PROGRAMMING LANGUAGE

INTRODUCTION

The Organiser Programming Language (OPL) is a high level language which has developed from a number of other languages:

- C ARCHIVE (the database module in Xchange)
- BASIC
- FORTH
- PL1

The language is designed to be:

- Fast
- Compact
- Flexible
- Accurate
- Extensible
- Simply overlayed

The language is stack based; all code is held on the stack as are all intermediate results. To achieve speed the source code is translated into an intermediate code (Q code) before it is run.

This chapter discusses the concepts of OPL, details and examples of Q code follow in the next chapter Q-Code.

DEFINITIONS

VARIABLES

All variables in OPL are held in one of three forms:

- Integer
- Floating pointing
- String

All this can either be simple variables or field variables.

All variables are zeroed when declared by a LOCAL, GLOBAL, OPEN or CREATE statements.

PROCEDURES

OPL is a procedure base language, a number of procedures normally go to make up a program. Up to 16 parameters can be passed to a procedure which always returns a variable.

When a procedure is called a header is placed on the stack, followed by space for variables declared and the Q code itself. When a procedure returns all the stack is freed for use by other procedures. This allows overlaying of code so that programs can run which are substantially bigger than the available memory on the machine.

PARAMETERS

Parameters passed to a procedure may be integer, floating point or string. They are passed by value. On the stack they are in reverse order to the order they are input.

For example the statement "PROC:(12,17.5,"ABC")" will generate the following stack entry before the procedure PROC is called:

```
high memory     00 12
                00                      ; Integer type
                00 00 00 00 50 17 01 00
                01                      ; Floating point type
                03 41 42 43             ; "ABC"
                02                      ; String type
low memory      03                      ; Parameter count
```

ADDRESSES

Memory addresses in OPL are held as integers. Pack addresses are held in 3 bytes. In the CM operating system the most significant byte is ignored.

INTEGERS

An integer is a number between 32767 and -32768. It is stored in memory as a single word. In the source code of the language an integer may be input in hexadecimal by preceding the number by a '$', so $FFFF is a valid number and equal to -1.

A number in an OPL program will be taken as integer if it is in the integer range with the one exception, -32768 is taken as a floating point number. The reason for this is that the translator translates a negative number as the absolute value, followed by a unary minus operator. 32768 is outside the range for integers and so is translated as a floating point number. A small increase in speed and compactness can be obtained by writing negative integers in hexadecimal.

It is very important to anticipate what is taken as integer. For example:

30001/2 is the integer 15000 but 40001/2 is floating point number 20000.5.

To ensure that a number is taken as a floating point number just add a trailing period. '2' is an integer, '2.' is a floating point number.

The calculator translates numbers as floating point. If you wish to put an integer into the calculator you must use the function INT. So, for example, from the calculator:

```
PRICE:(INT(10))
```

passes the integer 10 to the procedure PRICE.

FLOATING POINT

Floating point numbers are in the range +/-9.99999999999E99 to +/-1E-99. They are held in Binary Coded Decimal (BCD) in 8 bytes; 6 bytes for the mantissa, 1 byte for the exponent, and 1 for the sign.

The decimal number -153 is held as:

```
00 00 00 00 30 15 02 80
```

where the last byte is the sign byte (either 00 or 80) and the preceding byte the exponent.

The decimal number .0234567 is held as:

```
00 00 00 67 45 23 FE 00.
```

It is possible for the exponent to go out of range, e.g. 1E99*10 or 1E-99/10. This is reported as an EXPONENT RANGE error.

When floating point numbers are translated they are held in a more compact form. The first byte contains both the sign, in the most significant bit, and the number of bytes following. The next bytes are the significant bytes of the mantissa, the final byte is the exponent.

In Q code the decimal number -153 is represented as:

```
83 30 15 02.
```

The decimal number .0234567 is represented as:

```
04 47 45 23 FE
```

This compact form is always preceded by a QI_STK_LIT_NUM operator.

STRINGS

Strings are up to 255 characters long, with a preceding length byte. The string "QWERTY" is held as:

```
06 51 57 45 52 54 59
```

All string variables, except field strings, are preceded by that variable's maximum length, as declared in the LOCAL or GLOBAL statement.

All strings in OPL have this format. For example when using USR$ the machine code should return with the X register pointing at the length byte of the string to be returned.

ARRAYS

One dimensional arrays are supported for integers, floating point numbers and strings. Multi-dimensional arrays can be easily simulated by the use of integer arithmetic.

Like all other variables, arrays are held on the stack. In the case of string arrays the maximum string length is the first byte, the next word contains the array size, this is followed by data. So, for example,

```
LOCAL A$(5,3),B%(2),C(3)
A$(4)="AB"
C(1)=12345
```

initially sets up memory as follows (from low memory to high memory):

```
High memory     00 00 00 00             ; 5th element of A$()
                00 00 00 00             ; 4th element of A$()
                00 00 00 00             ; 3rd element of A$()
                00 00 00 00             ; 2nd element of A$()
                00 00 00 00             ; 1st element of A$()
                00 05                   ; array size of A$()
                03                      ; max string length of A$()
                00 00                   ; 2nd element of B%()
                00 00                   ; 1st element of B%()
                00 02                   ; array size of B%()
                00 00 00 00 00 00 00 00 ; 3rd element of C()
                00 00 00 00 00 00 00 00 ; 2nd element of C()
                00 00 00 00 00 00 00 00 ; 1st element of C()
 Low memory     00 03                   ; array size of C()
```

After running the procedure it looks like:

```
High memory     00 00 00 00             ; 5th element of A$()
                02 41 42 00             ; 4th element of A$()
                00 00 00 00             ; 3rd element of A$()
                00 00 00 00             ; 2nd element of A$()
                00 00 00 00             ; 1st element of A$()
                00 05                   ; array size of A$()
                03                      ; max string length of A$()
                00 00                   ; 2nd element of B%()
                00 00                   ; 1st element of B%()
                00 02                   ; array size of B%()
                00 00 00 00 00 00 00 00 ; 3rd element of C()
                00 00 00 00 00 00 00 00 ; 2nd element of C()
                00 00 00 50 34 12 04 00 ; 1st element of C()
 Low memory     00 03                   ; array size of C()
```

The string and array limits are inserted into the variable space after it has been zeroed. This process is referred to as "fixing up" the variables.

Only available memory limits the size of arrays.

TYPE CONVERSION

Automatic type conversion takes place where possible. For instance:

```
A=10
```

and

```
A=FLT(10)
```

produce exactly the same Q code. Whereas:

```
A=10.
```

has different Q code. All three place the floating point number 10 into the variable A.

When expressions are evaluated the standard left to right rule is applied with type integer being maintained as long as possible. So, for example:

```
A=1000*1000*1000.
```

generates an "INTEGER OVERFLOW" error. But:

```
A=1000.*1000*1000
```

does not. This applies to any sub-expressions inside brackets, so:

```
A=1000.*(1000*1000)
```

generates the overflow error.

Another side effect is that that divisions of only integers will have an integer result:

```
(2/3) = 0
(2./3) = 0.66666666667
```

NOTE VERY WELL: In the calculator all numeric constants are automatically converted to floating point. So in the calculator NOT(3) evaluates to 0, whereas NOT(INT(3)) is -4.

Note also: Outside the calculator a simple number is taken as an integer if is is less than 32768 and more than -32768, so in a procedure 10**10 gives an INTEGER OVERFLOW error.

COMPARISONS

NOT, AND, and OR are bitwise on integers, but on floating point numbers they are logical. So the following equalities are true:

```
(NOT 3.0) = 0           (NOT 3) = -4
(3.0 AND 5.0) = -1      (3 AND 5) = 1
(3.0 OR 5.0) = -1       (3 OR 5) = 7
```

The string compares are case sensitive.

RECORDS AND FIELDS

A file consists of a file name record with a number of data records.

A record contains at least one character and at most 254 characters. A record may contain up to 16 fields, delimited by the TAB character (ASCII 9).

Strings are held as the ASCII characters, numbers are held in the ASCII form. So for example after:

```
OPEN "A:ABC",A,A%,B,C$
A.A=12
A.B=3.4
A.C$="XYZ"
```

the file buffer contains:

```
 len         tab           tab
 0A   31 32  09  33 2E 34  09  58 59 5A.
```

When a file is opened the field names are given. The field names and types are not fixed and may be varied from OPEN to OPEN. When a numeric field is accessed the contents are converted from ASCII to integer or floating point. Should this conversion fail the error "STR TO NUM FAIL" is reported.

When searching for a particular field the field name is matched with the field name buffer and the corresponding field split out of the file buffer using UT$SPLT.

Note that any string can be assigned to a string field but that if it includes a TAB character it will generate an extra field. For example:

```
OPEN "A:ABC",A,A$,B$,C$
A.B$="Hello"
A.A$="AB"+CHR$(9)+"CD"
PRINT A.C$ GET
```

will print "Hello" to the screen. The file buffer contains:

```
0B 41 42 09 43 44 09 48 65 6C 6C 6F
```

Saving data in ASCII is simple but it is easy to see how data can be compressed by using BCD, hexadecimal or other techniques.

VARIABLE SCOPE

When a procedure is loaded all the LOCALs and GLOBALs declared in it are allocated space on the stack. This area is zeroed and the strings and arrays are fixed up. In other words, the maximum length of each string and the array sizes are filled in.

These variables remain in memory at fixed locations, until execution of the declaring procedure terminates. LOCAL variables are valid only in that procedure, whereas GLOBAL variables are valid in all procedures called by the declaring procedure.

EXTERNALS

If a variable used in a procedure is not declared LOCAL or GLOBAL in that procedure it is taken as external. The Q code contains a list of externals and these are resolved at run time.

Using the frame pointer, the previous procedures are checked for all entries in the GLOBAL tables. If a match is found the variable address is inserted in an indirection table. If an external is not found it is reported as an error.

Note that neither the LOCAL names nor the parameter names are present in the Q code, but that GLOBAL names are.

LANGUAGE POINTERS

There are three key pointers used by the language:

- RTA_SP Language stack pointer
- RTA_PC Program counter
- RTA_FP Frame (procedure) pointer

RTA_SP points at the lowest byte of the stack. So if an integer is stacked, RTA_SP is decremented by 2 and the word is saved at the address pointed to by RTA_SP.

RTA_PC points at the current operand/operator executed and is incremented after execution - except at the start of a procedure or a GOTO when RTA_PC is set up appropriately.

RTA_FP points into the header of the current procedure.

Each procedure header has the form:

- Device (zero if top procedure)
- Return RTA_PC
- ONERR address
- BASE_SP

RTA_FP points at:

- Previous RTA_FP
- Start address of the global name table
- Global name table
- Indirection table for externals/parameters

This is followed by the variables, and finally by the Q code.

RTA_FP points at the previous RTA_FP, so it is easy to jump up through all the procedures above. The language uses this when resolving external references and when handling errors.

ADDRESSING MODES

Local variables and global variables declared in the current procedure are accessed directly. A reference to such variables is by an offset from the current RTA_FP.

Parameters and externally declared global variables are accessed indirectly. The addresses of these variables are held in the indirection table, the required address in this table is found by adding the offset in the Q code to the current RTA_FP.

TOP LOOP

Each procedure consists of two parts, a header and Q code. The Q code contains all the operands and operators in a table that is run by the TOP LOOP.

The TOP LOOP controls the language, it performs the following functions:

1. Increment RTA_PC by the B register
2. Test for the ON/CLEAR key
3. Test for low battery
4. Load and execute the next operand/operator
5. Test carry - if set then initiate error handling

FILES

CREATING

Before a file is created a check is made that no file exists with the specified name on that device. The first unused record number over $90 is assigned to the file and the file name record is written to the device. The process then continues in the same way as opening a file.

The file name records are type $81. The file name record for a file called "AMANDA", with record file type $95 looks like:

```
09 81 41 4D 41 4E 44 41 20 20 20 95
```

OPENING

First the file name record is located to ensure that the file exists. The file record type and the device on which the file was found are saved in the file block (RTT_FILE). The field names are saved in the allocator field name cell corresponding to the logical name and the file buffer cell is expanded to 256 bytes. The record position is initialised to 1 and the first record, if it exists, is read.

If the file has just been created or the record is empty the current record will be null and the EOF flag is set.

LOGICAL FILE NAMES

Up to 4 files may be open at one time; to distinguish between then logical file names are used. The 4 logical file names: A,B,C, and D, are used to determine which file is to be operated on by the file commands.

This means that you can open files in any order but have a constant way of referring to them. The USE operator selects which file is affected by the following commands:

APPEND BACK CLOSE ERASE

FIRST NEXT LAST POSITION

UPDATE

and the following functions:

COUNT DISP EOF FIND

POS RECSIZE SPACE

USING FILES

There is no functional difference between the logical file names.

When opening a file the file name record and the first record are located; two cells, one a buffer and one for the field names are grown. Closing a file entails the two cells being shrunk.

All references to fields must include the logical file name. This serves two purposes; it allows statements such as "A.MAX=B.VALUE" and it allows the language to distinguish between ordinary variables and field names.

PROCEDURE CALLS

To write compact, fast code it is important to understand the way procedures are loaded and automatically overlaid.

A procedure call consists of a procedure name followed by up to 16 parameters. The procedure name may include an optional '$' or '%' but must terminate with a ':'. If parameters are supplied they must be separated by commas and be enclosed in brackets.

There are two main types of procedure. In standard OPL procedures the Q code is loaded onto the stack and then executed. The second type are known as a device procedure or language extensions; they are identical to standard procedures in appearance, but differs in that it is recognised by the device lookup and runs as self-contained machine code.

STANDARD PROCEDURES

When a QCO_PROC operator is encountered the parameters will already be on the stack, along with the parameter count and the parameter types. After the operator is the name of the procedure.

The following list of actions are then carried out:

1. Check if it is a language extension/device call
2. Search for the procedure starting with the default device
3. Check that there is sufficient memory
4. Set new RTA_SP, RTA_FP
5. Check the parameter count
6. Check the parameter types
7. Set up a table of variables declared GLOBAL
8. Set up the parameter table
9. Resolve the externals, build an externals table
10. Zero all variable space
11. Fix-up strings
12. Fix-up arrays
13. Load the code
14. Set new RTA_PC

The code is loaded every time a procedure is called. This means that recursive procedures are allowed but that the stack will grow by the size of the Q code + data space + overhead for each call. On an XP, following a Reset, the procedure:

```
RECURS:(I%)
IF I%
 RECURS:(I%-1)
ENDIF
```

allows values up to 315 before an 'OUT OF MEMORY' error is given.

LANGUAGE EXTENSIONS

Language extension are also referred to as device procedures. Examples are LINPUT, LSET and LTRIG in the RS232 interface.

To test if a procedure is a language extension, call DV$LKUP. This looks through the devices loaded in order of priority. If a language extension is found it returns with carry clear, the device number in the A register and the vector number in the B register, suitable for an immediate call to DV$VECT to run the code.

The machine code should check that any parameters that have been passed are correct, do whatever it has to do, add the return variable to the stack and return. It is essential to return the right variable type. If the extension name terminates with a '$' it must return a string, if with a '%' it requires an integer, otherwise an 8 byte floating point number.

Note that a variable number of parameters can be passed to a device.

As a simple example, consider a language extension to add two integers without giving an error if the sum overflows. If only one parameter is given the value is simply incremented, again without giving an error. The assembler for this extension called "ADD%" is:

```
XADD:
        LDX     RTA_SP:
        LDA     A,0,X
        BEQ     1$                      ; wrong number of parameters
        DEC     A
        BEQ     INCREM                  ; increment 1 parameter
        DEC     A
        BEQ     XXADD                   ; add the two
1$:
        LDA     B,#ER_RT_NP             ; wrong number of parameters
        SEC                             ; bad return
        RTS
INCREM:
        LDA     A,1,X                   ; load parameter type
        BNE     WRGTYP                  ; branch if not integer
        LDD     2,X
        ADDD    #1
EXIT:
        DEX
        DEX
        STX     RTA_SP:
        STD     0,X                     ; save return value
        CLC                             ; good return
        RTS
XXADD:
        LDA     A,1,X
        BNE     WRGTYP                  ; branch if not integer
        LDA     A,4,X
        BNE     WRGTYP                  ; branch if not integer
        LDD     2,X                     ; and add the two integers
        ADDD    5,X
        BRA     EXIT
WRGTYP:
        lda     b,#ER_FN_BA             ; report wrong parameters type
        SEC                             ; bad return
        RTS
```

WRITING OPL

Like any programming language there is an infinite number of approaches to every problem. The aim should be to produce fast, compact Q code that runs in a minimum of memory but is also easy to write and understand. These aims inevitably conflict with each other; the correct balance varies from application to application.

For example, the decision to use a separate procedure, rather than writing the code in line, is a matter of considering the difference in Q code size, the extra stack required at run time, the time overhead required to load and return from a procedure and finally style.

It is impossible to give definitive rules on writing code but it is worth taking the following points into account.

COMPACT Q CODE

1. Only use procedures where appropriate
2. If it makes no difference, use Locals instead of GLOBALs
3. Use short field names
4. Use short global names
5. If you repeatedly use a CHR$ with the same value, assign it to a variable
6. Use "RETURN" instead of "RETURN 0" or "RETURN """
7. Use hexadecimal integers instead of negative integers

COMPACT ON RUN TIME

1. Write short Q code (as above)
2. Use a small main procedure to call several small procedures.
3. Use integers instead of floating point numbers
4. Use short field names
5. Use short global names

6. Check the deepest part of the code by adding, temporarily, PRINT FREE :GET. Then consider restructuring the procedures to decrease the amount of stack used.

FAST CODE

1. don't use too many procedures, regard them as being similar to overlays
2. place the procedures at the beginning of the pack, with the most frequently used at the start
3. Use Locals or GLOBALs rather than field variables
4. Don't use procedures inside time critical loops, write the code in-line
5. Use integers instead of floating point numbers
6. Write short Q code (less code to load)
7. Use Locals instead of GLOBALs

Each operand/operator has an overhead of .05 ms. Most integer based operands/operators are very fast and run in less than .1 ms.

The following timings are rough and should only be used as a guide:

OPERAND	Time (ms)
RND	10
AT	0.15
PRINT a string	0.5
INT_TO_NUM	2.5
NUM_TO_INT	2
SIN/COS	150
TAN	350
ATAN	170
SQR	240
EXP	130
LOG/LN	200
Integer add/subtract	0.1
Integer multiply/divide	1
Floating point add/subtract	3
Floating point multiply	10
Floating point divide	20
Accessing a field	5

PRINT_CR has a default delay of 500 milliseconds. This value can be altered by poking the value in DPW_DELY.

PROCEDURES

The smallest time overhead on loading, and returning form a procedure is 8 ms. This overhead increases if the procedure follows other blocks or records on the device. It also increases if the procedure is not on the same device as the top level procedure (as it will have to search that device first).

FILES

Some of the file operators have to count up the pack each time they are used. For the sake of speed NEXT remembers its position on each of the packs. However it only remembers one position on each pack so:

```
USE B
NEXT
A.MAX=B.VAL
USE A
APPEND
```

where file A is on B: and file B on C: is significantly faster than if they are both on the same device.

BACK however always has to count up the pack to locate a record and this can take a noticeable time. Remember that erased records, as well as readable ones, will slow down the location of a record.

CODE STYLE

Before starting to write a program (which normally will consist of a number of procedures) first decide the relative importance of speed of execution, compactness of the Q code and the amount of stack used.

Then *rough out the procedure structure*. For example, in the case of the finance pack the main procedure is called FINS:

```
fins:
local i%,j%
do
  i%=menu("BANK,EXPENSES,NPV,IRR,COMPOUND,BOND,MORTGAGE,APR,END")
  if     i%=1 : bank:
  elseif i%=2 : expenses:
  elseif i%=3 : npv:
  elseif i%=4 : irr:
  elseif i%=5
    do
      j%=menu("VALUE,FUTURE,PAYMENT,DURATION,INTEREST,END")
      if     j%=1 : value:
      elseif j%=2 : future:
      elseif j%=3 : payment:
      elseif j%=4 : duration:
      elseif j%=5 : interest:
      endif
    until j%=0 or j%=6
  elseif i%=6 : bond:
  elseif i%=7 : mortgage:
  elseif i%=8 : apr:
  endif
until i%=0 or i%=9
```

Your style may vary if you are writing on the emulator or the ORGANISER itself. On the ORGANISER it is worth, as a general rule, making only limited use of the ':' option to have more than one statement on a line. On the emulator you may prefer to write multiple statements on a line. The procedure above was written using a full screen editor which is reflected in the elegant use of non-functional spaces.

