THE PSION SIBO HARDWARE DEVELOPMENT KIT

Version 1.00

May 26 1995

Contents

1. INTRODUCTION

This document is intended to provide guidance to anyone wishing to construct peripherals for the Psion SIBO (**si**xteen **b**it **o**rganiser) range of computers. It describes in detail all aspects of Psion peripheral hardware development and the structure of the software required to drive such peripherals. It is the aim of this document to aid third party development engineers in producing production ready peripherals for any of the following Psion products: Series 3/3a, Workabout, HC and HCDOS. Mechanical and plastic moulding information and information on how to develop production test equipment is therefore also included. The emphasis throughout is on the two key Psion peripheral chips ASIC4 and ASIC5. Detailed information regarding their functionality is provided. The structure of Psion hardware device drivers is examined both in general outline and then with regard to two specific examples whose source code is provided in the appendix to this document. It has been assumed that the reader has some knowledge of a Psion computer such as the Series 3/3a and an understanding of how such a machine is programmed. A good understanding of electronics, the C programming language and 8086 assembler is also assumed.

Due to the continuous nature of development, information in this manual may change without notice. Developers are advised to contact Psion Support to confirm critical details prior to committing products to manufacture.

2. SYSTEM OVERVIEW

All present Psion computers are based around the proprietary SIBO architecture. A SIBO machine is a battery-powered, 8086-based, computer system. SIBO stands for SIxteen Bit Organiser. The architecture has been designed with the size, weight and power consumption of computers designed for the portable environment in mind. The key components of the SIBO architecture are:

- An 8086 class processor .
- A sophisticated power management system that selectively powers subsystems under software control.
- A synchronous, high speed, serial protocol (the Psion SIBO serial interface) for communication between a machine and its peripherals.
- Solid State Disks (SSDs) that provide fast, low-power, silicon-based mass storage with no moving parts.
- Hardware protection of the system from aberrant software processes (trapping of out of range addressing and a watch-dog timer on interrupts being disabled).
- Real-time clock.
- ROM-resident system software.
- Graphics LCD display.
- A touch sensitive digitising pad that provides a pointing device (only available on some models).
- ISDN-8bit standard combo sound system (only available on some models).

The SIBO architecture has primarily been implemented in custom ICs called ASICs. At the time of writing there are ten different SIBO ASICs. Some of these ASICs have been designed for use inside peripherals and these will discussed in detail throughout this document. All SIBO ASICs have been implemented in surface mount packages and are based on a static CMOS technology. Current SIBO products in the MC, HC and Series 3 range are based on the same three principal chips. These are the V30H (an 8086-compatible processor) and two Psion custom chips known as ASIC1 and ASIC2. Later SIBO products including the Series 3a and Workabout have these three devices integrated into a single Psion custom chip known as ASIC9. The V30H is an enhanced 16-bit CMOS version of the 8088 found in the original IBM PC. It is software compatible with the 8088. The V30H is a fully static design which means that all the internal storage elements (i.e. its registers) are made from static rather than dynamic storage components. This in turn means that there is no minimum clock speed required to refresh the storage elements and the system clock can be stopped at any time with no loss of internal state. This technique is used extensively in the SIBO architecture to save power while the processor is idle (i.e. waiting for an event).

The Psion SIBO serial protocol is a proprietary synchronous two wire serial standard by which host Psion handhelds communicate with external devices. These devices will typically be Memory Packs (usually referred to as Solid State Disks or SSDs), RS232 and Centronics printer interfaces, fax modems, bar-code scanners, and so on. The SIBO architecture provides for two basic forms of expansion device, namely the extended internal expansion connection (as with SSDs) and the reduced external expansion connection (the 6-pin S3a serial port or the 11-pin LIF connector). The MC and HC range of computers have two SSD ports and two separate independent single row 25-way extended internal expansion ports. These ports have in addition to a Psion SIBO Channel, direct, parallel I/O from the processor. Direct connection to these machines 7.2 volt battery is included to support high power peripherals such as Printers and Barcode readers. The Series 3 range of computers have two ports for SSDs and a single, reduced, 6-pin expansion port, which provides only a Psion SIBO serial channel and limited power (<25mA). The Psion Workabout has two SSD ports, two internal expansion points, and one external expansion port. The single external expansion port uses an 11-pin Low Insertion Force (LIF) socket which provides a Psion SIBO channel, 25mA of current and additional lines required for detecting the presence of the Workabout cradle. Each internal expansion port consists of a single row 26-way connector carrying two high speed serial ports.