It is very helpful to indent the code by logical function. This is very useful in matching IF/ENDIF and loop commands.

Comment the code. The logic may seem very obvious when you write it but other people may want to read it, or you may return to the code after several months. In most cases the extra space taken by the comments is well worth it. Remember that comments make no difference to the Q code size.

Use brackets if you are unsure of the operator precedence. This adds nothing to the Q code size but makes your intentions absolutely clear.

When using the ':' separator it is not necessary to precede it by a space when the preceding characters cannot be taken as a variable name. So "A%=1:B%=2" is valid but "A%=B%:B%=C%" gives a syntax error. It can, however, save time and make the code more readable if you always proceed the ':' separator with a space.

TRANSLATOR

The translator scans the source code, statement by statement, translating it into Q code. All expressions are converted to reverse polish (postfix) form so that, at run time, the operators can be executed as soon as they are encountered.

It is beyond the scope of this document to describe the detailed working of the translator. Fortunately, such a description is not necessary in order to understand either the execution of the code or the writing of efficient code.

SYSTEM SERVICES INTERFACE

RM$RUNP

Runs the language by loading and running the OPL procedure. The procedure can not have any parameters.

LN$STRT

Runs the translator, either

- translating language procedures
- translating CALC expressions
- locating errors in CALC
- locating errors in language procedures

LN$XSTT (LZ only)

Acts like LN$STRT except that there is a choice whether the source is to be translated in 2-line mode or

LG$ENTR (LZ only)

Provides an entry point to the PROG application in the top-level menu. There are 2 functions available:

Function 1 - Call the PROG application as from the top-level menu.

Function 2 - Search block files of a given type that are on packs for a given string.

MACHINE CODE INTERFACE

From the information in this chapter, the programmer knows exactly where everything is on the stack.

When variables are declared they are used in order, so:

```
LOCAL A%,B%
PRINT ADDR(A%)=ADDR(B%)+2
GET
```

will print -1, i.e. TRUE.

For short machine code routines you can use this crude, but effective, procedure:

```
LOADR:(ADDR%,CODE$)
LOCAL A%,B1%,B2%,I%
A%=ADDR%
I%=1
WHILE I%9 :B1%=B1%-7 :ENDIF
  B2%=ASC(MID$(CODE$,I%+1,1))-%0
  IF B2%>9 :B2%=B2%-7 :ENDIF
  POKEB A%,B1%*16+B2%
  A%=A%+1
  I%=I%+2
ENDWH
```

When calling this procedure you must pass the machine code in digital form and the address where to put the machine code. It is essential that the programmer ensures there is enough room for the machine code at the address given.

A calling sequence might look like:

```
MAIN:
GLOBAL MC%,MC$(10)
MINIT: :REM Initialise the machine code
..
CELL%=USR(MC%,100) :REM GRABs a cell of size 100
IF CELL%=0
 PRINT "No cell free"
 GET :RAISE 0
ENDIF
..
RETURN
MINIT:
A$="3F012403CE000039"
IF LEN(A$)/2>LEN(MC$)
  PRINT "Not enough room for MC"
  GET :RAISE 0
ENDIF
MC%=ADDR(MC$)
LOADR:(MC%,A$)
```

The machine code is:

```
        OS      AL$GRAB
        BCC     1$
        LDX     #0
1$:     RTS
```

EXCEPTION HANDLING

ERROR HANDLING

When an error is first detected the following actions are taken:

1. The error saved in RTB_EROR
2. If the TRAP flag is set then the language continues
3. The ON_ERR address for that procedure and each procedure above is tested. If one is found to be non-zero, RTA_PC is set to that value and RTA_SP set to the BASE_SP for that procedure. The language then continues on.
4. If no error handling is detected then the error is reported along with the name of the procedure in which the error was detected and the language exits.

If the error is ER_RT_UE (undefined external) then the externals which are undefined are displayed with DP$VIEW.

If the error is ER_RT_PN (procedure not found) then the name of the procedure not found is displayed (as well as the procedure where it was called).

OUT OF MEMORY

Every time round the top loop the difference between RTA_SP and ALA_FREE is calculated. If this difference is less than 256 bytes, "OUT OF MEMORY" is reported. Note that no operand or operator can grow the stack by more than 256 bytes.

The filing system can also generate the "PACK FULL" error if it detects that after an operation fewer than 256 bytes will be free on device A. In this case it means essentially the same thing as "OUT OF MEMORY".

The only time when OPL uses memory, other than on the stack, is when it opens files.

LOW BATTERY

If the voltage goes below the threshold value (5.2 volts) while the language is running, it is detected either in the top loop or during the execution of an operator. In either case it is treated as a standard error. If no error handling is in force, the error is reported and the machine turns off.

If the error is handled by an ONERR, the low battery error number is saved in RTB_EROR. It is not reported again by the top level until the battery voltage has gone back above the minimum voltage. This allows the procedure to take some action (e.g. to turn the organiser off). If the procedure just continues on the battery will eventually die completely and there is a risk of having to cold boot the machine.

Note that the battery is more likely to drop below the threshold voltage when devices, such as the packs or the RS232 interface, are switched on because they drain substantially more current than the Organiser by itself. See section power supply for more details of the power drain of different devices. Also note that a battery naturally recovers some of its power after being turned off for a while.

ON/CLEAR KEY

In normal operation pressing the ON/CLEAR key results in the execution of the language being frozen until another key is pressed. If the key pressed is 'Q','q' or '6' it creates an error condition ER_RT_BK. If there is no user error handling, execution of the language will terminate.

If ESCAPE OFF has been executed then the ON/CLEAR key has no special effect.

In an input statement then the ON/CLEAR key acts in one of 3 different ways:

1. If there is any input it is cleared
2. Or if the TRAP option has been used then the input exits with the error condition ER_RT_BK
3. Otherwise it is ignored

SYSTEM CRASHES, INFINITE LOOPS

OPL is a powerful flexible language and as such it has the potential to crash the operating system or get into an infinite loop. This is particularly unfortunate in the case of the ORGANISER because all the data held in device A: is lost when the machine reboots. For extensive development of 'dangerous' routines a RAMPACK has a lot to recommend it.

There are trivial ways to crash such as poking system variables or using USR function with wrong addresses or bad machine code. It is impossible to describe all the other ways in which such problems can arise. The examples listed below show the most obvious ways in the simplest possible form.

- ESCAPE OFF :DO :UNTIL 0 :REM Impossible to get out
- WHILE GET :ENDWH :REM Hard to get out of
- DO :KEY :UNTIL 0 :REM Hard to get out of
- A:: ONERR A:: :RAISE 0 :REM Impossible to get out
- A:: ONERR A:: :DO :UNTIL 0 :REM Impossible to get out

Error handling is best added at the end of a development cycle. Turning ESCAPE OFF substantially increases the chances of getting into an infinite loop from which there is no exit.

MODEL LZ

COMPATIBILITY

The LZ is fully back-compatible with 2-line Organisers, so any existing OPL programs will run exactly the same on the LZ as they do on the CM or XP but will use 2 lines in the center of the screen with a border around.

When OPL programs which have been translated on a 2-line Organiser (CM, XP, etc.) are run either from the top level menu or under PROG the machine is automatically put into "2-line compatibility mode".

The mode in which the OPL program runs is determined by the **first** OPL procedure run. Any subsequent OPL procedures which are loaded will run in the same mode.

The OPL object code generated when translated on an LZ contains a STOP code followed by a SINE code at the front of the procedure. Thus the object code of all 4-line procedures will be two bytes longer than the same procedure translated on a 2-line machine. The STOP/SINE configuration is used for 2 reasons:

1. The combination will identify the procedure as 4-line since it can never be generated on a 2-line Organiser.
2. When run on a 2-line machine, the procedure will cause OPL to stop running, i.e. terminate the OPL program.

Note that LZ machines can translate procedures as though they were translated on a 2-line Organiser using the "XTRAN" option in the PROG EDITOR menu.

It follows from this that 2-line code can run in 4-line mode providing the initial procedure is 4-line, but 4-line code can never run in 2-line mode.

APPLICATIONS FOR 2 AND 4 LINE MACHINES

To create an OPL application which will run on both types of machine and make use of all 4 lines when run on an LZ, use the following method:

1. The main procedure should be translated in 2-line mode and should contain the following code:

```
          LOCAL M%(2)
          M%(1)=$3F82 :REM OS DP$MSET
          M%(2)=$3900 :REM RTS
          IF PEEKB($FFE8) AND 8)=8 AND (PEEKB($FFCB) AND $80)=$80
            USR(ADDR(M%()),256) :REM switch to 4-line mode if LZ
```

```
    ENDIF
```

   This code will switch to 4-line mode if it is run on an LZ but will do nothing if run on a 2-line machine.
7. Code which can be used on both 2 and 4 line machines (generally code that does not print to the screen or use LZ only features) should be put in subroutines and translated in 2-line mode.
8. Code to print to the 2 line screen should be put in one procedure and code to print to the 4 line screen in another - translated in 2-line mode and 4-line mode respectively.
9. The main code must read DPB_MODE (address $2184) to decide which OPL procedures to call to print in the correct mode.

For example the following program will print "HELLO" on the 2nd line of a 2-line machine and on the 4th line of an LZ. The first two modules have to be translated in 2-line mode and the 3rd one in 4-line mode:

```
MAIN:
LOCAL M%(2)
M%(1)=$3F82 :REM OS DP$MSET
M%(2)=$3900 :REM RTS
IF PEEKB($FFE8) AND 8)=8 AND (PEEKB($FFCB) AND $80)=$80
  USR(ADDR(M%()),256) :REM switch to 4-line mode if LZ
ENDIF
IF PEEKB($2184)
  HELLO4: :REM call "HELLO4" if 4-line mode
ELSE
  HELLO2: :REM else call "HELLO2"
ENDIF
GET
HELLO4:
AT 1,4 :PRINT "HELLO"
HELLO2:
AT 1,2 :PRINT "HELLO"
```

ADDITIONAL COMMANDS AND FUNCTIONS

OPL has been extended on the LZ. Some existing commands and functions have been extended to use the 4 lines (e.g. AT 20,4, VIEW(4,A$) etc.) and some new commands and functions have been added.

The OPL commands and functions which are not available on the CM or XP are as follows:

- OFF x% - Turns the Organiser off for a limited time only.
- UDG - Defines a display character (user-defined graphic).
- COPYW - Copies any type of file with wild card matching.
- DELETEW - Deletes any type of file with wild card matching.
- CLOCK - Displays the "UDG CLOCK".
- MENUN - Displays the different types of menu.
- DIRW$ - Returns name of any type of file with wild card matching.
- FINDW - Like FIND but allows the use of wild cards.
- ACOS - Returns the arc cosine of a number.
- ASIN - Returns the arc sine of a number.
- MAX - Returns the greatest item in a list.
- MIN - Returns the smallest item in a list.
- MEAN - Returns the mean of items in a list.

- STD - Returns the standard deviation of items in a list.
- SUM - Returns the sum of items in a list.
- VAR - Returns the variance of items in a list.
- DAYS - Returns the number of days since 01/01/1900
- DAYNAME$ - Converts 1 - 7 to the day of the week.
- DOW - Returns the day a date falls on as a number 1 - 7.
- MONTH$ - Converts 1 - 12 to the month.
- WEEK - Returns the week number a date falls in.

Note that XTRAN will give an error if used to translate any of the above commands.

Q-CODE

OPERANDS

Each operand stacks either a constant value or a pointer to a variable.

There are a number of types of operands. Operands are named after their type, the types are:

Integer	INT
Floating point	NUM
String	STR
Constants (i.e. not variables)	CON
Arrays	ARR
Simple (i.e. not array)	SIM
Offset from RTA_FP	FP
Indirect offset from RTA_FP	IND
Left side (i.e. assigns)	LS
Field	FLD
Stack byte/word	LIT
Refer to the fixed memories	ABS

Internal Name	Op + Bytes		Added to the stack
QI_INT_SIM_FP	$00	2	The integer
QI_NUM_SIM_FP	$01	2	The floating point number
QI_STR_SIM_FP	$02	2	The string

These operands take the following word, add it to RTA_FP (see section Language Pointers) and stack the variable at that address.

Internal Name	Op + Bytes		Stack
QI_INT_ARR_FP	$03	2	Drops element number, adds an integer from the array
QI_NUM_ARR_FP	$04	2	Drops element number, adds a floating point number from the array
QI_STR_ARR_FP	$05	2	Drops element number, adds a string from the array

These operands take the following word, adds it to RTA_FP to get the start of the array. The required element number is dropped off the stack and checked against the maximum size of the array. The address of the element is then calculated and the variable stacked.

Internal Name	Op + Bytes		Added to the stack
QI_NUM_SIM_ABS	$06	2	Floating point number

This operand gives access to the calculators memories, M0 to M9. The operand is followed by the offset to the memory required.

Internal Name	Op + Bytes	Added to the stack
QI_INT_SIM_IND	$07 2	The integer
QI_NUM_SIM_IND	$08 2	The floating point number
QI_STR_SIM_IND	$09 2	The string

These operands take the following word, add it to RTA_FP, load the address at that address and stack the variable at that address.

Internal Name	Op + Bytes	Stack
QI_INT_ARR_IND	$0A 2	Drops element number, adds the integer from the array
QI_NUM_ARR_IND	$0B 2	Drops element number, adds the floating point number from the array
QI_STR_ARR_IND	$0C 2	Drops element number, adds the string from the array

These operands take the following word, adds it to RTA_FP, loads the address at that address to get the start of the array. The element of the array required is dropped off the stack, it is then checked against the maximum size of the array. The address of the element is then calculated and the variable stacked.

Internal Name	Op + Bytes	Added to the stack
QI_LS_INT_SIM_FP	$0D 2	The address of the integer + field flag
QI_LS_NUM_SIM_FP	$0E 2	The address of the floating point number + field flag
QI_LS_STR_SIM_FP	$0F 2	The maximum size + the address of the string + field flag
QI_LS_INT_ARR_FP	$10 2	The address of the integer from the array + field flag
QI_LS_NUM_ARR_FP	$11 2	The address of the floating point number from the array + field flag
QI_LS_STR_ARR_FP	$12 2	The maximum size + the address of the string from the array + field flag
QI_LS_NUM_SIM_ABS	$13 2	The address of the calculator memory + field flag
QI_LS_INT_SIM_IND	$14 2	The address of the integer + field flag
QI_LS_NUM_SIM_IND	$15 2	The address of the floating point number + field flag
QI_LS_STR_SIM_IND	$16 2	The maximum size + the address of the string + field flag
QI_LS_INT_ARR_IND	$17 2	The address of the integer from the array + field flag
QI_LS_NUM_ARR_IND	$18 2	The address of the floating point number from the array + field flag
QI_LS_STR_ARR_IND	$19 2	the maximum size + the address of the string from the array + field flag

These operands correspond to their right side equivalents. In the case of strings the maximum length is stacked first. Then, in all cases, the address of the variable is stacked. The field flag byte is then stacked, in all these cases it is zero to show that it is not a field reference.

Internal Name	Op + Bytes	Stack
QI_INT_FLD	$1A 1	Drops the field name, adds the integer
QI_NUM_FLD	$1B 1	Drops the field name, adds the floating point number
QI_STR_FLD	$1C 1	Drops the field name, adds the string

These operands are followed by a logical file name, 0,1,2 or 3, which says which logical file to use. First it looks for the field name in the Field Name Symbol Table. If it is found the corresponding field is split from the corresponding File Buffer. If it is a string it is immediately placed on the stack. If it is numeric it is converted from ASCII to the relevant format and placed on the stack.

Internal Name	Op + Bytes	Stack
QI_LS_INT_FLD	$1D 1	Stacks the logical file name + field flag
QI_LS_NUM_FLD	$1E 1	Stacks the logical file name + field flag
QI_LS_STR_FLD	$1F 1	Stacks the logical file name + field flag

These operands stacks the logical file, the byte following the operand, and the field flag which in this case is non-zero. All the work is done by the assign.

Internal Name	Op + Bytes	Added to the stack
QI_STK_LIT_BYTE	$20 1	The byte
QI_STK_LIT_WORD	$21 2	The word

Stacks the following byte or word. QI_STK_LIT_WORD is identical to QI_INT_CON.

Internal Name	Op + Bytes	Added to the stack
QI_INT_CON	$22 2	Integer
QI_NUM_CON	$23 *	Floating point number
QI_STR_CON	$24 *	String

Stacks the constant value following.

OPERATORS

Operators generally do things to the variables already on the stack.

ERRORS, CALLS AND PARAMETERS

In the following section if an operand cannot return an error then no errors are listed.

Any access to a device can result in the following errors. They are no given explicitly as error for that operand/operator:

ER_FL_NP - no pack
ER_PK_IV - unknown pack
ER_DV_CA - bad device name and if the pack was not blank
ER_PK_NB - pack not blank

When writing to a pack the following are always possible:

ER_FL_PF - pack full
ER_PK_RO - read only pack
ER_PK_DE - write error

If the operator calls an operating system then that is listed. If no calls are given then the run time code handles it all itself. In general there is no difference between call with a $ and with an _, the $ calls are called through SWIs whereas the _ calls are made directly. Direct calls are faster, but SWIs can be redirected for the addition of extra features.

If there is more than one parameter they are listed. The values are stacked in order. So para1 is stacked before para2 - when the operator is called the last parameter is the one pointed to by the RTA_SP.

LOGICAL AND ARITHMETIC COMPARE OPERATORS

Internal Name	Op	Stack
QCO_LT_INT	$27	Drops 2 INTs, returns 0 or -1 as an INT
QCO_LTE_INT	$28	Drops 2 INTs, returns 0 or -1 as an INT
QCO_GT_INT	$29	Drops 2 INTs, returns 0 or -1 as an INT
QCO_GTE_INT	$2A	Drops 2 INTs, returns 0 or -1 as an INT
QCO_NE_INT	$2B	Drops 2 INTs, returns 0 or -1 as an INT
QCO_EQ_INT	$2C	Drops 2 INTs, returns 0 or -1 as an INT
QCO_ADD_INT	$2D	Drops 2 INTs, returns result as an INT
QCO_SUB_INT	$2E	Drops 2 INTs, returns result as an INT
QCO_MUL_INT	$2F	Drops 2 INTs, returns result as an INT
QCO_DIV_INT	$30	Drops 2 INTs, returns result as an INT
QCO_POW_INT	$31	Drops 2 INTs, returns result as an INT
QCO_UMIN_INT	$32	Drops an INT, returns result as an INT
QCO_NOT_INT	$33	Drops an INT, returns result as an INT
QCO_AND_INT	$34	Drops 2 INTs, returns result as an INT
QCO_OR_INT	$35	Drops 2 INTs, returns result as an INT
QCO_LT_NUM	$36	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_LTE_NUM	$37	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_GT_NUM	$38	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_GTE_NUM	$39	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_NE_NUM	$3A	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_EQ_NUM	$3B	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_ADD_NUM	$3C	Drops 2 NUMs, returns result as an NUM
QCO_SUB_NUM	$3D	Drops 2 NUMs, returns result as an NUM
QCO_MUL_NUM	$3E	Drops 2 NUMs, returns result as an NUM
QCO_DIV_NUM	$3F	Drops 2 NUMs, returns result as an NUM
QCO_POW_NUM	$40	Drops 2 NUMs, returns result as an NUM
QCO_UMIN_NUM	$41	Drops a NUM, returns result as an NUM
QCO_NOT_NUM	$42	Drops a NUM, returns 0 or -1 as an INT
QCO_AND_NUM	$43	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_OR_NUM	$44	Drops 2 NUMs, returns 0 or -1 as an INT
QCO_LT_STR	$45	Drops 2 STRs, returns 0 or -1 as an INT
QCO_LTE_STR	$46	Drops 2 STRs, returns 0 or -1 as an INT
QCO_GT_STR	$47	Drops 2 STRs, returns 0 or -1 as an INT
QCO_GTE_STR	$48	Drops 2 STRs, returns 0 or -1 as an INT
QCO_NE_STR	$49	Drops 2 STRs, returns 0 or -1 as an INT
QCO_EQ_STR	$4A	Drops 2 STRs, returns 0 or -1 as an INT

QCO_ADD_STR $4B Drops 2 STRs, returns result as a STR

The compares drop whatever is on the stack and return an integer either TRUE(-1) or FALSE(0).
The string compares are case sensitive.

PERCENTAGE OPERATORS

These operators are only available on the LZ model range.

Internal Name	Op	Stack
QCO_LT_PERC	$CC	Drops 2 FLTs, returns result as a FLT
QCO_GT_PERC	$CD	Drops 2 FLTs, returns result as a FLT
QCO_ADD_PERC	$CE	Drops 2 FLTs, returns result as a FLT
QCO_SUB_PERC	$D0	Drops 2 FLTs, returns result as a FLT
QCO_MUL_PERC	$D1	Drops 2 FLTs, returns result as a FLT
QCO_DIV_PERC	$D2	Drops 2 FLTs, returns result as a FLT

COMMAND OPERATORS

QCO_AT

Positions the cursor.

OP: $4C
OPL: AT
Para1: New X position (1 to 16)
Para2: New Y position (1 or 2)
Stack: Drops the two integers on the stack
Calls: DP$STAT
Errors: ER_FN_BA - Bad parameter if either parameter out of range.

Clears RTB_CRFL, the carriage return flag.

QCO_BEEP

Beeps with a frequency of 460800/(39+para2).

OP: $4D
OPL: BEEP
Para1: Integer duration in milliseconds
Para2: Integer period
Stack: Drops the two integers
Calls: BZ$TONE
Bugs: If para1 is negative BEEP returns immediately. Para2 is regarded as an unsigned word.

QCO_BREAK

Break the execution of OPL. Note that this is not equivalent to the OPL word BREAK.

OP: $26
Calls: UT$LEAV

QCO_CLS

Clears the screen. The cursor is homed to the top left.

OP: $4E
OPL: CLS
Stack: No effect
Calls: DP$CLRB

QCO_CURSOR

Set the cursor on or off.

OP: $4F
OPL: CURSOR ON, CURSOR OFF
Stack: No effect
Calls: DP$STAT

Gets byte after operator, sets or clears most significant bit of DPB_CUST.

QCO_ESCAPE

Enables or disables the ON/CLEAR key freeze and quit.

OP: $50
OPL: ESCAPE ON, ESCAPE OFF
Stack: Drops the integer on the stack

Gets byte after operator, sets or clears RTA_ESCF.

QCO_GOTO

Jump RTA_PC to a new location in the same procedure.

OP: $51
OPL: GOTO, BREAK, CONTINUE, ELSE
Stack: No effect

Adds word after the operator to RTA_PC. See QCO_BRA_FALSE.

QCO_OFF

Turns off the machine. Does not terminate language execution.

OP: $52
OPL: OFF
Stack: No effect
Calls: BT$SWOF

This is exactly the same state as when the machine is turned off at the top level. The drain on the battery is minimal.

QCO_OFF_TIM

Turns off the machine for a specified amount of seconds. Only available on the LZ model range.

OP: $D2
OPL: OFF n%
Stack: Drops the integer
Calls: BT$TOFF

Works like QCO_OFF but turns organiser on again after n% seconds. n% must be in the range 2-1600. Organiser wakes prematurely if an alarm is due.

QCO_ONERR

Set up error handling.

OP: $53
OPL: ONERR, ONERR OFF

Stack: No effect

The following word contains the offset to the address to jump to in the event of an error being detected. ONERR OFF is the same operator followed by a zero word. The ONERR address is saved in the procedure header.

QCO_PAUSE

If positive it pauses for that many 50 millisecond units, if negative it pauses for that many 50 millisecond units or until the first key press. If it is zero it waits for the next key press.

OP: $54

OPL: PAUSE

Stack: Drops the integer

Bugs: If a key is pressed it is not removed from the input buffer, so it should be read by a KEY or GET function.

Uses the 'SLP' processor instruction, so less power is used when PAUSEd compared to normal operation. It does however use more power than being switched off.

QCO_POKEB

Pokes a byte into memory.

OP: $55

OPL: POKEB

Para1: Address to write to

Para2: Byte to be written

Stack: Drops the two integers

Errors: ER_FN_BA - Bad parameter

Reports an error if para2 is not a byte. If the address is in the protected range $00 to $3F or $282 to $400 then it does nothing.

QCO_POKEW

Pokes a word into memory.

OP: $56

OPL: POKEW

Para1: Address to write to

Para2: Word to be written

Stack: Drops the two integers on the stack

Errors: ER_FN_BA - Bad parameter

If the address is in the protected range $00 to $3F or $282 to $400 then it does nothing.

QCO_RAISE

Generates an error condition.

OP: $57

OPL: RAISE

Stack: Drops the integer

Errors: ER_FN_BA - Bad parameter

If integer on the stack is not a byte it reports error. Otherwise it has exactly the same effect as if that error was generated. Errors generated by RAISE are handled in the normal way by ONERR.

Using this command and ONERR the programmer can completely take-over the handling and reporting of errors.

If the error is out of the range normally reported by the OS the message "*** ERROR ***" is reported.

RAISE 0 is special as it does not report an error.

QCO_RANDOMIZE

Set the seed of the random number generator. The sequence numbers generated by RND becomes repeatable.

OP: $58

OPL: RANDOMIZE

Stack: Drops the floating point number on the stack

Calls: FN$RAND

QCO_SPECIAL

Special operator used to vector to machine code.

OP: $25

OPL: See below

Stack: No effect

Vectors via the contents of the location RTA_1VCT to machine code. The machine code should return with the carry flag set to report an error.

If the ASCII value 1 is encountered in the OPL source code it is taken to be a SPECIAL call which returns an integer. A 2 is for a floating point return and 3 for a string. It is impossible to get these values into the source code from the editor, it must be generated by another program.

QCO_STOP

Stops executing the language.

OP: $59

OPL: STOP

Resets RTA_SP, zeroes the file buffers by calling AL_ZERO and leaves the language.

QCO_TRAP

Disables the reporting of any error arising from the execution of the following operator. Instead the error number is saved in RTB_EROR which can be read by the function ERR.

OP: $5A

OPL: TRAP

Stack: No effect

Clears RTB_EROR and sets the trap flag RTB_TRAP. The following operators can be used with TRAP:

APPEND	BACK	CLOSE
COPY	CREATE	DELETE
ERASE	EDIT	FIRST
INPUT	LAST	NEXT
OPEN	POSITION	RENAME
UPDATE	USE	

If no error occurs these operators clear RTB_TRAP.

Most of these are file-related operator. The programmer will frequently either need to report errors arising from the operators himself or handle them in a discriminating way. For example:

```
TRAP OPEN "B:XYZ",A,A$
IF ERR
  TRAP OPEN "C:XYZ",A,A$
  IF ERR
    CLS :PRINT "FILE XYZ NOT" :PRINT "FOUND"
    BEEP 100,100 :GET :STOP
  ENDIF
ENDIF
```

INPUT and EDIT are different. TRAP changes the conditions under which they exit. "EDIT A$" will not exit on the ON/CLEAR key, "TRAP EDIT A$" will exit with RTB_EROR set to ER_RT_BK. When inputting a number without the TRAP option, the routine will not exit until a valid number is input; however with TRAP any input will be accepted and the corresponding error condition placed in RTB_EROR.

See QCO_INPUT_INT, QCO_INPUT_NUM, QCO_INPUT_STR, QCO_EDIT.

FILE OPERATORS

QCO_APPEND

Adds the current record buffer to the current file as a new record.

OP: $5B

OPL: APPEND

Stack: No effect

Errors: ER_RT_FC - file not open

Calls: FL$SETP, FL$RECT, FL$RSET, FL$WRIT

Bugs: If the current length of the current record is zero, it is automatically made non-zero by adding a TAB, the field delimiter.

The contents of the file buffer are saved at the end of the current device. The first byte of the buffer is the length of the buffer.

QCO_CLOSE

Closes the current file.

If several files are open it is unpredictable which will become current.

OP: $5C

OPL: CLOSE

Stack: No effect

Errors: ER_RT_FC - file not open

Calls: FL$SETP, FL$RECT, FL$RSET, AL$ZCEL

Bugs: After closing the file it looks for another file to make current.

CLOSE has no effect on the file itself, it checks that the file is open, clears the record type in RTT_FIL, and zeroes the two cells.

QCO_COPY

Copies a file from one device to another. If the target already exists the data is appended.

OP: $5D

OPL: COPY

Stack: Drops the names of the two files

Errors: ER_FL_NX - file does not exist
ER_PK_CH - changed pack

Calls: FL$COPY

Bugs: You cannot copy to the same device.

QCO_COPYW

Copies files from one device to another using wildcards. Only available on the LZ model range.

OP: $D3

OPL: COPYW

Stack: Drops the names of the two file specifications

Errors: ER_FL_NX - file does not exist
ER_PK_CH - changed pack

Calls: FL$WCPY

Bugs: You cannot copy to the same device.

QCO_CREATE

Creates a file.

OP: $5E

OPL: CREATE

Stack: Drops the name of the file to be created

Errors: ER_FL_EX - file already exists
ER_AL_NR - out of memory

Calls: FL$CRET, AL$GROW, FL$SETP, FL$RECT, FL$RSET, FL$READ

QCO_DELETE

Deletes a file.

OP: $5F

OPL: DELETE

Stack: Drops the name of the file to be deleted.

Errors: ER_FL_NX - file does not exist
ER_RT_FO - file open

Calls: FL$DELN

Checks that the file is not open. Deletes all records, starting with the first, and finally the file name record of the file.

QCO_DELETEW

Deletes files using wildcards. Only available on the LZ model range.

OP: $D4

OPL: DELETEW

Stack: Drops the specification of the files to be deleted.

Errors: ER_FL_NX - file does not exist
ER_RT_FO - file open

Calls: FL$DELN

See QCO_DELETE.

QCO_ERASE

Erases the current record of the current file.

OP: $60

OPL: ERASE

Stack: No effect

Errors: ER_RT_FC - file not open
ER_FL_EF - end of file

Calls: FL$ERAS, FL$SETP, FL$RECT, FL$RSET, FL$READ

Bugs: The current record becomes the record following the erased record. If, after the erase, FL$READ returns an 'END OF FILE', the length of the current record is set to zero and the current record number set to the number of records (as found by FL$SIZE) plus one.
'END OF FILE' error will be generated if already at the end of the file. This includes the case of a file with no records.

QCO_FIRST

Goes to the first record of the current file.

OP: $61

OPL: FIRST

Stack: No effect

Errors: ER_RT_FC - file not open

Calls: FL$SETP, FL$RECT, FL$RSET, FL$READ

Bugs: No error reported if there are no records.

QCO_LAST

Goes to the last record of the current file.

OP: $62
OPL: LAST
Stack: No effect
Errors: ER_RT_FC - file not open
Calls: FL$SIZE, FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs: No error reported if there are no records.

QCO_NEXT

Goes to the next record.

OP: $63
OPL: NEXT
Stack: No effect
Errors: ER_RT_FC - file not open
Calls: FL$NEXT, FL$READ
Bugs: No error reported if at the end of file. If FL$READ returns an "END OF FILE" error, the length of the current record is set to zero and the current record number set to the number of records (as found by FL$SIZE) plus one.

QCO_BACK

Steps back one record.

OP: $64
OPL: BACK
Stack: No effect
Errors: ER_RT_FC - file not open
Calls: FL$BACK
Bugs: No error reported if already on the first record.

QCO_OPEN

Open a file.

OP: $65
OPL: OPEN
Stack: Drop the name of the file.
Errors: ER_RT_FO - file open
Calls: FL$OPEN, FL$SETP, FL$RECT, FL$RSET, FL$READ

OPEN has exactly the same form as CREATE.

QCO_POSITION

Position at that record.

OP: $66
OPL: POSITION
Stack: Drops the integer
Errors: ER_RT_FC - file not open
Calls: FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs: If the FL$READ returns an 'END OF FILE', the length of the current record is set to zero and the current record number set to the number of records (as found by FL$SIZE) plus one.

QCO_RENAME

Renames a file.

OP: $67
OPL: RENAME
Stack: Drops the two file names
Errors: ER_RT_FO - file open
ER_FL_NX - file exists
ER_FL_NX - file does not exist
Calls: FL$RENM

Erases the file name record and writes a new one.

QCO_UDG

Defines an UDG. Only available on the LZ model range.

OP: $D6
OPL: UDG
Stack: Drops 9 integers
Calls: DP$UDG

QCO_UPDATE

Updates a record.

OP: $68
OPL: UPDATE
Stack: No effect
Errors: ER_RT_FC - file not open
Calls: FL$ERAS, FL$WRIT, FL$SETP, FL$RECT, FL$RSET, FL$READ
Bugs: If the APPEND fails, with 'PAK FULL' for example, the original record is already erased.

It deletes the current record in the current file and then APPENDs the contents of the buffer.

QCO_USE

Changes the current file.

OP: $69
OPL: USE
Stack: No effect
Errors: ER_TR_BL - bad logical name (logical name not in use)

Takes the byte following the operator and after checking it makes it the new current logical file. See logical file names.

OTHER OPERATORS

QCO_KSTAT

Set the shift state of the keyboard.

OP: $6A
OPL: KSTAT
Stack: Drops integer
Errors: ER_FN_BA - function argument error
Calls: KB$STAT

Use KSTAT to change the upper/lower alpha/numeric case:

1 alpha, upper case (default setting)
2 alpha, lower case
3 numeric, upper case
4 numeric, lower case

QCO_EDIT

Edits a string.

OP: $6B

OPL: EDIT

Stack: Drop the left side reference to string

Errors: ER_RT_BK - ON/CLEAR key pressed
ER_RT_FC - file not open
ER_RT_NF - field not found
ER_RT_RB - record too big

Calls: ED$EDIT

If the string to be edited is a field then the maximum length of the string is 252. Otherwise the maximum length allowed is the length of the string as defined in the LOCAL or GLOBAL statement. The string to be edited is copied into RTT_BUF. Once the string is edited it is assigned to the source. If the EDIT is preceded by TRAP then the edit will exit on the ON/CLEAR key with the error condition ER_RT_BK. The string remains unchanged. Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

QCO_INPUT_INT

Input an integer.

OP: $6C

OPL: INPUT

Stack: Drops the left side integer reference

Errors: ER_RT_BK - ON/CLEAR key pressed
ER_MT_IS - conversion to number failed
ER_RT_IO - integer overflow
ER_RT_FC - file not open
ER_RT_NF - field not found
ER_RT_RB - record too big

Calls: ED$EDIT

If the INPUT is preceded by TRAP then the input will exit on the ON/CLEAR key with the error condition ER_RT_BK. It will also exit if an invalid integer is input, e.g. 99999 or $1. If there is no TRAP then the INPUT will not exit on the ON/CLEAR key and invalid integers generate a '?' on the next line and the INPUT is repeated. Up to 6 characters, including leading spaces, are allowed. Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

QCO_INPUT_NUM

Inputs a floating point number.

OP: $6D

OPL: INPUT

Stack: Drops left side reference to floating point number

Errors: ER_RT_BK - ON/CLEAR key pressed
ER_MT_IS - conversion to number failed
ER_RT_IO - integer overflow
ER_RT_FC - file not open
ER_RT_NF - field not found
ER_RT_RB - record too big

Calls: ED$EDIT

If the INPUT is preceded by TRAP then the input will exit on the ON/CLEAR key with the error condition ER_RT_BK. It will also exit if an invalid floating point number is input, e.g. 999999999999999 or $1. If there is no TRAP then the INPUT will not exit on the ON/CLEAR key and invalid integers generate a '?' on the next line and the INPUT is repeated. Up to 15 characters, including leading spaces, are allowed. Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

QCO_INPUT_STR

Inputs a string.

OP: $6E

OPL: INPUT

Stack: Drops left side reference to string

Errors: ER_RT_FC - file not open
ER_RT_NF - field not found
ER_RT_RB - record too big

Calls: ED$EDIT

QCO_INPUT_STR is exactly equivalent to QCO_EDIT with an initial null string.

QCO_PRINT_INT

Prints an integer to the screen.

OP: $6F

OPL: PRINT

Stack: Drops the integer

Calls: UT$DISP

Bugs: If the number $FFFF is assigned to an integer and then it is printed it will be represented as -1.

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

QCO_PRINT_NUM

Prints a floating point number to the screen.

OP: $70

OPL: PRINT

Stack: Drops the floating point number

Calls: UT$DISP

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared. The format in which a number is displayed is integer, decimal or scientific in that order of precedence.

QCO_PRINT_STR

Print a string to the screen.

OP: $71

OPL: PRINT

Stack: Drops the string

Calls: UT$DISP

Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

QCO_PRINT_SP

Prints a space to the screen.

OP: $72

OPL: PRINT

Stack: No effect

Calls: UT$DISP

This operator is generated by use of the ',' separator in a PRINT statement. Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared.

QCO_PRINT_CR

Print a carriage return to the screen.

OP: $73

OPL: PRINT

Stack: No effect

Calls: UT$DISP

If a PRINT, INPUT or EDIT statement is not followed by a ';' or ',' then this operator is automatically inserted. It is not acted on immediately; it sets the flag RTB_CRFL. Before execution of this operator RTB_CRFL is tested and, if set, a carriage return is sent to the screen and the flag cleared. Note that if a carriage return results in scrolling the screen there is an automatic delay; the length of this delay is defined by DPW_DELY which is in 50 millisecond units, the default being 10.

QCO_LPRINT_INT

Sends an integer to the RS232.

OP: $74

OPL: LPRINT

Errors: ER_DV_NP - device missing
ER_DV_CS - device load error

Exactly as PRINT_INT, except the CR flag is not tested.

QCO_LPRINT_NUM

Send a floating point number to the RS232.

OP: $75

OPL: LPRINT

Errors: ER_DV_NP - device missing
ER_DV_CS - device load error

Exactly as PRINT_NUM, except the CR flag is not tested.

QCO_LPRINT_STR

Send a string to the RS232.

OP: $76

OPL: LPRINT

Errors: ER_DV_NP - device missing
ER_DV_CS - device load error

Exactly as PRINT_STR, except the CR flag is not tested.

QCO_LPRINT_SP

Send a space character to the RS232.

OP: $77

OPL: LPRINT

Errors: ER_DV_NP - device missing
ER_DV_CS - device load error

Exactly as PRINT_SP, except the CR flag is not tested.

QCO_LPRINT_CR

Send a carriage return to the RS232.

OP: $78

OPL: LPRINT

Errors: ER_DV_NP - device missing
ER_DV_CS - device load error

As PRINT_CR except it is acted on immediately.

QCO_RETURN

Return from a procedure.

OP: $79

OPL: RETURN

Stack: Unwinds the procedure

This operator follows the operator which stacks the return value. All procedures return a value. If no explicit value is returned then it will return integer zero for integer procedures, floating point zero for floating point procedures or a null string for string procedures.

QCO_RETURN_NOUGHT

For an integer procedure this is the default return.

OP: $7A

OPL: RETURN

Stack: Stack the integer zero and then unwind the procedure

Stacks default return value, then exactly the same as QCO_RETURN.

QCO_RETURN_ZERO

For an floating point procedure this is the default return.

OP: $7B

OPL: RETURN

Stack: Stack a floating point zero and then unwind the procedure

Stacks default return value, then exactly the same as QCO_RETURN.

QCO_RETURN_NULL

For a string procedure this is the default return.

OP: $7C

OPL: RETURN

Stack: Adds a null string and the unwinds the procedure

Stacks default return value, on the stack, then exactly the same as QCO_RETURN.

QCO_PROC

Call a procedure.

OP: $7D

OPL: procnam:

Stack: Initialises procedure

Errors: ER_RT_PN - procedure not found
ER_RT_NP - wrong number of parameters
ER_RT_UE - undefined external
ER_EX_TV - parameter type mis-match
ER_AL_NR - out of memory
ER_GN_BL - test explicitly for low battery error

Calls: PK$RBYT, PK$RWRD, PK$READ, DV$LKUP, DV$VECT

First checks to see if a language extension of that name has been booted into memory. If not it searches the 4 devices for an OPL procedure of the right name. It starts with the default device. So if the procedure called was on C: then it searches in the order C:, D:, A: and B:. If a language extension has been found (for example LINPUT) it calls the relevant vector and the device is then responsible for checking the parameters and handling the stack. See language extensions. If it is an OPL procedure the header information is read in and the memory required checked. The external references are then checked and the fixups on the strings and arrays performed. The Q code is then read in, and RTA_PC and RTA_SP are set to their new values.

QCO_BRA_FALSE

Branches if the integer on the stack is false.

OP: $7E

OPL: UNTIL, WHILE, IF, ELSEIF

Stack: Drop the offset

Adds the integer following the operator to RTA_PC if the value on the stack is zero.