A range of Psion peripherals have been produced for connection to the Psion handhelds outlined above. These peripherals currently incorporate one of two custom integrated circuits (ASIC4 and ASIC5) that convert SIBO serial protocol signals to data bus TTL level voltages which enable memory and memory-mapped peripherals to be addressed. ASIC4 is used in SSDs and for memory-mapped peripherals. A typical ASIC4 peripheral for a Psion S3a would consist of an ASIC4 connected to port C of the host machine and a peripheral chip/device mapped into ASIC4's addressing space. ASIC5 is a general purpose I/O chip with a UART on board that can be run in several different modes. For example ASIC5 can be used for MCRs (magnetic card readers) or Centronics interfaces thereby simplifying peripheral design. Psion extended internal expansion ports carries an active low interrupt input line to the host controller circuitry. The reduced external expansion ports has an active high interrupt input line. The function of the interrupt can thus be programmed into the host machine's ASIC1 or ASIC9.

The low-level programming interface to a Psion handheld peripheral is encapsulated within an appropriate device driver. Psion device drivers are written in 8086 assembler and follow a prescribed pattern outlined later in this document. The construction of a peripheral and the coding of its complimentary device driver enable the developer to access its functionality through the means of library calls in a C program. Examples of such calls are `p_loadldd()`, `p_open()` and `p_close()`. I/O requests are routed through the device driver's strategy vector which maps to the PLIB `p_iow()` call. The device driver is built using the Borland Turbo Assembler and resides in a single code segment. The device driver can be stored in either RAM or a ROM on board the peripheral or can be supplied on an SSD (solid state disk).

3. HARDWARE OVERVIEW

The Psion SIBO serial protocol

The Psion SIBO serial protocol is a general purpose method of bi-directional serial data transfer. It has been designed for synchronous communication between a host controlling device and a number of slave devices. On a hardware level, the SIBO serial protocol is implemented through Psion ASICs. The controlling device must contain an ASIC2 (or ASIC9) and the slave devices an ASIC4 or ASIC5. The various Psion ASICs are described in more detail below.
The synchronous SIBO serial protocol interface consists of 2 wires:
CLK - A clock output from the controller to the slaves. Nominally 3.84 MHz.
DATA - A bi-directional synchronous data line.
The data is transferred using a series of 12 bit frames including 8 data bits each. This equates to a theoretical maximum data transfer rate of approximately 312 Kbytes/second. Other bits of the frame contain control information. The "system" is generically defined by 2 protocol layers, namely the Physical layer and the Transport layer. These layers are described in detail in the next chapter.

Psion ASICs and what they do

ASIC stands for **A**pplication **S**pecific **I**ntegrated **C**ircuit and as previously indicated, these devices are widely used within Psion hardware. Summaries of the functionality of each ASIC that is relevant to peripheral development are presented below:

ASIC1: ASIC1 is the main system controller chip for the SIBO architecture. It connects directly to the 8086-based processor (i.e. the V30H) controlling all bus cycles to and from the processor. This configuration effectively forms a micro-controller like device that executes 8086 instruction codes. ASIC 1 is made up of a number of functional blocks including a bus controller, a programmable timer, an eight input interrupt controller, an LCD controller and the memory decoding circuitry.

ASIC2: ASIC2 is the peripheral controller chip for the SIBO architecture. It contains the system clock oscillator and controls switching between the standby and operating states. ASIC2 provides an interface to the power supply, keyboard, buzzer and SSDs. ASIC 2 includes the eight-channel SIBO serial protocol controller and provides interface circuitry to both the reduced external and extended internal peripheral expansion ports.

ASIC4: ASIC4 is a serial protocol slave IC for addressing memory and general memory-mapped peripherals. It is used in SSDs to convert SIBO serial protocol signals into addresses within the memory range of the memory pack. ASIC4 was designed to be a cut-down version of ASIC5 which was the original SIBO serial protocol slave chip.

ASIC5: ASIC5 is a general purpose I/O chip with a built-in UART that can be set to run in a number of different modes thereby simplifying the task of peripheral design. For example, it is possible to set up ASIC5 to run as a Centronics parallel port interface, an 8-bit parallel I/O port, a serial bar code controller or a serial RS232 converter.