QCO_ASS_INT

Assign an integer to a variable.

OP: $7F

OPL: =

Stack: Drops the integer and the integer reference

Errors: ER_RT_RB - field too big
ER_RT_FC - file not open
ER_RT_NF - field not found
ER_RT_RB - record too big

At the start of the operand the stack looks like:

```
High memory             Address of integer variable
                        0 (field flag)
Low memory              Integer
                or:
High memory             Field name
                        Logical file name (0,1,2 or 4)
                        1 (field flag)
Low memory              Integer
```

If the assign is to a field, it checks that the file is open, checks the field name and saves the value. If not a field it simply saves the integer to the address.

QCO_ASS_NUM

Assigns a floating point number.

OP: $80

OPL: =

Stack: Drops the floating point number and the floating point reference

Errors: ER_RT_RB - field too big
ER_RT_FC - file not open
ER_RT_NF - field not found
ER_RT_RB - record too big

Exactly the same as QCO_ASS_INT except it handles floating point numbers.

QCO_ASS_STR

Assigns a string.

OP: $81

OPL: =

Stack: Drops the string and the string reference

Errors: ER_RT_RT - field too big
ER_LX_ST - string too long

Exactly the same as QCO_ASS_INT except it handles strings.

QCO_DROP_BYTE

Drops a byte off stack.

OP: $82

OPL: -

Stack: Drops byte

QCO_DROP_WORD

Drops a word off the stack.

OP: $83

OPL: -

Stack: Drops word

Used internally to drop unwanted results off the stack, for example a statement "GET" which translates into RTF_GET,QCO_DROP_WORD.

QCO_DROP_NUM

Drops a floating point number off the stack.

OP: $84

OPL: -

Stack: Drops a floating point number

Used internally to OPL when, for example, a floating point procedure returns a value that is not required.

QCO_DROP_STR

Drops a string off the stack.

OP: $85

OPL: -

Stack: Drops a string off the stack

Used internally to OPL when, for example, a string procedure returns a string that is not required.

QCO_INT_TO_NUM

Converts an integer into a floating point number.

OP: $86

OPL: -

Stack: Drops an integer, stacks a float

Calls: MT$BTOF

Bugs: Integers are always taken as signed. To make unsigned:

```
A=I% :IF I%<0 :A=A+65536 :ENDIF
```

Used for automatic type conversion.

QCO_NUM_TO_INT

Converts a floating point number to integer.

OP: $87

OPL: -

Stack: Drops float, stacks integer

Errors: ER_RT_IO - integer overflow

Calls: IM$DINT, IM$FLOI

Bugs: Always rounds down, 3.9 becomes 3 and -3.9 becomes -4.

Used for automatic type conversion.

QCO_END_FIELDS

Indicates where the field names end.

OP: $88

OPL: OPEN, CREATE

Stack: No effect

Only used internally at the end of an OPEN or CREATE command.

QCO_RUN_ASSEM

Runs machine code immediately after operator.

OP: $89

OPL: -

Stack: No effect

Runs the code immediately after the operator as machine code. On return if there are no errors carry must be clear and the B register must be the number of bytes for RTA_PC to jump. If there is an error carry must be set and the B register should contain the number of the error to be reported. This cannot be generated from the editor.

INTEGER FUNCTIONS

These functions return integer values.

RTF_ADDR

Returns the address of a numeric variable.

OP: $8A

OPL: ADDR

Stack: Drops the 'left side' reference, stacks the address.

Bugs: Cannot deal with elements of arrays, though they may be easily calculated.

In the case of arrays ADDR returns the address of the first element which is immediately after the word giving the size of the array. So "PRINT PEEKW(ADDR(A%))" is exactly the same as "PRINT A%" and "PRINT PEEKW(ADDR(A%())) is the same as "PRINT A%(1)".

RTF_ASC

Returns the ASCII value of the first character of the string.

OP: $8B

OPL: ASC

Stack: Drops the string, stacks an integer

Bugs: If the string is zero length it returns zero.

RTF_CLOCK

Turns UDG clock on or off. Only available on the model LZ range.

OP: $D6

OPL: CLOCK

Stack: Drops an integer, stacks an integer

Calls: DP$CSET

RTF_DAY

Returns the current day of the month - in the range 1 to 31.

OP: $8C

OPL: DAY

Stack: Stack an integer

RTF_DAYS

Returns number of days since 1/1/1900 for a specified date. Only available on the model LZ range.

OP: $DD

OPL: DAYS

Stack: Drops 3 integers, stacks an integer

Calls: TM$NDYS

RTF_DISP

Displays a string, a record or the last string displayed, using cursor keys for viewing and waiting for any other key to exit.

OP: $8D

OPL: DISP

Para1: Integer:
1 - displays para2
0 - redisplays the last DISPed string (ignores para2)
-1 - displays the current record (ignores para2)

Para2: String to be displayed

Stack: Drops the two parameters, stacks the exit key as an integer.

Calls: UT$DISP

Bugs: In the case para1 is zero it displays the contents of RTT_BUF.
RTT_BUF is used by a number of other operand/operators, for instance by string adds.

The display used is the same as that used by FIND in the top level. Each field, delimited by a TAB character, is on a different line. There is no limit to the number of fields.

RTF_DOW

Returns day of the week of the given date (1...Monday). Only available on LZ model range.

OP: $D5

OPL: DOW

Stack: Drops 3 integers, stacks an integer

RTF_ERR

Returns the current error value.

OP: $8E

OPL: ERR

Stack: Stack the error number as an integer

When the language starts running the value of RTB_EROR is zero. If an error is encountered and handled by a TRAP or ONERR the value remains until the next error or a TRAP command.

RTF_FIND

Finds a string in the current file.

OP: $8F

OPL: FIND

Stack: Drops the search string, stacks the record number.

Calls: FL$FIND

Bugs: FIND does not do an automatic NEXT, the correct loop structure is:

```
DO
  IF FIND "ABC"
    statement(s)
  ENDIF
  NEXT
UNTIL EOF
```

If no record is found zero is returned and the current record remains the same as before the FIND.

RTF_FINDW

Finds a string in the current file using wildcards. Only available on LZ model range.

OP: $D8

OPL: FINDW

Stack: Drops the search string, stacks the record number.

Calls: FL$WFND

Bugs: See RTF_FIND.

RTF_FREE

Returns the amount of free memory.

OP: $90

OPL: FREE

Stack: Stack the resulting integer.

Calculates the amount of free memory by subtracting ALA_FREE from RTA_SP and then subtracting $100.

RTF_GET

Get a single character.

OP: $91

OPL: GET

Stack: Stack the character as an integer.

Calls: KB$GETK

Bugs: The ON/CLEAR key returns 1. It can be difficult to break out of a tight loop with a GET using the ON/CLEAR, Q keys. With perseverance it is normally possible.

If there is a key in the buffer it gets that key first. If no key is received the Organiser will turn itself off after the timeout.

RTF_HOUR

Returns the current hour of the day - in the range 0 to 23.

OP: $92

OPL: HOUR

Stack: Stack the number as an integer.

RTF_IABS

Does an ABS on an integer.

OP: $93

OPL: IABS

Stack: Leaves the integer on the stack.

Converts a negative integer to a positive integer. If ABS is used in place of IABS the result would be the same but the function would require two unnecessary type conversions. IABS is significantly faster than ABS.

RTF_INT

Converts a floating point number to an integer.

OP: $94

OPL: INT

Stack: Drops float, stacks integer

Errors: ER_RT_IO - integer overflow

Calls: IM$DINT, IM$FLOI

Bugs: Always rounds down, INT(3.9) is 3 and INT(-3.9) is -4.

Identical to QCO_NUM_TO_INT.

RTF_KEY

Returns any key in the input buffer. Zero if no key is waiting.

OP: $95

OPL: KEY

Stack: Stack the integer

Bugs: Except after an "ESCAPE OFF" statement, KEY cannot pick up the ON/CLEAR key.

RTF_LEN

Returns the length of the string.

OP: $96

OPL: LEN

Stack: Drops string, stacks the length as an integer

RTF_LOC

Locates one string in another, returns zero if not found.

OP: $97

OPL: LOC

Para1: String to be searched

Para2: String to locate

Stack: Drops the two strings, stacks the resulting position as an integer

---

RTF_MENU

Gives a menu of options.

OP: $98

OPL: MENU

Stack: Drops the string, stacks the exit item as an integer

Calls: MN_DISP

Errors: ER_RT_MU - menu error
ER_FN_BA - bad argument

Bugs: In the input string the menu items are delimited by commas. Before MN_DISP is called the string is converted to individual strings each terminated by a null word. It is possible to have too many items.
Don't have spaces or tabs as part of menu items, they can have unpredictable effects.

The normal input is a string with each menu item delimited by a comma. An item is selected either by a unique first letter or by positioning on that item and pressing the EXE key. If the menu exits by the ON/CLEAR key it returns zero.

RTF_MENUN

Gives a one-line menu of options. Only available on LZ model range.

OP: $D9

OPL: MENUN

Stack: Drops the integer and the string, stacks the exit item as an integer

Calls: MN_1DSP

Errors: ER_RT_MU - menu error
ER_FN_BA - bad argument

Bugs: See RTF_MENU.

RTF_MINUTE

Returns the current minute of the hour - in the range 0 to 59.

OP: $99

OPL: MINUTE

Stack: Stack the number as an integer.

RTF_MONTH

Returns the current month of the year - in the range 0 to 11.

OP: $9A

OPL: MONTH

Stack: Stack the number as an integer.

RTF_PEEKB

Peeks a byte at the given address.

OP: $9B

OPL: PEEKB

Stack: Drops the address, stacks the result as an integer

If the address is in the ranges $00-$3F and $282-$400 then it returns zero. These ranges are the processor registers and the custom chip's control addresses. The informed user may access these addresses via machine code.

RTF_PEEKW

Peeks a word at the given address.

OP: $9C

OPL: PEEKW

Stack: Drops the address, stacks the result as an integer

See the comments after RTF_PEEKB.

RTF_RECSIZE

Returns the size of the current record.

OP: $9D

OPL: RECSIZE

Stack: Stack the size as an integer.

Bugs: The maximum size of a record is 254, this includes the field separators.

See Records and Fields for more details.

RTF_SECOND

Returns the current second of the minute - in the range 0 to 59.

OP: $9E

OPL: SECOND

Stack: Stack the number as an integer.

RTF_WEEK

Returns the week number of a specified date. Only available on LZ model range.

OP: $DA

OPL: WEEK

Stack: Drops 3 integers, stacks the week number as an integer.

Calls: TM$WEEK

RTF_IUSR

Calls machine code.

OP: $9F

OPL: USR

Para1: Address of the machine code

Para2: The value to be passed in the D register

Stack: Drops the parameters, stacks the X register on return

RTF_SADDR

Returns the address of a string.

OP: $C9

OPL: ADDR

Stack: Stack the result

Returns the address of the length byte, the byte after the the maximum length. In the case of an array it returns the address of the length byte of the first element of the array. So "ADDR(A$())-2" is the address of the size the array (a word) and "ADDR(A$())-3" is the address of the maximum string length (a byte).

RTF_VIEW

View a string, or the last string viewed.

OP: $A0

OPL: VIEW

Para1: Line on which to view (1 or 2)

Para2: String to be viewed

Stack: Drops the parameters, stacks the exit character as an integer

If the string is null it re-displays the last string VIEWed (which is held in RTT_BUF).

RTF_YEAR

Returns the current year - in the range 0 to 99.

OP: $A1

OPL: YEAR

Stack: Stack the number as an integer

RTF_COUNT

Returns the number of records in the current file.

OP: $A2

OPL: COUNT

Stack: Stack the result as an integer

Calls: FL$SIZE

RTF_EOF

Returns TRUE if the position in the file is at the end of file. If the current record is the last record of the file, EOF returns FALSE.

OP: $A3
OPL: EOF
Stack: Stack result as an integer
Errors: ER_RT_FC - file not open
Bugs: If there are no records this returns true.

Returns TRUE if the current record buffer is zero. When OPL appends a record with zero length it adds a TAB ($09) character so that it never actually saves a null string.

RTF_EXIST

Returns TRUE is the file exists.

OP: $A4
OPL: EXIST
Stack: Drops string, stacks result
Calls: FL$OPEN

RTF_POS

Returns the current record number in the current file.

OP: $A5
OPL: POS
Stack: Stack the result
Calls: FL$SETP, FL$RECT, FL$RSET
Errors: ER_RT_FC - file not open
Bugs: If no records still return 1.

FLOATING POINT FUNCTIONS

These functions return a floating point value.

RTF_ABS

Does an ABS on a floating point number.

OP: $A6
OPL: ABS
Stack: Leaves the floating point number on the stack.
Calls: FN$ABS

RTF_ACOS

Returns the reverse cosine of the input in radians. Only available on LZ model range.

OP: $DB
OPL: ACOS
Stack: Drops the input floating point number, stacks the result
Calls: FN$ACOS

RTF_ASIN

Returns the reverse sinus of the input in radians. Only available on LZ model range.

OP: $DC
OPL: ASIN
Stack: Drops the input floating point number, stacks the result
Calls: FN$ASIN

RTF_ATAN

Returns the arctangent of the input in radians.

OP: $A7
OPL: ATAN
Stack: Drops the input floating point number, stacks the result
Calls: FN$ATAN
Bugs: Returns values in the range plus or minus pi/2

RTF_COS

Returns the cosine of the input, the input being in radians.

OP: $A8
OPL: COS
Stack: Drops the input floating point number, stacks the result
Calls: FN$COS
Errors: ER_FN_BA - bad argument if the absolute value is greater than 3141590.

RTF_DEG

Converts the input from radians to degrees.

OP: $A9
OPL: DEG
Stack: Drops the input floating point number, stacks the result
Calls: FN$DEG
Bugs: All this does is multiply the input by 57.29...

RTF_EXP

Returns the value of e raise to the specified power.

OP: $AA
OPL: EXP
Stack: Drops the input floating point number, stacks the result
Calls: FN$EXP
Errors: ER_FN_BA - bad argument if the absolute value is greater than 229.

RTF_FLT

Converts an integer to floating point format.

OP: $AB
OPL: FLT
Stack: Drops the input integer, stacks the result
Calls: MT$BTOF
Bugs: Integers are always taken as signed. To make unsigned:

```
A=I% :IF I%<0 :A=A+65536 :ENDIF
```

Exactly the same effect as QCO_INT_TO_NUM.

RTF_INTF

Rounds a floating point number down to a whole number.

OP: $AC
OPL: INTF
Stack: Drops the input floating point number, stacks the result
Calls: IM$DINT, IM$FLOI

Essential to use INTF rather than INT if the number is out of the integer range.

RTF_LN

Returns the natural logarithm of the input.

OP: $AD
OPL: LN
Stack: Drops the input floating point number, stacks the result
Errors: ER_FN_BA - bad argument
Calls: FN$LN
Bugs: The input must be greater than 0.

RTF_LOG

Returns the base 10 logarithm of the input.

OP: $AE
OPL: LOG
Stack: Drops the input floating point number, stacks the result
Errors: ER_FN_BA - bad argument
Calls: FN$LOG
Bugs: The input must be greater than 0.

RTF_PI

Returns the number pi = 3.14159265359.

OP: $AF
OPL: PI
Stack: Stack the result
Calls: FN$PI

RTF_RAD

Converts the input number to radians. The inverse of DEG.

OP: $B0
OPL: RAD
Stack: Drops the input floating point number, stacks the result
Calls: FN$RAD
Bugs: All this does is divide the input by 57.29...

RTF_RND

Returns a pseudo-random number in the range 0(inclusive) to 1(exclusive).

OP: $B1
OPL: RND
Stack: Stack the result
Calls: FN$RND

RTF_SIN

Returns the sine of the input, the input being in radians.

OP: $B2
OPL: SIN
Stack: Drops the input floating point number, stacks the result
Calls: FN$SIN
Errors: ER_FN_BA - bad argument if the absolute value is greater than 3141590.

RTF_SQR

Returns the square root of the input.

OP: $B3
OPL: SQR
Stack: Drops the input floating point number, stacks the result
Calls: FN$SQRT
Errors: ER_FN_BA - bad argument if negative

RTF_TAN

Returns the tangent of the input, the input being in radians.

OP: $B4
OPL: TAN
Stack: Drops the input floating point number, stacks the result
Calls: FN$TAN
Bugs: At the the discontinuities in TAN, pi/2, 3*pi/2, etc, the values returned are either greater than 1E10 or less than -1E10.

RTF_VAL

Returns the input string as a number.

OP: $B5
OPL: VAL
Stack: Drops the input string, stacks the result
Errors: ER_MT_FL - conversion to number failed
Calls: MT$BTOF
Bugs: This routine insists that the whole string is used in the conversion, so VAL("12.34 ") generates an error. The null string also gives an error.

RTF_SPACE

Returns the amount of space on the current device.

OP: $B6
OPL: SPACE
Stack: Stack the result as floating point number
Calls: FL$SIZE
Errors: ER_RT_FC - file not open

Bugs: This may be longer than a word!

The following functions are only available on the LZ model range. They accept either

- an array and an integer (number of items to inspect)
- or a list of floats

as parameters.

On execution first the last two bytes are dropped from the stack. If the last byte is $01, the second last byte is the number of array items to be inspected and the array reference is dropped. If it is $00, the second last byte denotes the number of list items. These are dropped as well.

RTF_MAX

Returns the maximum of the specified array items or list.

OP: $DE

OPL: MAX

Stack: Drops the parameters (see above), stacks the result as float

Calls: FN$MAX

RTF_MEAN

Returns the mean of the specified array items or list.

OP: $DF

OPL: MEAN

Stack: Drops the parameters (see above), stacks the result as float

Calls: FN$MEAN

RTF_MIN

Returns the minimum of the specified array items or list.

OP: $E0

OPL: MIN

Stack: Drops the parameters (see above), stacks the result as float

Calls: FN$MIN

RTF_STD

Returns the standard deviation of the specified array items or list.

OP: $E1

OPL: STD

Stack: Drops the parameters (see above), stacks the result as float

Calls: FN$STD

RTF_SUM

Returns the sum of the specified array items or list.

OP: $E2

OPL: SUM

Stack: Drops the parameters (see above), stacks the result as float

Calls: FN$SUM

RTF_VAR

Returns the variance of the specified array items or list.

OP: $E3

OPL: VAR

Stack: Drops the parameters (see above), stacks the result as float

Calls: FN$VAR

STRING FUNCTIONS

RTF_DAYNAME

Converts an integer (0-7) to the day of the week. 1 returns "Mon" etc. Only available on the LZ model range.

OP: $E4

OPL: DAYNAME$

Stack: Drops the input integer, stacks the resulting string

Calls: TM$DNAM

RTF_DIR

Returns the name of the first/next file on a device.

OP: $B7

OPL: DIR$

Stack: Drops the input string, stacks the resulting string

Calls: FL$CATL

Errors: ER_FN_BA - bad argument

If the string is non-null it checks that it is of the form "A:" or "A". It splits out the device name and returns the first file name preceded by the device name. If the string is null it returns the next file name, on the device already specified. When there are no more file it returns a null string.

RTF_DIRW

Returns the name of the first/next file (of any type) on a device using wildcards. Only available on the LZ model range.

OP: $E5

OPL: DIRW$

Stack: Drops the input string, stacks the resulting string

Calls: FL$WCAT

Errors: ER_FN_BA - bad argument

See RTF_DIR.

RTF_CHR

Converts the integer input to a one character string.

OP: $B8

OPL: CHR$

Stack: Drops the input integer, stacks the resulting string

Errors: ER_FN_BA - bad argument if out of the range 0-255

RTF_DATIM

Returns the date-time string in the form: "TUE 04 NOV 1986 10:44:29"

OP: $B9

OPL: DATIM$

Stack: Stacks the resulting string

RTF_SERR

Returns the error string associated with the integer error number.

OP: $BA

OPL: ERR$

Stack: Drops the input integer, stacks the resulting string

Errors: ER_FN_BA - bad argument (if not a byte)

Calls: ER$LKUP

Bugs: Returns "*** ERROR ***" if less than the lowest recognised error number.

RTF_FIX

Returns the floating point number as a string with a fixed number of decimal places.

OP: $BB

OPL: FIX$

Para1: The floating point number

Para2: The require number of decimal places

Para3: The field size

Stack: Drops input parameters, stacks the resulting string

Calls: MT$FBDC

Bugs: If the number does not fit, '*'s are inserted

RTF_GEN

Returns the floating point number as a string. This is the same format as used by QCO_PRINT_NUM.

OP: $BC

OPL: GEN$

Stack: Drops the floating point number

Calls: MT$FBGN

Bugs: If the number does not fit, '*'s are inserted

The format in which the number is displayed is integer, decimal or scientific in that order of precedence.

RTF_SGET

Get a character and return it as a one character string.

OP: $BD

OPL: GET$

Stack: Stack the resulting string

Calls: KB$GETK

Bugs: The ON/CLEAR key returns a valid string. It can be difficult to break out of a tight loop with a GET$ using ON/CLEAR, Q keys. With perseverance it is normally possible.

RTF_HEX

Converts the integer into a hexadecimal string.

OP: $BE

OPL: HEX$

Stack: Drops input integer, stacks resulting string

Calls: UT$XTOB

Bugs: Input must be in the integer range.

RTF_SKEY

Returns any keys in the input buffer as a string. Returns the null string if no key is waiting.

OP: $BF

OPL: KEY$

Stack: Stack the string

Calls: KB$TEST, KB$GETK

Bugs: Except after an "ESCAPE OFF" statement, KEY cannot pick up the ON/CLEAR key. ON/CLEAR key normally suspends OPL execution.

RTF_LEFT

Returns the first n characters of the string.

OP: $C0

OPL: LEFT$

Para1: The string

Para2: Number of characters to keep

Stack: Drops the input parameters, stacks the resulting string

Bugs: If the string is shorter than the number of characters the entire string is returned.

RTF_LOWER

Converts the string to lower case.

OP: $C1

OPL: LOWER$

Stack: Drops the input string, stacks the result

RTF_MID

Returns the middle of a string.

OP: $C2

OPL: MID$

Para1: The string

Para2: The start character

Para3: The number of characters to be kept

Stack: Drops the input parameters, stacks the resulting string

Bugs: If there are insufficient characters the rest of the string is returned.

You can get all the characters after the n'th by the statement: MID$(a$,n,255)

RTF_MONTHNAME

Converts an integer (1-12) to the name of the month. 1 returns "Jan" etc. Only available on the LZ model range.

OP: $E6

OPL: MONTH$

Stack: Drops the input integer, stacks the resulting string

Calls: TM$MNAM

RTF_NUM

Converts a number to an integer string.

OP: $C3

OPL: NUM$
Para1: The floating point number
Para2: The maximum size of the string
Stack: Drops the input parameters, stacks the resulting string.
Calls: MT$FBIN
Bugs: If the number does not fit, '*'s are inserted The number does not have to be in usual integer range.

RTF_RIGHT

Returns the last n characters of a string.

OP: $C4
OPL: RIGHT$
Para1: The string
Para2: The number of characters wanted
Stack: Drops the input parameters, stacks the resulting string
Bugs: If the string is shorter than the number of characters the entire string is returned.

RTF_REPT

Repeats the string n times.

OP: $C5
OPL: REPT$
Para1: The string
Para2: The repeat count
Stack: Drops the integer and input string, stacks the result
Bugs: If the repeat count is zero no error is given.
Errors: ER_MT_FN - function argument error
ER_LX_ST - string too long

RTF_SCI

Returns the floating point number as a string in scientific form.

OP: $C6
OPL: SCI$
Para1: The floating point number
Para2: Number of decimal places required
Para3: Field width
Stack: Drops the floating point number, stacks the result
Calls: MT$FBEX
Bugs: If the number does not fit, '*'s are inserted

RTF_UPPER

Converts the string to upper case.

OP: $C7
OPL: UPPER$
Stack: Drops the input string, stacks the result

RTF_SUSR

Calls machine code.

OP: $C8

OPL: USR$
Para1: Address of the machine code
Para2: The value to be passed in the D register
Stack: Drops the parameters, stacks the string pointed at by the X register

---

INDEX OF OPERANDS

00	QI_INT_SIM_FP	0D	QI_LS_INT_SIM_FP	1A	QI_INT_FLD
01	QI_NUM_SIM_FP	0E	QI_LS_NUM_SIM_FP	1B	QI_NUM_FLD
02	QI_STR_SIM_FP	0F	QI_LS_STR_SIM_FP	1C	QI_STR_FLD
03	QI_INT_ARR_FP	10	QI_LS_INT_ARR_FP	1D	QI_LS_INT_FLD
04	QI_NUM_ARR_FP	11	QI_LS_NUM_ARR_FP	1E	QI_LS_NUM_FLD
05	QI_STR_ARR_FP	12	QI_LS_STR_ARR_FP	1F	QI_LS_STR_FLD
06	QI_NUM_SIM_ABS	13	QI_LS_NUM_SIM_ABS	20	QI_STK_LIT_BYTE
07	QI_INT_SIM_IND	14	QI_LS_INT_SIM_IND	21	QI_STK_LIT_WORD
08	QI_NUM_SIM_IND	15	QI_LS_NUM_SIM_IND	22	QI_INT_CON
09	QI_STR_SIM_IND	16	QI_LS_STR_SIM_IND	23	QI_NUM_CON
0A	QI_INT_SIM_IND	17	QI_LS_INT_SIM_IND	24	QI_STR_CON
0B	QI_NUM_SIM_IND	18	QI_LS_NUM_SIM_IND		
0C	QI_STR_SIM_IND	19	QI_LS_STR_SIM_IND		

INDEX OF OPERATORS

Operators that are only available on the LZ model range are marked by an asterisk.

25	QCO_SPECIAL	47	QCO_GT_STR	69	QCO_USE	CC*	QCO_LT_PERC
26	QCO_BREAK	48	QCO_GTE_STR	6A	QCO_KSTAT	CD*	QCO_GT_PERC
27	QCO_LT_INT	49	QCO_NE_STR	6B	QCO_EDIT	CE*	QCO_ADD_PERC
28	QCO_LTE_INT	4A	QCO_EQ_STR	6C	QCO_INPUT_INT	D0*	QCO_SUB_PERC
29	QCO_GT_INT	4B	QCO_ADD_STR	6D	QCO_INPUT_NUM	D1*	QCO_MUL_PERC
2A	QCO_GTE_INT	4C	QCO_AT	6E	QCO_INPUT_STR	D2*	QCO_DIV_PERC
2B	QCO_NE_INT	4D	QCO_BEEP	6F	QCO_PRINT_INT		
2C	QCO_EQ_INT	4E	QCO_CLS	70	QCO_PRINT_NUM		
2D	QCO_ADD_INT	4F	QCO_CURSOR	71	QCO_PRINT_STR		
2E	QCO_SUB_INT	50	QCO_ESCAPE	72	QCO_PRINT_SP		
2F	QCO_MUL_INT	51	QCO_GOTO	73	QCO_PRINT_CR		
30	QCO_DIV_INT	52	QCO_OFF	74	QCO_LPRINT_INT		
31	QCO_POW_INT	53	QCO_ONERR	75	QCO_LPRINT_NUM		
32	QCO_UMIN_INT	54	QCO_PAUSE	76	QCO_LPRINT_STR		
33	QCO_NOT_INT	55	QCO_POKEB	77	QCO_LPRINT_SP		
34	QCO_AND_INT	56	QCO_POKEW	78	QCO_LPRINT_CR		
35	QCO_OR_INT	57	QCO_RAISE	79	QCO_RETURN		
36	QCO_LT_NUM	58	QCO_RANDOMIZE	7A	QCO_RETURN_NOUGHT		
37	QCO_LTE_NUM	59	QCO_STOP	7B	QCO_RETURN_ZERO		
38	QCO_GT_NUM	5A	QCO_TRAP	7C	QCO_RETURN_NULL		
39	QCO_GTE_NUM	5B	QCO_APPEND	7D	QCO_PROC		
3A	QCO_NE_NUM	5C	QCO_CLOSE	7E	QCO_BRA_FALSE		
3B	QCO_EQ_NUM	5D	QCO_COPY	7F	QCO_ASS_INT		
3C	QCO_ADD_NUM	5E	QCO_CREATE	80	QCO_ASS_NUM		
3D	QCO_SUB_NUM	5F	QCO_DELETE	81	QCO_ASS_STR		
3E	QCO_MUL_NUM	60	QCO_ERASE	82	QCO_DROP_BYTE		
3F	QCO_DIV_NUM	61	QCO_FIRST	83	QCO_DROP_WORD		
40	QCO_POW_NUM	62	QCO_LAST	84	QCO_DROP_NUM		
41	QCO_UMIN_NUM	63	QCO_NEXT	85	QCO_DROP_STR		
42	QCO_NOT_NUM	64	QCO_BACK	86	QCO_INT_TO_NUM		
43	QCO_AND_NUM	65	QCO_OPEN	87	QCO_NUM_TO_INT		
44	QCO_OR_NUM	66	QCO_POSITION	88	QCO_END_FIELDS		
45	QCO_LT_STR	67	QCO_RENAME	89	QCO_RUN_ASSEM		
46	QCO_LTE_STR	68	QCO_UPDATE				

INDEX OF FUNCTIONS

Functions that are only available on the LZ model range are marked by an asterisk.

8A	RTF_ADDR	A0	RTF_VIEW	B6	RTF_SPACE	D6*	RTF_CLOCK
8B	RTF_ASC	A1	RTF_YEAR	B7	RTF_DIR	D7*	RTF_DOW

8C	RTF_DAY	A2	RTF_COUNT	B8	RTF_CHR	D8*	RTF_FINDW
8D	RTF_DISP	A3	RTF_EOF	B9	RTF_DATIM	D9*	RTF_MENUN
8E	RTF_ERR	A4	RTF_EXIST	BA	RTF_SERR	DA*	RTF_WEEK
8F	RTF_FIND	A5	RTF_POS	BB	RTF_FIX	DB*	RTF_ACOS
90	RTF_FREE	A6	RTF_ABS	BC	RTF_GEN	DC*	RTF_ASIN
91	RTF_GET	A7	RTF_ATAN	BD	RTF_SGET	DD*	RTF_DAYS
92	RTF_HOUR	A8	RTF_COS	BE	RTF_HEX	DE*	RTF_MAX
93	RTF_IABS	A9	RTF_DEG	BF	RTF_SKEY	DF*	RTF_MEAN
94	RTF_INT	AA	RTF_EXP	C0	RTF_LEFT	E0*	RTF_MIN
95	RTF_KEY	AB	RTF_FLT	C1	RTF_LOWER	E1*	RTF_STD
96	RTF_LEN	AC	RTF_INTF	C2	RTF_MID	E2*	RTF_SUM
97	RTF_LOC	AD	RTF_LN	C3	RTF_NUM	E3*	RTF_VAR
98	RTF_MENU	AE	RTF_LOG	C4	RTF_RIGHT	E4*	RTF_DAYNAME
99	RTF_MINUTE	AF	RTF_PI	C5	RTF_REPT	E5*	RTF_DIRW
9A	RTF_MONTH	B0	RTF_RAD	C6	RTF_SCI	E6*	RTF_MONTHNAME
9B	RTF_PEEKB	B1	RTF_RND	C7	RTF_UPPER		
9C	RTF_PEEKW	B2	RTF_SIN	C8	RTF_SUSR		
9D	RTF_RECSIZE	B3	RTF_SQR	C9	RTF_SADDR		
9E	RTF_SECOND	B4	RTF_TAN				
9F	RTF_IUSR	B5	RTF_VAL				

EXAMPLES

In these examples all values are given in hexadecimal; word values are given as 4 digits, bytes as 2 digits each one separated by a space. If values are undefined they are written as **.

EXAMPLE 1

Source code:

```
EX1:
LOCAL A$(5)
A$="ABC"
```

The Q code header is:

```
High memory     0009            size of the variables on stack
                000A            length of Q code
                00              number of parameters
                                  type of parameter
                0000            size of global area
                                  global name
                                  global type
                                  offset
                0000            size of externals
                                  external name
                                  external type
                0003            bytes of string fix-ups
                FFF7              string fix-up offset (from FP)
                05                max length of string
Low memory      0000            bytes of array fix-ups
                                  array fix-up offset (from FP)
                                  size of array
```

The Q code is:

```
                0F FFF8         QI_LS_STR_SIM_FP
                24              QI_STR_CON
                03 41 42 43     "ABC"
                81              QCO_ASS_STR
                7B              QCO_RETURN_ZERO
```

If this program is run on a CM the stack looks like:

```
        Initially           Left Side   Constant    Assign      On Return
3EFF    '1'                 '1'         '1'         '1'         '1'
3EFE    'X'                 'X'         'X'         'X'         'X'
3EFD    'E'                 'E'         'E'         'E'         'E'
3EFC    ':'                 ':'         ':'         ':'         ':'
3EFB    'A'                 'A'         'A'         'A'         'A'
3EFA    05                  05          05          05          05
3EF9    00   - Top proc     00          00          00          00
3EF8    00   - No. paras    00          00          00          00
3EF6    3EF9 - Return PC    3EF9        3EF9        3EF9        3EF9
```



```
3EF4    0000 - ONERR        0000        0000        0000        0000
3EF2    3EDB - BASE SP      3EDB        3EDB        3EDB        3EDB
3EF0    0000 - FP           0000        0000        0000        0000
3EEE    3EEE - Global table 3EEE        3EEE        3EEE        3EEE
3EED    00                  00          00          00          00
3EEC    00                  00          00          00          00
3EEB    00                  00          'C'         'C'         'C'
3EEA    00                  00          'B'         'B'         'B'
3EE9    00                  00          'A'         'A'         'A'
3EE8    00                  00          03          03          03
3EE7    05                  05          05          05          05
3EE6    **                  **          **          **          **
3EE5    **                  **          **          **          **
3EE4    QCO_RETURN_ZERO     7B          7B          7B          7B
3EE3    QCO_ASS_STR         81          81          81          81
3EE2    'C'                 'C'         'C'         'C'         'C'
3EE1    'B'                 'B'         'B'         'B'         'B'
3EE0    'A'                 'A'         'A'         'A'         'A'
3EDF    03                  03          03          03          03
3EDE    QI_STR_CON          24          24          24          24
3EDC    FFF8                FFF8        FFF8        FFF8        FFF8
3EDB    QI_LS_STR_SIM_FP    0F          0F          0F          0F
3EDA    **                  3EE8        3EE8        **          00
3ED9    **                  05          05          **          00
3ED8    **                  00          00          **          00
3ED7    **                  **          'C'         **          00
3ED6    **                  **          'B'         **          00
3ED5    **                  **          'A'         **          00
3ED4    **                  **          03          **          00
3ED3    **                  **          **          **          00
3ED2    **                  **          **          **          00

FP      3EF0                3EF0        3EF0        3EF0
PC      3EDB                3EDE        3EE3        3EE4
SP      3EDB                3ED8        3ED4        3EDB
```

EXAMPLE 2

When a file is created the operator QCO_CREATE is followed by the logical name to use and the field type and names. The list is terminated by the operator QCO_END_FIELDS.

For example:

```
CREATE "B:ABC",B,AAA$,B%,CC
```

is translated as the Q code:

```
24                      QI_STR_CON
05 42 3A 41 42 43       "B:ABC"
5E                      QCO_CREATE
01                      Logical name B
02                      Type string
04 41 41 41 24          "AAA$"
00                      Type integer
02 42 25                "B%"
01                      Type floating point
02 43 43                "CC"
88                      QCO_END_FIELDS
```

EXAMPLE 3

```
RECURS:(I%)
IF I%
  RECURS:(I%-1)
ENDIF
```

Looks like this on the stack:

```
        Address         Contents        Description

        3D5A            0010            Parameter
        3D59            00              Parameter type
        3D58            01              Number of parameters
        3D57            41              Device A
```

```
3D55            3D6D            Return RTA_PC
3D53            0000            ONERR address
3D51            3D29            BASE SP
3D4F            3D82            Previous FP
3D4D            3D4D            Globals start address
3D4B            3D5A            Indirect address to parameter
3D49            **
3D48            7B              QCO_RETURN_ZERO
3D47            84              QCO_DROP_NUM
3D40            "RECURS"
3D3F            7D              QCO_PROC
3D3E            01
3D3D            20              QCO_STK_LIT_BYTE
3D3C            00
3D3B            20              QCO_STK_LIT_BYTE
3D3A            2E              QCO_SUB_INT
3D38            0001
3D37            22              QI_INT_CON
3D35            FFFC
3D34            07              QI_INT_SIM_IND
3D32            001B
3D31            7E              QCO_BRA_FALSE
3D2F            FFFC
3D2E            07              QI_INT_SIM_IND
3D2C            000F            Parameter for next call
3D2B            00              parameter type
3D2A            01              parameter count
```

Note that the top 4 byte and the bottom 4 bytes are almost identical, this is shown at the point where the procedure is about to be invoked:

```
RTA_PC          3D3F
RTA_SP          3D2A
RTA_FP          3D4F
```

EXAMPLE 4

Source code:

```
EX4:(PPP$)
LOCAL A$(5)
GLOBAL B,C%(3),D$(5)
J$=PPP$
```

The Q code header is:

```
0035            size of the variables on stack
0008            size of Q code length
01              number of parameters
02                type of parameter
0011            size of global area
01 42             global name
01                global type
FFE1              offset
02 43 25          global name
03                global type
FFD9              offset
02 44 24          global name
02                global type
FFD3              offset
0004            bytes of externals
02 4A 24          external name
02                external type
0006            bytes of string fix-ups
FFCB              string fix-up offset (from FP)
05                max length of string
FFD2              string fix-up offset (from FP)
05                max length of string
0004            bytes of array fix-ups
FFD9              array fix-up offset (from FP)
0003              size of array
```


The Q code is:

```
16 FFE9         QI_LS_STR_SIM_IND
09 FFEB         QI_STR_SIM_IND
81              QCO_ASS_STR
7B              QCO_RETURN_ZERO
```

If this program is run on a CM from the procedure:

```
XXX:
GLOBAL J$(3)
EX4:("RST")
```

The stack looks like:

```
3EFA            "A:XXX"
3EF9            00                      Number of parameters
3EF8            00                      Top procedure
3EF6            3EF9                    Return PC
3EF4            0000                    ONERR address
3EF2            3ED1                    BASE SP
3EF0            0000                    FP
3EEE            3EE8                    Start of global table
3EEC            3EE4                    Address of global
3EEB            02                      Global type
3EE8            "J$"                    Global name
3EE3            03 00 00 00 00          Global J$
3EE1            **
3EE0            7B                      QCO_RETURN_ZERO
3EDF            84                      QCO_DROP_NUM
3EDB            "EX4"
3EDA            7D                      QCO_PROC
3ED8            20 01                   QI_STK_LIT_BYTE
3ED6            20 02                   QI_STK_LIT_BYTE
3ED2            "RST"
3ED1            24                      QI_STR_CON
3ECD            "RST"                   Parameter
3ECC            02                      Parameter type
3ECB            01                      Number of parameters
3ECA            00                      Device A:
3EC8            3EDA                    Return PC
3EC6            0000                    ONERR
3EC4            3E83                    BASE SP
3EC2            3EF0                    FP
3EC0            3EAF                    Start global table
3EBE            3E95
3EBD            02
3EBA            02 44 24                Global D$
3EB8            3E9B
3EB7            03
3EB4            02 43 25                Global C%()
3EB2            3EA3
3EB1            01
3EAF            01 42                   Global B
3EAD            3ECD                    Indirection to PPP$
3EAB            3EE4                    Indirection to J$
3EA3            00 00 00 00 00 00 00 00         GLOBAL B
3E9B            00 03 00 00 00 00 00 00         GLOBAL C%()
3E94            05 00 00 00 00 00 00            GLOBAL D$
3E8D            05 00 00 00 00 00 00            LOCAL A$
3E8B            **
3E8A            7B                      QCO_RETURN_ZERO
3E89            81                      QCO_ASS_STR
3E87            FFEB
3E86            09                      QI_STR_SIM_IND
3E84            FFE9
3E83            16                      QI_LS_STR_SIM_IND
```

When running EX4 the offset FFE9 is added to RTA_FP (3EC2) to give 3EAB. The address at 3EAB is 3EE4 which is the address of the global J$. This address with a non-field flag is stacked. Similarly FFEB is added to RTA_FP to give 3EAD, which contains the address 3ECD, the address of the parameter PPP$.

EXAMPLE 5

Source code:

```
TOP:
PRINT ABC:(GET)
GET
ABC:(N%)
RETURN(N%*N%)
```

At the point when ABC: has just been called the stack looks like:

```
3EFA        "A:TOP"
3EF9        00          NO. of parameters
3EF8        00          Top procedure
3EF6        3EF9        Return PC
3EF4        0000        ONERR address
3EF2        3EDD        BASE SP
3EF0        0000        FP
3EEE        3EEE        Global table
3EEC        **
3EEB        7B          QCO_RETURN_ZERO
3EEA        83          QCO_DROP_WORD
3EE9        91          RTF_GET
3EE8        73          QCO_PRINT_CR
3EE7        70          QCO_PRINT_NUM
3EE3        "ABC"
3EE2        7D          QCO_PROC
3EE0        20 01       QI_STK_LIT_BYTE
3EDE        20 00       QI_STK_LIT_BYTE
3EDD        91          RTF_GET
3EDB        0020
3EDA        00          Integer
3ED9        01          No. parameters
3ED8        41          Device A:
3ED6        3EE2        Return PC
3ED4        0000        ONERR
3ED2        3EC1        BASE SP
3ED0        3EF0        FP
3ECE        3ECE        global table
3ECC        3EE4        Address of N%
3ECA        **
3EC9        79          QCO_RETURN
3EC8        86          QCO_INT_TO_NUM
3EC7        2F          QCO_MUL_INT
3EC4        07 FFF7     QI_INT_SIM_IND
3EC1        07 FFF7     QI_INT_SIM_IND
3EBF        0020            0400            0300
3EBD        0020            **              1024
3EBB        **              **              0000
3EB9        **              **              0000

PC          3EC7            3EC8            3EC9
```


TABLE INTERPRETER

This chapter is included for completeness only, since it is not expected that these services will be used outside of PSION.

Described here is Psion's Table Interpreter and the very specialised OS services used in it.

THE TABLE INTERPRETER

The language translator and the scientific functions in the operating system are called through an interpreter. The use of an interpreter allows a major saving in code size, since subroutine calls which would need 2 or 3 bytes, can be replaced by the single interpreted byte. The table of bytes to be interpreted is compiled from relatively high-level source code by a specific TABLE COMPILER which is not available outside of PSION; this is therefore only a brief description.

The table interpreter can usefully be compared to a microprocessor system which uses microcode. There, a machine-code opcode is read and the microprocessor runs the appropriate microcode program for that one opcode, updating the program-counter to accommodate any opcode arguments. In this analogy, the table interpreter itself corresponds to the microprocessor, while each byte in the table to be interpreted corresponds to an opcode or its required argument.

Each byte of the table specifies an action (or its parameters) to be performed through the interpreter, so that a customised set of actions can be designed for some particular purpose. There is also a set of predefined actions (see below) which have byte values from 0 to 20, so user-defined actions can have values consecutively from 21 to 255. Action 20 ends the interpretation.

For each action defined by the user, a normal machine-code subroutine must be written for calling from the interpreter. In the analogy above, the subroutine for an action corresponds to the microcode program which actually performs the opcode.

The compiled table has a fixed format. The first two bytes must contain the offset from the beginning of the table to an array of pointers to the user-defined action subroutines; the pointers must be ordered exactly as their corresponding user-defined actions (21-255). These two bytes are followed by the action-specifying bytes, each of which is followed by its parameters. The table program-counter, ITA_PCNT, is used by the interpreter to step through the table. The subroutines for the actions must return in the B register the number of table bytes to be stepped over till the next action. For instance, if the action has 2 parameter-bytes, 3 must be returned in B.

The interpreter maintains its own stack which is used by some of the predefined actions. The stack grows downwards in memory and the stack-pointer points to the last word pushed.

WARNING

The table interpreter itself cannot be called from within a table. This would result in the corruption of the system variables used by the interpreter. As mentioned above, the OS calls the interpreter for the language translator and for the scientific functions, so that none of these may be used within a table.

TABLE REGISTERS

The table interpreter has a block of 32 bytes of RAM allocated to it, which are referred to as table-registers. This block is labeled ITT_REGS. The values from $E0 to $FF are used to access these byte registers when passed as parameters to the actions. If a parameter's value is in this range, $E0 is subtracted from it and the result is the offset into ITT_REGS. In this documentation, the registers are named tabreg0, tabreg1, etc. starting at ITT_REGS.

Note that the subroutine for a user-defined action can in fact use parameter-bytes in any way needed, whatever their values, but the service routines IT$GVAL and IT$RADD, and also the predefined actions, use the above convention for accessing the registers.

EXAMPLE:

The code to load B with the constant parameter that is directly after the action-specifying byte. ITA_PCNT points to the action-specifying byte currently being interpreted.

```
        LDX     ITA_PCNT:       ; Get the table program-counter
        LDA     B,1,X           ; Get the parameter byte
```

If the X register has not been corrupted since entry to the subroutine, it points to the current action-specifying byte already.

THE PREDEFINED ACTIONS

The byte values 0 to 20 specify actions that are predefined in OS. These actions provide a framework on which to build a useful, customised programming language. These include handling of control constructs, table-subroutines, assignment to table-registers and calling regular machine-code subroutines.

This section describes each of these actions in some detail, using the following terminology and conventions.

1. The ACTION NUMBER is the byte value specifying the action in a table.
2. The ACTION NAMES used in the examples are defined as equivalent to their ACTION NUMBERS (e.g. #define CALL_ACTION 1).

3. The ACTION PARAMETERS follow the ACTION NUMBER in the table. "P1_B" refers to the first parameter, where "_B" specifies that it is a byte. "P2_W" refers to the second parameter, where "_W" specifies that it is a word.
4. The base of the table to be interpreted is assumed to be at the label TABLE_BASE.

RETURN

ACTION NUMBER: 0
ACTION PARAMETERS: None.

DESCRIPTION

This action returns from a table-subroutine in much the same way as the 'RTS' instruction returns from a machine-code subroutine. The return address is popped from the table-stack. Table-subroutines are invoked by the predefined actions CALL or JSR.

EXAMPLE:

```
        .BYTE   JSR_ACTION
        .WORD   SUBRTN1-TABLE_BASE      ; Offset to the table-subroutine
          .
          .
SUBRTN1:
          .
          .
        .BYTE   RETURN_ACTION
```

CALL

ACTION NUMBER: 1
ACTION PARAMETERS:
P1_B - The number of parameter-bytes for the called subroutine. (See the warning below).
P2_W - The offset to the subroutine.
Subsequent bytes - Up to 31 parameter-bytes for the subroutine being called.

DESCRIPTION

This action calls a table-subroutine offset from TABLE_BASE by P2_W. The number of parameter-bytes to be passed to the table-subroutine must be specified by P1_B. The table-subroutine's parameter-bytes must be placed from action parameter-bytes 4 up to 34.

WARNING

The value in P1_B is stored in tabreg0 ($E0), and the subsequent parameter bytes to the table-subroutine are stored consecutively from tabreg1 onwards. While allowing the table-subroutine to access these bytes, this prevents more than 31 parameter-bytes being passable to the table subroutine. Extreme care must be taken when using the CALL action, since the registers corrupted in this way are **not** restored.

Note that if no parameters need to be passed, then the JSR action ought to be used.

EXAMPLE:

To call a table-subroutine 255 bytes from the start of the table, passing 2 parameter-bytes.

```
        .BYTE   CALL_ACTION
        .BYTE   2       ; Two parameters to be passed to table-subroutine
        .WORD   255     ; Word offset from start of table to
                        ;   table-subroutine
        .BYTE   6,8     ; Pass 6 and 8 to the table-subroutine
```

If the table sub-routine label is SUBRTN1, the following table extract applies. The parameter byte-count for SUBRTN1 is placed by the CALL action in tabreg0 ($E0), and the parameter-bytes themselves in tabreg1 ($E1), tabreg2 ($E2), etc.

```
        .BYTE   CALL_ACTION
        .BYTE   3               ; 3 parameters passed to table-subroutine
        .WORD   SUBRTN1-TABLE_BASE
                                ; Word offset from start of table to the
                                ;   table-subroutine
        .BYTE   2,25,6          ; Pass 2,25 and 6 to the table-subroutine
        .BYTE   END_ACTION      ; (to end the interpretation)
SUBRTN1:
        .BYTE   30              ; User-defined action
        .BYTE   1               ; Parameter-byte for Action Number 30
        .BYTE   RETURN_ACTION   ; RETURN from table-subroutine
```

IF

ACTION NUMBER: 2
ACTION PARAMETERS:
P1_B - Signed offset for branching.

DESCRIPTION

Tests the system variable ITB_TEST. If false, the signed value in P1_B is added to ITA_PCNT; if true, 2 is added to ITA_PCNT. In either case ITA_PCNT points to the next action which needs to be interpreted in the table.

EXAMPLE:

An IF/ELSE control construct. If tabreg0 is equal to 2, do action number 35, otherwise do action number 40.

```
        .BYTE   EQL_ACTION
        .BYTE   $E0,2           ; Set ITB_TEST if tabreg0 ($E0) equals 2
IF_POS:
        .BYTE   IF_ACTION
        .BYTE   ELSE_POS-IF_POs
                                ; Offset for potential branch
        .BYTE   35              ; User-defined action
 BRANCH_POs:
        .BYTE   BRANCH_ACTION
        .BYTE   ENDIF_POS-BRANCH_POs
                                ; Offset for BRANCH
ELSE_POs:
        .BYTE   40              ; User-defined action
ENDIF_POs:
```

IF_NOT

ACTION NUMBER: 3
ACTION PARAMETERS:
P1_B - Signed offset for branching.

DESCRIPTION

Tests the system variable ITB_TEST. If true, the signed value in P1_B is added to ITA_PCNT; if false, 2 is added to ITA_PCNT. In either case ITA_PCNT points to the next action which needs to be interpreted in the table.

EXAMPLE:

If tabreg1 is not equal to 2 do action number 35, otherwise branch to ENDIF_POS.

```
        .BYTE   EQL_ACTION
        .BYTE   $E1,2           ; Set ITB_TEST if tabreg1 ($E1) equals 2
IF_NOT_POs:
        .BYTE   IF_NOT_ACTION
        .BYTE   ENDIF_POS-IF_NOT_POs
                                ; Offset for potential branch
        .BYTE   35              ; User-defined action ENDIF_POs:
```

CASE

ACTION NUMBER: 4
ACTION PARAMETERS:
P1_B - Selector parameter.
P2_W - Offset to case-table.

DESCRIPTION

Causes table interpretation to continue at different addresses depending on the value in the selector parameter, P1_B. The user needs to set up a case-table with the format shown below. This is best illustrated by the example.

EXAMPLE:

Use the selector parameter tabreg1 ($E1) to branch to the required position in the table. If tabreg1 equals 2 go to CASE2 to do action 38 and then go to END_CASE. If tabreg1 equals 5 go to CASE5 to do actions 31 and 45, otherwise go to DEFAULT to do just action 45.

```
DO_CASE:
        .BYTE   CASE_ACTION
        .BYTE   $E1                     ; Selector parameter is tabreg1
        .WORD   CASE_TABLE-TABLE_BASE   ; Offset to CASE_TABLE CASE2:
        .BYTE   38                      ; User-defined action
B1:     .BYTE   BRANCH_ACTION           ; Branch out of the case construct
        .BYTE   END_CASE-B1
CASE5:
        .BYTE   31                      ; Do user-defined action 31 and
                                        ; drop through to action 45
DEFAULT:
        .BYTE   45
END_CASE:
          .
          .
          .
CASE_TABLE:
```

```
        .BYTE   2                       ; If selector equals 2 go to CASE2
        .WORD   CASE2-TABLE_BASE
        .BYTE   5                       ; If selector equals 5 go to CASE5
        .WORD   CASE5-TABLE_BASE
        .BYTE   255                     ; If selector is not 2 or 5 go to
        .WORD   DEFAULT-TABLE_BASE      ;   DEFAULT
```

VECTOR

ACTION NUMBER: 5
ACTION PARAMETERS:
P1_B - Selector parameter.
P2_W - Offset to vector-table.

DESCRIPTION

Causes table interpretation to continue at different places depending on the value in the selector parameter, P1_B. The selector parameter is used to index into the vector-table. Each vector in the vector-table is a word offset from TABLE_BASE to the next action to be interpreted. A selector equal to 2, for example, will get the 3rd offset in the vector-table.

EXAMPLE:

Use the selector parameter tabreg2 ($E2) to branch to the required position in the table.

```
DO_VECTOR:
        .BYTE   VECTOR_ACTION
        .BYTE   $E2                     ; Selector parameter is tabreg2
        .WORD   VECTOR_TABLE-TABLE_BASE
                                        ; Offset to VECTOR_TABLE
ADDR0:
        .BYTE   38                      ; User-defined action
B1:     .BYTE   BRANCH_ACTION           ; Branch out of the case construct
        .BYTE   END_VECTOR-B1
ADDR1:
        .BYTE   31                      ; Do user-defined action 31 and
ADDR2:
        .BYTE   45                      ; drop through to action 45
END_VECTOR:
          .
          .
          .
VECTOR_TABLE:
        .WORD   ADDR0-TABLE_BASE        ; Offset when selector equals 0
        .WORD   ADDR1-TABLE_BASE        ; Offset when selector equals 1
        .WORD   ADDR2-TABLE_BASE        ; Offset when selector equals 2
```

GOTO

ACTION NUMBER: 6
ACTION PARAMETERS:
P1_W - Offset from TABLE_BASE to next action to be interpreted.

DESCRIPTION

Causes interpretation to continue at the action that is offset by P1_W from TABLE_BASE.

EXAMPLE:

Go to ADDR1.

```
        .BYTE   GOTO_ACTION
        .WORD   ADDR1-TABLE_BASE
          .
          .
ADDR1:
```

BRANCH

ACTION NUMBER: 7
ACTION PARAMETERS:
P1_B - Signed offset for branching.

DESCRIPTION

Causes interpretation to branch to the action that is offset by P1_B from the current action. This is used for short branches forward or backward.

EXAMPLE:

Branch to ADDR1.


```
DO_BRANCH:
        .BYTE   BRANCH_ACTION
        .BYTE   ADDR1-DO_BRANCH
          .
          .
ADDR1:
```

EQL

ACTION NUMBER: 8
ACTION PARAMETERS:
P1_B - 1st operand.
P2_B - 2nd operand.

DESCRIPTION

Compares P1_B with P2_B, setting ITB_TEST if equal, otherwise clearing it. Either operand may be a register or a constant.

EXAMPLE:

```
        .BYTE   EQL_ACTION
        .BYTE   $E0,$E3         ; Set ITB_TEST if tabreg0 equals tabreg3
IF_POs:
        .BYTE   IF_ACTION       ; Test ITB_TEST and branch if set
        .BYTE   ENDIF_POS-IF_POs
                                ; Offset for potential branch
        .BYTE   35              ; User-defined action ENDIF_POs:
```

NEQ

ACTION NUMBER: 9
ACTION PARAMETERS:
P1_B - 1st operand. P2_B - 2nd operand.

DESCRIPTION

Compares P1_B with P2_B, clearing ITB_TEST if equal, otherwise setting it. Either operand may be a register or a constant.

EXAMPLE:

```
        .BYTE   NEQ_ACTION      ; Set ITB_TEST if tabreg0 not equal to 2
        .BYTE   $E0,2
IF_POs:
        .BYTE   IF_ACTION       ; Branch if ITB_TEST set
        .BYTE   ENDIF_POs - IF_POs
                                ; Offset for potential branch
        .BYTE   35              ; User-defined action
ENDIF_POs:
```

ASSIGN

ACTION NUMBER: 10
ACTION PARAMETERS:
P1_B - Register to be assigned to.
P2_B - Operand to assign to the register.

DESCRIPTION

Assigns P2_B, which may be a table-register or a constant, to the table-register specified by P1_B.

EXAMPLE:

Assign the value in tabreg4 to tabreg1.

```
        .BYTE   ASSIGN_ACTION
        .BYTE   $E1,$E4
```

ADD2

ACTION NUMBER: 11
ACTION PARAMETERS:
P1_B - Table-register operand to be added to.
P2_B - Operand to be added.

DESCRIPTION

Adds P2_B, which may be a table-register or a constant, to the table-register specified by P1_B.

EXAMPLE:

Add 6 to tabreg5.

```
        .BYTE   ADD2_ACTION
        .BYTE   $E5,6
```

SUB2

ACTION NUMBER: 12
ACTION PARAMETERS:
P1_B - Table-register operand from which to subtract.
P2_B - Operand to be subtracted.

DESCRIPTION

Subtracts P2_B, which may be a table-register or a constant, from the table-register specified by P1_B.

EXAMPLE:

Subtract 6 from tabreg5.

```
        .BYTE   SUB2_ACTION
        .BYTE   $E5,6
```

PUSH

ACTION NUMBER: 13
ACTION PARAMETERS:
P1_W - Offset to address in table to be pushed onto the table stack.

DESCRIPTION

Pushes an address onto the table-stack. The address pushed is offset from TABLE_BASE by the word in P1_W.

EXAMPLE:

Push the address of LABEL1 onto the table-stack.

```
        .BYTE   PUSH_ACTION
        .WORD   LABEL1-TABLE_BASE
          .
          .
LABEL1:
```

CALL_MC

ACTION NUMBER: 14
ACTION PARAMETERS:
P1_W - Address of machine-code subroutine to be called.

DESCRIPTION

Calls the machine-code subroutine at the address in P1_W. The value in tabreg0 ($E0) is passed to the subroutine in the machine-register B.

EXAMPLE:

Call MC_ROUTINE passing 6.

```
        .BYTE   ASSIGN_ACTION
        .BYTE   $E0,6                   ; Assign 6 to tabreg0
        .BYTE   CALL_MC_ACTION
        .WORD   MC_ROUTINE
```

POP

ACTION NUMBER: 15
ACTION PARAMETERS: None.

DESCRIPTION

Pops a word from the table-stack by incrementing ITA_SPTR by 2. The value popped is then lost.

JSR

ACTION NUMBER: 16
ACTION PARAMETERS:
P1_W - Offset from TABLE_BASE to the table-subroutine.

DESCRIPTION

Calls a table-subroutine with no parameters. The table-subroutine is offset from the base of the table by P1_W.

EXAMPLE:

```
        .BYTE   JSR_ACTION
        .WORD   SUBRTN1-TABLE_BASE
          .
          .
SUBRTN1:
          .
          .
        .BYTE   RETURN_ACTION
```


RANGE

ACTION NUMBER: 17
ACTION PARAMETERS:
P1_B - Value to be tested.
P2_B - Lower limit.
P3_B - Upper limit.

DESCRIPTION

Checks that the value in P1_B is in the inclusive range with lower limit in P2_B and upper limit in P3_B. Any of these values may be table-registers or constants. If the value is in range ITB_TEST is set, otherwise it is cleared.

EXAMPLE:

Do user-defined action 35 only if tabreg1 is in ASCII range '0' to '9'.

```
        .BYTE   RANGE_ACTION
        .BYTE   $E1,A/0/,A/9/   ; ('0' <= tabreg1 <= '9')?
B1:     .BYTE   IF_ACTION       ; Test ITB_TEST and branch if false
        .BYTE   ENDIF_POS-B1    ; Offset for branch
        .BYTE   35              ; User-defined action
ENDIF_POs:
```

LOADB

ACTION NUMBER: 18
ACTION PARAMETERS:
P1_B - Table-register to be loaded.
P2_W - Address of byte to be stored.

DESCRIPTION

Load the table-register specified by P1_B, with the byte at the address in P2_W.

EXAMPLE:

Load tabreg1 with the byte at ITB_TEST.

```
        .BYTE   LOADB_ACTION
        .BYTE   $E1
        .WORD   ITB_TEST
```

STOREB

ACTION NUMBER: 19
ACTION PARAMETERS:
P1_B - Operand to be stored.
P2_W - Address to be loaded.

DESCRIPTION

Store the byte in P1_B at the address in P2_W. P1_B may be a table-register or a constant.

EXAMPLE:

Store the byte in tabreg1 in the variable USER_FLAG.

```
        .BYTE   STOREB_ACTION
        .BYTE   $E1
        .WORD   USER_FLAG
```

END

ACTION NUMBER: 20
ACTION PARAMETERS: None.

DESCRIPTION

Terminates the interpretation and causes the OS service routine IT$STRT to return with the Z-flag set (useful for signaling a non-error exit).

SYSTEM SERVICES

The OS service routines for this chapter include the table interpreter itself, as well as several that are useful when writing a subroutine for a user-defined action. This section should be read after the rest of the chapter.

IT$GVAL

VECTOR NUMBER: 066
INPUT PARAMETERS:
B register - Offset to ACTION PARAMETER.
OUTPUT VALUES:

B register - Value of ACTION PARAMETER.
REGISTERS PRESERVED: A,X

DESCRIPTION

Get the value of the ACTION PARAMETER byte offset by B bytes from ITA_PCNT, into B. If the value specifies a table-register (i.e. is in range $E0 to $FF) get the value in that register into B instead.

EXAMPLE:

Get value offset by 3 bytes from ITA_PCNT into B.

```
        LDA     B,#3
        OS      IT$GVAL
```

IT$RADD

VECTOR NUMBER: 067
INPUT PARAMETERS:
B register - Offset to ACTION PARAMETER specifying a register.
OUTPUT VALUES:
X register - Address of the table-register.
REGISTERS PRESERVED: A,B

DESCRIPTION

Sets X to point to the table-register specified B bytes after ITA_PCNT.

EXAMPLE:

Get address of table-register 2 bytes after ITA_PCNT.

```
        LDA     B,#2
        OS      IT$RADD
```

IT$TADD

VECTOR NUMBER: 069
INPUT PARAMETERS:
B register - Offset to ACTION PARAMETER.
OUTPUT VALUES:
D register - Word that is offset by B from ITA_PCNT.
REGISTERS PRESERVED: X.

DESCRIPTION

Sets D to word value at B bytes after ITA_PCNT. This routine is useful in OS since the table-compiler compiles all labels in a table as a word-offset from the beginning of the table, for relocatability. This value must then be added to the address of the base of the table to obtain the address of the label.

EXAMPLE:

Get the address of the action specified 4 bytes after the current ACTION NUMBER.

```
        LDA     B,#4
        OS      IT$TADD
        ADDD    ITA_BASE:
```

IT$STRT

VECTOR NUMBER: 068
INPUT PARAMETERS:
D register - Base address of the table to be interpreted.
OUTPUT VALUES:
B register - Clear if END action performed last.
Z-flag - Set if END action performed last.

DESCRIPTION

This is the table interpreter itself, interpreting the table of bytes starting at D. The first 2 bytes at D are the offset from the start of the table to the array of pointers to the user-defined action subroutines. The absolute address of this array is saved in ITA_UVC. The table program-counter ITA_PCNT points to the current byte being interpreted, initially pointing to the third byte. On entry to the subroutine for a user-defined action, X also contains the value in ITA_PCNT. The table is interpreted until the END action (with value 20) is fetched, upon which IT$STRT returns with the B register cleared and the Z-flag set. An alternative method for ending interpretation, for exceptional cases such as error handling, is to perform a user-defined action as illustrated by the example below.

WARNING

IT$STRT must not be called recursively (i.e. by any subroutine for a user-defined action).

EXAMPLE:

Interpret the table at TABLE_BASE. Note that although this is a trivial use of the table interpreter, it usefully illustrates the structure of a self-contained table. Errors are handled by performing the user-defined action ERROR_USER_ACTION, value 21, which terminates interpretation with the non-zero error code in B and Z-flag clear.

```
; The user-defined ACTION NUMBERS:
#define ERROR_USER_ACTION 21    ; Action subroutine is ERROR_ACT
#define ISDIGIT_USER_ACTION 22  ; Action subroutine is ISDIGIT_ACT

; The Assembler Code to call the interpreter:
CALL_INTERPRETER:
        LDD     #TABLE_BASE
        OS      IT$STRT
        BNE     ERROR           ; IT$STRT returns Z-flag cleared when
                                ;   ERROR_USER_ACTION performed, with
                                ;   the error code in B
        RTS

; The table to be interpreted:
TABLE_BASE::
        .WORD   JUMP_ADDRS-TABLE_BASE   ; Offset to array of pointers
                                        ;   to subroutines for the user-
                                        ;   defined actions
        .BYTE   ISDIGIT_USER_ACTION     ; Is value in tabreg1 a digit?
        .BYTE   $E1                     ;   (sets ITB_TEST if it is)
IFNPOS: .BYTE   IF_NOT_ACTION           ; Predefined action: do action
                                        ;   ERROR_USER_ACTION if
                                        ;   ITB_TEST is 0 (false)
        .BYTE   ENDIFN-IFNPOS           ; Signed offset to ENDIFN
        .BYTE   ERROR_USER_ACTION       ; End interpretation with
        .BYTE   1                       ;   error-code 1 in B and
                                        ;   Z-flag clear
ENDIFN:
        .BYTE   END_ACTION              ; Predefined action: end the
                                        ;   interpretation with B clear
                                        ;   and Z-flag set


; The array of pointers to subroutines for the 2 user-defined actions:
JUMP_ADDRS:
        .WORD   ERROR_ACT       ; For ACTION NUMBER 21
        .WORD   ISDIGIT_ACT     ; For ACTION NUMBER 22

; Subroutine for the user-defined action to end table-interpretation for
; error-handling, with an error code passed as a parameter-byte to the
; action:
ERROR_ACT:
        LDA     B,1,X           ; Get the 1st parameter-byte into B
                                ;   setting Z-flag if B=0, else reset it
        PULX                    ; Pull the return address (points back into
                                ;   the interpreter code, IT$STRT)
        RTS                     ; Return to the caller of IT$STRT with B
                                ;   and Z-flag conditioned appropriately

; Subroutine for the user-defined action to test whether the parameter-byte
; is an ASCII coded digit. Sets ITB_TEST if a digit:
ISDIGIT_ACT:
        LDA     B,#1
        STA     B,ITB_TEST      ; Assume it is a digit
        OS      IT$GVAL         ; Get the parameter-byte into B
        CMP     B,#^A/0/        ; Set carry if less than '0'
        BCS     1$
        ADD     B,#$C6          ; Add (-'9'-1) setting carry if not a digit
        BCC     2$
1$:     DEC     ITB_TEST        ; Not a digit
2$:     LDA     B,#2            ; Add 2 to ITA_PCNT, stepping over the
                                ;   ACTION NUMBER and the parameter-byte
        RTS
```

VARIABLE USAGE

Described below are the variables used by the table interpreter system services.

ITA_UVC Pointer to the array of addresses of the subroutines for the user-defined actions. This variable is set up on entry to the interpreter.

ITT_REGS Block of 32 8-bit table-registers. Used extensively by the predefined actions. See Table Registers.

ITT_STAK Table interpreter's stack used for table-subroutine calls and for saving temporary data. See the predefined actions CALL, JSR, PUSH and POP.

ITA_SPTR Table interpreter's stack-pointer pointing into the table interpreter's stack, ITT_STAK. The stack grows downwards in memory and the stack-pointer points to the last word pushed (i.e. ITA_SPTR is decremented before a word is pushed).

ITA_PCNT Table interpreter's program-counter. ITA_PCNT points to the ACTION NUMBER currently being interpreted. On entry to a user-defined action subroutine, the X register contains this same value.

ITA_BASE Pointer to the base of the table being interpreted. This is the same address passed as a parameter to IT$STRT, the interpreter itself.

ITB_TEST Flag set when a test is true. User-defined actions may use ITB_TEST, but note that it is affected by the predefined actions EQL, NEQ, RANGE and it is tested by the actions IF and IF_NOT.

Appendix A

PSION LCD Information by Zac Schroff Revised 1997 02 24

The Psion display is memory mapped to two addresses. Reading or writing to these addresses will access certain ports, as shown below.

Memory	Read	Write
0180	Busy & address counter	Commands
0181	Data at addr counter	Data at addr counter

Note that the address counter is incremented or decremented at each read from the port at 0181.

The busy & address counter register is split into two parts. The Msb (bit 7) of the byte is the busy flag (this flag MUST be zero when accessing the display). The other seven bits (0..6) are the address counter within its particular space (there are two spaces: display memory and character generator). Note that reads seem to be able to occur at any time, where writes and commands can sometimes take up to several milliseconds before the display is ready for the next command or
write. It is not documented what overrunning the screen will do.
Commands go to control port, optional data can follow. Some commands cause the data port to do certain things, others make the data port write to the character generator or the display. Note that the cursor is always displayed at the current offset for text accesses to the data port (moving the cursor moves where these accesses will occur).
Note that when the machine is in a `powered down' state, the display hardware gets no power. Therefore, the entire display state is quite indeterminate on power up. The machine keeps track of most of the display state in scratchpad memory.
See the book review: Machine Language Programming on the Psion Organiser by Bill Aitken for information about the system's internal memory structures.

Command bitmap	Command description (footnote 1)
00000000	NOP? (this command is not documented and seems to do nothing whatsoever on my CM or LZ64)
00000001	Clear display memory Move text cursor to home position Move display to home position (cancel offsets) Prepare for text writes to data port
0000001x	Move text cursor to home position Move display to home position (cancel offsets) Prepare for text writes to data port x values do not seem to matter? (footnote 6)
000001AD	Set address index auto-increment if A set Set address index auto-decrement if A clear Set display offset auto-increment if D set (footnote 7)
00001EUF	Set display enable to E Set underline cursor enable to U (footnote 2) Set flashing block cursor enable to F (footnote 2)
0001SDxx	If S clear and D clear, move the text cursor left If S clear and D set, move the text cursor right If S set and D clear, move the display offset left If S set and D set, move the display offset right x values do not seem to matter? (footnote 6)
001BLHxx	Set byte data port enable to B (footnote 3) Set two-line enable to L (footnote 4) Set 11 raster character mode to H (footnote 0.5) x values do not seem to matter? (footnote 6)
01AAAAAA	Set data index to character generator offset AAAAAA
1AAAAAAA	Set data index to display offset AAAAAAA

The character generator memory is documented by Psion, except for a few minor items that might prove useful with all the other data that are provided in this file. It seems that in 8-raster mode, there is enough character generator memory for eight characters. However, in 11-raster mode, there is only enough character generator memory for four characters. I have taken the 11-raster mappings straight from Hitachi's documentation, since I have not experimented with them.

Offset range	8-raster character	11-raster character
000000..000111	00,08	
001000..001111	01,09	
010000..010111	02,0A	
011000..011111	03,0B	
100000..100111	04,0C	
101000..101111	05,0D	
110000..110111	06,0E	
111000..111111	07,0F	
000000..001111		00,01,08,09

010000..011111 02,03,0A,0B

100000..101111 04,05,0C,0D

110000..111111 06,07,0E,0F

Pixels are mapped 5*8 as eight rows of xxxDDDDD where xxx does not matter, and DDDDD are the displayed pixels (1 is dark (on). In 5*11 mode, the pixels are mapped as 11 rows of xxxDDDDD where xxx does not matter, and DDDDD are displayed pixels as above, followed by five bytes that do not matter (they are ignored).
The CM and XP use the following display memory layout (cursor positions in hex arranged in the order they appear on the display):

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

Off screen: 20..7F (seem to be arranged as additional 16-byte lines, as above).
MY LZ64 uses the following layout (no, it does not seem to make sense, and everybody at Psion I have mentioned this to swears I am totally out of my mind, but I have carefully verified it many many times):

00 01 02 03 08 09 0A 0B 0C 0D 0E 0F 18 19 1A 1B 1C 1D 1E 1F

40 41 42 43 48 49 4A 4B 4C 4D 4E 4F 58 59 5A 5B 5C 5D 5E 5F

04 05 06 07 10 11 12 13 14 15 16 17 20 21 22 23 24 25 26 27

44 45 46 47 50 51 52 53 54 55 56 57 60 61 62 63 64 65 66 67

Off screen: 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 (these positions are only my best guess at the actual
34 35 36 37 38 39 3A 3B 3C 3D 3E 3F cursor positions of the data)
74 75 76 77 78 79 7A 7B 7C 7D 7E 7F

The Psion start-up sequence appears to use initially:

00000110 Cursor position auto increment, no display offset increment

00001100 Display enabled, no underline cursor, no flashing box cursor

00111000 Data port byte mode, two-line enabled, 8-raster characters

Footnote 0 : The Psion Organiser II seems to make no allowances for this feature -- hardware, firmware, or software.

Footnote 1 : The cursor is not moved unless the command indicates it is moved. Some commands shift the cursor (move by one), others move it directly. ALL WRITES/READS to the data port will shift the cursor automatically according to the address index auto-increment/decrement flag.

Footnote 2 : It is possible to have any combination of cursor modes : both off (Psion uses this when no input is desired), underline on (Psion uses this for numeric input), flashing box on (Psion does not use this), flashing box and underline on (Psion uses this for text input).

Footnote 3 : In byte mode, the data port is accessed all at once. In nibble mode (byte mode clear), the data port is accessed high-nibble then low-nibble. Note that BOTH nibbles are in the upper nibble of the port when in nibble mode the way the Psion has the device wired.

Footnote 4 : 2-line on CM and XP. 4-line on LZ. The LZ uses an odd sort of interlaced screen. I have not yet found a quite satisfactory method of addressing it. I am working on it.

Footnote 5 : This can not coexist with 2-line enable on the Organiser II. It appears that the 2-line enable takes priority.

Footnote 6 : Hitachi indicates all `don't care' bits should be left zero for what they call `future compatibility'...

Footnote 7 : Hitachi's documentation is /very/ vague here, but a little experimentation indicates that the auto-update of the LCD offset is in the `right' direction for the cursor movement. If the cursor is set to auto-increment, so is the display. If the cursor is set to auto-decrement, so is the display. This makes it easy to set up a scrolling banner on a standard Hitachi part, but works out to be well nigh useless on the Psion parts.

If you would like to contact me to ask a question, generally chat or comment about this subject, or any other aspect of programming the Psion Organiser II I would be very pleased to speak with you:

Appendix B (Keyboard Coding)

```
KB_NEW:                 ;INSTALL NEW KEYBOARD
      TPA
      PSH   A
      SEI
      BSR   GETP
      XGDX
      OS    UT$CPYB     ;COPY OLD VECTORS TO SAVEVCT
      BSR   GETP
      LDX   #NEWVCT
      OS    UT$CPYB     ;COPY NEW VECTORS IN
      PUL   A
      TAP
      RTS

KB_OLD:                 ;RESTORE OLD KEYBOARD
      TPA
      PSH   A
      SEI
      BSR   GETP
      OS    UT$CPYB     ;RESTORE OLD VECTORS
      PUL   A
      TAP
      RTS

NEWVCT:
      .WORD  NEWPOLL,NEWTRAN,NEWTABL    ;ADDRESSES OF NEW ROUTINES
SAVEVCT:
      .WORD  0,0,0       ;6 BYTES TO SAVE OLD VECTORS IN.

GETP:  LDD    #6
      STD    UTW_S0:
      LDD    #BTA_POLL
      LDX    #SAVEVCT
      RTS

NEWPOLL:

;TURBO CHARGED KEYBOARD POLL ROUTINE
;CHECKS FOR A/C FIRST
;CHECKS FOR ANY KEY QUICKLY, THEN FINDS WHICH KEY WITH DOUBLE COUNTING
;A := KEY PRESSED (0-36, 0=NOKEY)
;KTABPOSN POINTS TO LAST CHAR IN EACH COLUMN IN TURN
;PRESERVES X

      TST    SCA_COUNTERRESET          ;CLEAR COUNTER
      AIM    #<$FF->,KBB_STAT:
      CLR    A                         ;NO KEY YET
      LDA    B,POB_PORT5:
      BPL    NOAC
      LDA    A,#36
RTS5:  RTS

NOAC:  STA    A,KBB_KNUM:    ;TABPOSN:=0
      PSH    A
      BSR    LOOKKEY        ;ZERO IF NO KEY FOUND
      PUL    A
      BEQ    ENDKYBD        ;BRANCH IF NO KEY FOUND
      LDA    B,#$40
      PSH    B
      DEC    B
      BRA    PP

GOCOL:  PSH    B
PP:    PSH    A
1$:    TST    SCA_COUNTERCLOCK       ;CLOCK COUNTER B TIMES
      DEC    B
      BNE    1$
      BSR    LOOKKEY        ;B:= OFFSET IN COL (IF KEY FOUND)
      PUL    A
      BEQ    NOPSH          ;BRANCH IF NO KEY IN THIS COL
      ADD    B,KBB_KNUM:
      TBA                   ;GOT KEY BUT MUST CHECK IF SHIFT
NOPSH:  LDA    B,KBB_KNUM:
      ADD    B,#5
      STA    B,KBB_KNUM:    ;POINT TO NEXT COL
      PUL    B
      LSR    B              ;SELECT NEXT COL
      BNE    GOCOL
ENDKYBD:
      RTS
```

```
LOOKKEY:
;B:=KEY OFFSET IN COL (1-5)
;B:=0 IF NOKEY IN THIS COL (OR JUST SHIFT)
;SETS KY_SHFT FLAG IF SHIFT PRESSED
;SETS KY_SHFT2 IF  PRESSED
;SETS KY_SHFT3 IF  PRESSED
;SETS Z ON B
;PRESERVES X

      LDA    B,#5
      LDA    A,POB_PORT5:
      ASL    A
NEXTROW:
      ASL    A            ;ROTATE KEY INTO CARRY
      BCC    GOTKEY       ;CARRY CLEAR IF KEY PRESSED
      DEC    B
      BNE    NEXTROW
      RTS

GOTKEY:                   ;B IS POSITION IN COL
      TST    KBB_SHFK     ;CLR IF SHIFT ENABLED
      BNE    13$          ;B IS NOT ZERO
      PSH    A
      LDA    A,KBB_KNUM:
      CMP    A,#25        ;IS IT COL 6 (COL CONTAINING SHIFT) ?
      BNE    11$          ;B IS NOT ZERO
      PUL    A
      DEC    B
      BEQ    10$          ;BRANCH IF B WAS 1 IE SHIFT

      PSH    B            ;ELSE CHECK IF SHIFT PRESSED AS WELL
1$:   ASL    A
      DEC    B
      BNE    1$
      PUL    B
      INC    B          ;RESTORE B
      BCS    13$          ;BRANCH IF C SET FROM ASL A

10$:  OIM    #KY_SHFT,KBB_STAT:
      TST    B
      RTS

11$:  CMP    A,#30        ;COL CONTAINING ,  ARROWS ?
      PUL    A
      BNE    13$          ;B IS NON-ZERO
      CMP    B,#4         ;LEFT ?
      BEQ    12$
      CMP    B,#5         ;RIGHT ?
      BNE    13$
      OIM    #KY_SHFT3,KBB_STAT:
      CLR    B            ;SET Z
      RTS
12$:
      OIM    #KY_SHFT2,KBB_STAT:
      CLR    B            ;SET Z
13$:  RTS

NEWTRAN:
;TRANSLATE KEY NUMBER IN A (0-36) INTO ASCII KEY USING KBB_STAT
;SETS DPB_CUST APPROPRIATELY + REFRESHES CURSOR STATE
;SETS Z=1 IF A=0 (NOKEY)
;SETS KY_CPNM IF CAP OR NUM

      AIM    #<$FF-KY_CPNM>,KBB_STAT:
      TAB
      TST    KBB_STAT
      BPL    2$                 ;BRANCH IF CAN'T BE CAP OR NUM
      CMP    A,KBB_CAPK         ; - CAPS KEY
      BNE    1$
      EIM    #KY_CAPS,KBB_STAT:
      OIM    #KY_CPNM,KBB_STAT:
      CLR    B
1$:   CMP    A,KBB_NUMK         ; - NUM KEY
      BNE    2$
      EIM    #KY_NUMB,KBB_STAT:
      OIM    #KY_CPNM,KBB_STAT:
      CLR    B
2$:   BSR    KCUST              ;FIX CURSOR STATE FOR KBB_STAT
                                ;RETURNS X POINTING TO APPROPRIATE
                                ;KEYBOARD TABLE !
                                ;DISPLAYS CURSOR TYPE. PRESERVES B
      TST    B
      BEQ    9$
      ABX
      LDA    B,0,X              ;GET ASCII KEY FROM TABLE
```



```
      LDA    A,KBB_STAT:
      LSR    A
      BCS    4$                 ;IF SET LEAVE AS LOWER CASE
      CMP    B,#^A/A/
      BCS    21$
      CMP    B,#^A/Z/
      BHI    21$
      ADD    B,#^A/a/-^A/A/     ;CONVERT TO UPPER CASE
21$:
4$:   CMP    B,#K_DEL
      BNE    9$
      LDA    A,KBB_STAT:
      BPL    9$
      LDA    B,#$7F             ;DEL
9$:   TBA
      RTS

KCUST:
      LDX    BTA_TABL           ;ADDRESS OF KEYBOARD TABLE
      DEX
      LDA    A,KBB_STAT:
      ASL    A
      ASL    A
      BMI    DO2
      ASL    A
      BMI    DO3
      LDA    A,KBB_STAT:
      ASL    A
      BVS    DOSHFT

      AIM    #<$FF-CURSOR_LINE>,DPB_CUST:   ;BLOCK CURSOR
      BRA    DP_CTYP            ;REFRESH CURSOR TYPE ON LCD

DO3:  BSR    ADD36
DO2:  BSR    ADD36
DOSHFT: OIM    #CURSOR_LINE,DPB_CUST:  ;LINE CURSOR FOR SHIFTED CHARS
      BSR    ADD36

DP_CTYP:
      LDA    A,#$0C
      TIM    #CURS_ON,DPB_CUST:
      BEQ    1$                 ;BRANCH IF CURSOR OFF
      ORA    A,#$02
      TIM    #CURSOR_LINE,DPB_CUST:
      BNE    1$                 ;BRANCH IF CURSOR IN LINE FORM
      ORA    A,#$01
1$:   TST    SCA_LCDCONTROL
      BMI    1$
      STA    A,SCA_LCDCONTROL
      RTS

ADD36:
      PSH    B
      LDA    B,#36
      ABX
      PUL    B
      RTS

NEWTABL:
      .ASCII  /ZVPJD/
      .BYTE   K_EXE
      .ASCII  /XRLF/
      .ASCII  / WQKE/
      .ASCII  /YUOIC/
      .BYTE   K_DEL
      .ASCII  /TNHB/
      .BYTE   0
      .ASCII  /SMGA/
      .BYTE   $81
      .BYTE   $82,$83
      .BYTE   K_LEFT,K_RGHT
      .BYTE   $80

SHFT1:
      .ASCII  /.258)/
      .BYTE   K_EXE
              $
      .ASCII  $+-$
      .ASCII  $/$
      .ASCII  / 369%/
      .ASCII  /0147(/
      .BYTE   K_DEL
      .ASCII  /:$">/
      .BYTE   0
      .ASCII  /;,=
```

Appendix C

Using the operating system.
List of services.
List of errors File types and Extensions
Control Codes

Using the operating system.

The Psion organiser has many features built into its ROM, and some of these can be called from machine code using the SWI instruction. The SWI instruction is followed by a byte indicating which of the system services you want. The SWI instruction calls a routine in the ROM which performs the task required, and after that has finished the program execution continues at the instruction which follows immediately after the service byte.

Most services need some parameters. Mostly the registers (A,B and X) will contain the information required by the service, but sometimes further information is needed which is often put in the scratch registers UTW_S0 and UTW_S1, at address $41/42 and $43/44 respectively. Sometimes the runtime buffer RTT_BF ($2188-$2287 with length byte RTT_BL $2187) is needed too. Information is passed back from the system services in the same way. Unless otherwise noted, assume the contents of the registers A,B,X and the scratch registers UTW_S0-5 are trashed by the system services.

Some services can not always perform their tasks, and an error can be returned. A service shows an error has occurred by returning with the carry flag set, and with B containing the error code. The error codes are the same as those used in OPL. Some services that can never return an error may not always clear the carry flag, so the carry flag should only be tested after services that can cause errors.

The scratch registers UTW_S0 to UTW_S5 (which lie between $41 and $4C) are used by most services, so any values stored there will most likely be trashed. You can freely use these addresses yourself as long as you keep in mind that the information stored there is lost by calling the operating system. See also BT$PPRG if more scratch variables are needed.

Note that strings are usually leading byte-count strings, which will from now on be abbreviated by 'lbc string'. An lbc string is a string of (ascii) characters which is preceded by a length byte. Thus the word 'hello' would be stored in lbc form as 05,68,65,6C,6C,6F in hex. The address of an lbc string is the address of the length byte, not the address of the first character.

List of services.

Here is an alphabetic list of all the operating system services. Each service has a short name. This allows us to use a slightly more descriptive way of denoting the SWI instruction. Instead of SWI 0 for example, we now write OS AL$FREE instead. A very short description is also given. Those marked with an asterisk are available on the LZ only, and those with a plus sign only on the LZ and later versions of the CM/XP.

AL$FREE	000	00	Free an allocator cell
AL$GRAB	001	01	Grab an allocator cell
AL$GROW	002	02	Grow an allocator cell
AL$REPL	003	03	Change cell size for a part replacement
AL$SHNK	004	04	Shrink an allocator cell
AL$SIZE	005	05	Get size of an allocator cell
AL$ZERO	006	06	Clear an allocator cell
AM$ENTR	158	9E*	Alarm entry
BT$NMDN	007	07	Disable NMI, makes system clock invalid.
BT$NMEN	008	08	Enable NMI which were switched off by BT$NMDN
BT$NOF	009	09	Disable NMI, keeping track of elapsed time for the system clock
BT$NON	010	0A	Enables NMI which were switched off by BT$NOF, corrects clock
BT$PPRG	011	0B	Push or Pop system variables UTW_R0 to UTW_R6
BT$SWOF	012	0C	Switches off the Psion, like menu option OFF
BT$TOFF	129	81*	Temporary switch off
BZ$ALRM	013	0D	Sound the alarm once
BZ$BELL	014	0E	Emits a standard beep
BZ$TONE	015	0F	Emits a beep with specified length & pitch
CA$ENTR	160	A0*	Calculator entry
DI$ENTR	167	A7*	Enter diary
DP$CPRN	132	84*	Redefine UDG-clock
DP$CSET	131	83*	Set UDG-clock status
DP$EMIT	016	10	Output single character to the display
DP$MSET	130	82*	Set display mode
DP$PRNT	017	11	Display a string
DP$PVEW	163	A3*	Partial view string
DP$REST	018	12	Restore screen to previously saved state
DP$SAVE	019	13	Save screen state (use DP$REST to restore)
DP$STAT	020	14	Move cursor and change its status
DP$UDG	133	85*	Read/write UDG
DP$VIEW	021	15	Display a string, like the OPL VIEW instruction
DP$WRDY	022	16	Wait until system variable DPW_REDY is zero
DV$BOOT	023	17	Boot all devices
DV$CLER	024	18	Unboot all devices
DV$LKUP	025	19	Language lookup (checks each device)
DV$LOAD	026	1A	Load relocatable code from a pack
DV$VECT	027	1B	Find and run device code
ED$EDIT	028	1C	Calls the line editor ED$EPOS, with cursor at beginning.
ED$EPOS	029	1D	The built-in editor
ED$VIEW	030	1E	Clears screen, displays a string, like OPL command DISP
ER$LKUP	031	1F	Look up an error string, given an error number
ER$MESS	032	20	Displays an error message given an error number
ER$PRNT	169	A9*	Display string as error message
FL$BACK	033	21	Works like OPL command BACK in ordinary files
FL$BCAT	034	22	Like OPL function DIR$, works on all file types
FL$BDEL	035	23	Deletes a block file with type $82 to $8F
FL$BOPN	036	24	Opens a block file with type $82 to $8F
FL$BSAV	037	25	Saves a block file with specified type
FL$CATL	038	26	Like OPL function DIR$, works on ordinary files
FL$COPY	039	27	Copy ordinary or block files (device to device)
FL$CRET	040	28	Creates an ordinary file, like OPL command CREATE
FL$DELN	041	29	Deletes an ordinary file, like OPL command DELETE
FL$ERAS	042	2A	Erases current record, like OPL command ERASE
FL$FDEL	176	B0*	Fast delete records
FL$FFND	043	2B	Find record with given start string on current pack
FL$FIND	044	2C	Find record containing given string on current pack
FL$FREC	045	2D	Returns info on given record on current pack
FL$GETX	152	98*	Convert file extension to type
FL$NCAT	157	9D*	Catalogue Nth file with wild card
FL$NEXT	046	2E	Move to next record, like OPL command NEXT
FL$OPEN	047	2F	Open an existing ordinary file
FL$PARS	048	30	Check given filename is legal
FL$READ	049	31	Read current record
FL$RECT	050	32	Set current record type
FL$RENM	051	33	Renames an ordinary file
FL$RSET	052	34	Make given record current,like OPL command POSITION
FL$SETP	053	35	Select a datapack
FL$SIZE	054	36	Returns free space on current datapack
FL$VALX	153	99*	Convert file type to extension
FL$WCAT	144	90*	Catalogue files with wild card
FL$WCPY	145	91*	Copy files with wild card
FL$WDEL	146	92*	Delete files with wild card
FL$WFND	147	93*	Find file record with given text with wild card
FL$WPAR	143	8F*	Check wild card filename
FL$WRIT	055	37	Appends new record to current device, like APPEND
FN$ACOS	173	AD*	Finds arccosine of a floating point number
FN$ASIN	172	AC*	Finds arcsine of a floating point number
FN$ATAN	056	38	Finds arctangent of a floating point number
FN$COS	057	39	Finds cosine of a floating point number
FN$EXP	058	3A	Finds "e to power of a floating point number"
FN$LN	059	3B	Find natural logarithm of floating point number

FN$LOG	060	3C	Finds logarithm of a floating point number
FN$MAX	142	8E*	Maximum of list of numbers
FN$MEAN	138	8A*	Mean of list of numbers
FN$MIN	141	8D*	Minimum of list of numbers
FN$POWR	061	3D	Finds x**y, where x & y are floating point nums
FN$RND	062	3E	Finds a random number between 0 and 1
FN$SIN	063	3F	Finds sine of a floating point number
FN$SQRT	064	40	Finds square root of a floating point number
FN$STD	140	8C*	Standard deviation of list of numbers
FN$SUM	137	89*	Sum of list of numbers
FN$TAN	065	41	Finds tangent of a floating point number
FN$VAR	139	8B*	Variance of list of numbers
IT$GVAL	066	42	Used to get value of a byte parameter in table interpreter routine
IT$RADD	067	43	Used to get memory address of a variable param in table interpreter routine
IT$STRT	068	44	The table interpreter
IT$TADD	069	45	Used to get value of a word parameter in table interpreter routine
KB$BREK	070	46	Test if ON/CLEAR is pressed
KB$CONK	177	B1*	Cursor on and get key
KB$FLSH	071	47	Flush the keyboard buffer
KB$GETK	072	48	Wait for a single character from keyboard
KB$INIT	073	49	Initialises keyboard (OCI) interrupts
KB$STAT	074	4A	Set the keyboard state, similar to OPL KSTAT
KB$TEST	075	4B	Test keyboard buffer, return ASCII key if found
KB$UGET	076	4C	Put ASCII character in the 'unget' buffer
LG$EDIT	171	AB*	Enter text editor
LG$ENTR	175	AF*	Enter prog
LG$NEWP	077	4D	Makes a new text block file
LG$RLED	078	4E	Perform RUN/LIST/EDIT/DELETE on existing file
LN$STRT	079	4F	Runs the translator
LN$XSTT	154	9A*	LZ translate procedure or calc expression
MN$1DSP	135	87*	One line menu
MN$DISP	080	50	Displays a menu, like OPL function MENU
MN$TITL	136	88*	Menu with icon and clock
MN$XDSP	134	86*	Titled menu
MT$BTOF	081	51	Converts ASCII string to floating point number
MT$FADD	082	52	Adds two floating point numbers
MT$FBDC	083	53	Convert floating point to ASCII in fixed decimal format, like FIX$
MT$FBEX	084	54	Convert floating point to ASCII in exponent format, like SCI$
MT$FBGN	085	55	Convert floating point to ASCII in general format, like GEN$
MT$FBIN	086	56	Convert floating point to ASCII in integer format, like NUM$
MT$FDIV	087	57	Divides two floating point numbers
MT$FMUL	088	58	Multiplies two floating point numbers
MT$FNGT	089	59	Negates a floating point number
MT$FSUB	090	5A	Subtracts two floating point numbers
NT$ENTR	159	9F*	Notepad entry
PK$PKOF	091	5B	Turns off all packs
PK$QADD	092	5C	Returns current pack address
PK$RBYT	093	5D	Reads a byte from current position in current pack
PK$READ	094	5E	Copies bytes from current position in current pack
PK$RWRD	095	5F	Reads a word from current position in current pack
PK$SADD	096	60	Sets current pack address
PK$SAVE	097	61	Copies bytes to current position in current pack
PK$SETP	098	62	Sets current pack
PK$SKIP	099	63	Skips bytes by updating current pack address
RM$RUNP	100	64	Loads/runs an OPL procedure or runs the calc
TI$ENTR	164	A4*	Enter time
TL$ADDI	101	65	Add an item to the top menu

TL$CPYX	102	66	Performs the COPY function from standard menu
TL$DELI	103	67	Delete item from the top menu
TL$LSET	128	80+	Selects the language for menus. MULTI-LINGUAL MACHINES ONLY.
TL$RSTR	127	7F+	Restores several top-level menu entries. VERSION 2.7+ ONLY.
TL$XXMD	104	68	Clear screen, show prompt, enter filename
TL$ZZMD	155	9B*	Edit filename, screen not pre-cleared
TM$DAYV	105	69	Calculate day of the week from a given date
TM$DNAM	151	97*	Get weekday name
TM$MNAM	170	AA*	Get month name
TM$NDYS	149	95*	Get days since 1/1/1900
TM$TGET	106	6A	Copies system time to a given buffer
TM$TSET	179	B3*	Set system time
TM$UPDT	107	6B	Add a value to date/time held in a buffer
TM$WAIT	108	6C	Wait for a number of clock ticks
TM$WEEK	150	96*	Get week number
UT$CDSP	126	7E+	Like UT$DISP but screen is cleared first. VERSION 2.5+ ONLY.
UT$CMPB	178	B2*	Compare buffer, case dependent
UT$CPYB	109	6D	Copy bytes from one place to another in RAM
UT$DDSP	110	6E	Display a format control string from given addr
UT$DISP	111	6F	Display an 'inline' format control string
UT$ENTR	112	70	Calls a routine at given address
UT$FILL	113	71	Fills a number of bytes with a given value
UT$ICPB	114	72	Compare two ASCII strings, ignoring case
UT$ISBF	115	73	Find a string within another, like OPL function LOC
UT$LEAV	116	74	Used to exit from a routine entered via UT$ENTR
UT$SDIV	117	75	Divides signed integers
UT$SMUL	118	76	Multiplies signed integers
UT$SORT	165	A5*	General sort utility
UT$SPLT	119	77	Finds address/length of a field in a string
UT$UDIV	120	78	Divides unsigned integers
UT$UMUL	121	79	Multiplies unsigned integers
UT$UTOB	122	7A	Convert unsigned integer to ASCII decimal
UT$WILD	148	94*	Search string for substring with wild cards
UT$XCAT	123	7B	Display all filenames of given type on current pack
UT$XTOB	124	7C	Convert unsigned integer to ASCII hex format
UT$YSNO	125	7D	Wait for Y,y,N,n or ON/CLEAR to be pressed
WL$ENTR	166	A6*	Enter world
XF$ENTR	168	A8*	Enter Xfiles
XF$SORT	174	AE*	Sort file
XT$BAR	162	A2*	Create UDG bar graph
XT$DIRM	156	9C*	Do directory on screen
XT$ENTR	161	A1*	Do utility application

List of Errors

Allocator errors

255	FF	NO ALLOC CELLS
254	FE	OUT OF MEMORY

Calculator errors

253	FD	EXPONENT RANGE
252	FC	STR TO NUM ERR
251	FB	DIVIDE BY ZERO
250	FA	NUM TO STR ERR
249	F9	STACK OVERFLOW
248	F8	STACK UNDERFLOW
247	F7	FN ARGUMENT ERR

Pack errors

246	F6	NO PACK

245 F5 WRITE PACK ERR
244 F4 READ ONLY PACK
243 F3 BAD DEVICE NAME
242 F2 PACK CHANGED
241 F1 PACK NOT BLANK
240 F0 UNKNOWN PACK

File system errors

239 EF PACK FULL
238 EE END OF FILE
237 ED BAD RECORD TYPE
236 EC BAD FILE NAME
235 EB FILE EXISTS
234 EA FILE NOT FOUND
233 E9 DIRECTORY FULL
232 E8 PAK NOT COPYABLE

Device system errors

231 E7 BAD DEVICE CALL
230 E6 DEVICE MISSING
229 E5 DEVICE LOAD ERR

Translator errors

228 E4 SYNTAX ERR
227 E3 MISMATCHED ()'s
226 E2 BAD FN ARGS
225 E1 SUBSCRIPT ERR
224 E0 TYPE MISMATCH
223 DF NAME TOO LONG
222 DE BAD IDENTIFIER
221 DD MISMATCHED "
220 DC STRING TOO LONG
219 DB BAD CHARACTER
218 DA BAD NUMBER
217 D9 NO PROC NAME
216 D8 BAD DECLARATION
215 D7 BAD ARRAY SIZE
214 D6 DUPLICATE NAME
213 D5 STRUCTURE ERR
212 D4 TOO COMPLEX
211 D3 MISSING LABEL
210 D2 MISSING COMMA
209 D1 BAD LOGICAL NAME
208 D0 BAD ASSIGNMENT
207 CF BAD FIELD LIST

Runtime errors

206 CE ESCAPE
205 CD ARG COUNT ERR
204 CC MISSING EXTERNAL
203 CB MISSING PROC
202 CA MENU TOO BIG
201 C9 FIELD MISMATCH
200 C8 READ PACK ERR
199 C7 FILE IN USE
198 C6 RECORD TOO BIG
197 C5 BAD PROC NAME
196 C4 FILE NOT OPEN
195 C3 INTEGER OVERFLOW

General errors

194 C2 BATTERY TOO LOW

193 C1 DEVICE READ ERR
192 C0 DEVICE WRITE ERR
191 BF Not used. ER$LKUP uses it as a marker for the end of the error table in ROM

External errors

These are not used in the ROM, so they have no standard error messages. Peripherals can cause these errors.

General device errors

190 BE Bad Parameter

Comms link errors

189 BD File Not Found
188 BC Server Error
187 BB File Already Exist
186 BA Disk Full
185 B9 Record Too Long

Printer errors

184 B8 Printer Battery Low
183 B7 Printer Time Out
182 B6 Printer Escape

File types and Extensions

Here are the various file type used by the Psion, and the extensions used in filenames on the LZ. Note that these are not the extensions used by the comms link for PC filenames. The comms link uses only ODB, OPL (text only procedure) and OBx where x is in the range 2-F.

81		ODB	Filenames of ordinary data files, including LZ diary files
82		DIA	CM/XP diary files
83	00	OPL	OPL procedures, both opl source and object code
	01	OPO	OPL procedures, object code only.
	02	OPT	OPL procedures, opl source only.
84		COM	Comms-link setup files
85		PLN	Spreadsheet files
86		PAG	Pager files
87		NTS	LZ Notepad files
88		TY8	Not used
89		TY9	Not used
8A		TYA	Not used
8B		TYB	Not used
8C		TYC	Not used
8D		TYD	Not used
8E		TYE	Not used
8F		TYF	Not used

The file type 82-8F are block files, which means that the record containing the filename is immediately followed by a single data block (which internally uses file type 80). Records of type 81 are used to store the names of ordinary data files but no further data other than the file type of its data records which lies in the range 90-FE. Since each ordinary file must have a unique data record type in that range, there can be only 111 ordinary files on a pack. The MAIN file is always assigned type 90 for its data. Note that the filetypes 00-7F are used to signify deleted records on a pack, and that FF is used for error correcting purposes.

Control codes

The following character codes will, when printed, not produce a character but control the screen or cursor in some way.

0-7	00-07	Prints a UDG character
8	08	Moves the cursor left 1 character
9	09	Moves the cursor to next position modulo 8 (or 10 on an LZ)
10	0A	Moves the cursor down one line
11	0B	Moves the cursor to top left of the screen
12	0C	Clears the screen
13	0D	Moves the cursor to left of the current line
14	0E	Clears the 1st line, and cursor is placed on the left of that line.
15	0F	Clears the 2nd line, and cursor is placed on the left of that line.
16	10	Emits a standard beep (see also BZ$BELL)
17	11	Refreshes the 1st and 2nd line of the display from the buffer
18	12	Refreshes the 1st line of the display from the buffer
19	13	Refreshes the 2nd line of the display from the buffer

On the LZ the following are also available.

20	14	Refreshes the 3rd line of the display from the buffer
21	15	Refreshes the 4th line of the display from the buffer
22	16	Clears the 3rd line, and cursor is placed on the left of that line.
23	17	Clears the 4th line, and cursor is placed on the left of that line.
24	18	Puts dots on line 2, clears line 1, and moves cursor to left of 1st line.
25	19	Puts dots on line 3, clears line 4, and moves cursor to left of 4th line.
26	1A	Clears till the end of the current line.