ASIC9: ASIC9 is a composite chip comprising of a V30H processor, ASIC1, ASIC2 and general I/O and PSU control logic all on one IC. ASIC9 thus integrates all the digital logic required to produce a SIBO architecture computer less the memory onto one chip. ASIC9 has a few additional features such as an extra free-running clock (FRC) and a codec interface for sound.

Interrupts

Psion peripherals usually incorporate some circuitry to generate hardware interrupts. Both reduced external expansion ports (such as the LIF connector on the Workabout or the 6-pin serial port connector on the S3a) and extended internal expansion ports (such as the two single row 25-way HC connectors) carry an interrupt line. This is an active high input to the host machine's interrupt controller circuitry which resides on the logical equivalent of ASIC1. The OS intercepts all interrupts and can be requested to call a particular function within a controlling device driver. Eight hardware interrupts are supported by SIBO hardware. IRQ0 is the highest priority and IRQ7 the lowest. All interrupts are level triggered and must be serviced in the following order:

- Device asserts the appropriate interrupt request line.
- The interrupt controller unit within either ASIC2 or ASIC9 places onto the data bus the vector of the highest priority device with an interrupt pending. This enables the CPU to jump to the correct interrupt service routine code.
- During the interrupt service routine, the software clears the interrupt line by some action specific to the device.
- The interrupt service routine then informs the interrupt controller that the interrupt has been cleared by writing to the non-specific end of interrupt (NSEOI) location.
- If another interrupt is pending then go back to the second step.

With 8086-based processors, it is not possible to have nested interrupts.

The current range of Psion peripherals

There are currently a number of Psion peripherals in use and some of the key ones are outlined below in order to provide the developer with a feel for peripheral design issues:

SSDs: Solid State Disks (or Memory Packs) use a built-in ASIC4 to decode SIBO serial protocol signals into memory addresses within the memory range of the SSD.

Psion 3-link: The 3-link translates the high speed SIBO serial protocol channel on the S3a 6-pin reduced external expansion socket into a serial RS232 format. This enables the host machine to communicate with a PC for example by means of connecting the 3-link unit from the handheld's 6-pin port to the PC's COM1 or COM2 port. The 3-link contains an ASIC5 which uses its on-board UART to convert SIBO serial protocol signals to RS232 format TTL level voltages.

The HC Printer: The HC Printer translates SIBO serial protocol signals transmitted across the single row 25-way extended internal expansion socket of the host HC into a parallel 8-bit format that is compatible with the universal Centronics printer interface standard. The HC Printer unit contains an ASIC5 running in Centronics interface mode which acts as the serial protocol slave and requires a small number of support chips.

Psion 3-Fax: The 3-Fax contains an ASIC4 and a memory-mapped modem chip set which permits the host machine to transmit (but not receive) fax messages.

Barcode: The Psion Barcode reader employs an ASIC5 running in serial mode to read the data received from the barcode decoder chip into a SIBO serial protocol format that can be transmitted to the host ASIC2/ASIC9.

Workabout RS232 Interface: This peripheral connects to the single row 26-way extended internal expansion port of the Workabout. It incorporates an ASIC5 running in its default mode to translate SIBO serial protocol signals into a TTL level (+/-5v) serial RS232 format using ASIC5's on-board UART. The TTL level RS232 signals are converted into the standard EIA format (+/-12v) before coming out on the conventional RS232 9-pin D-type connector.

4. THE PSION SIBO SERIAL PROTOCOL

Introduction

The Psion SIBO serial protocol is a proprietary standard for bi-directional serial data transfer between a controlling device and a number of slave devices. The synchronous interface consists of 2 wires CLK and DATA as mentioned earlier:
CLK - A clock output from the controller to the slaves. Nominally 3.84 Mhz for memory interfaces or 1.536.Mhz continuous for peripherals.
DATA - A bi-directional synchronous data line.
The data is transferred using a series of 12 bit frames including 8 data bits each. This equates to a theoretical maximum data transfer rate of approximately 312 Kbytes/second. Other bits of the frame contain control information. The "system" is generically defined by 2 protocol layers:-

- The Physical layer defining the hardware interface and frame structure.
- The Transport layer defines system control and register transfers between the controller and the slaves.

Using this system, a large number of higher level implementations can be defined. In normal use the controller will communicate to slaves in a point to point configuration. Multidrop configurations with a number of slaves attached to one channel of the controller are also supported.

As indicated in the previous chapter, the SIBO serial protocol controller circuitry resides in either an ASIC2 or an ASIC9 depending on the particular Psion hardware platform. The S3a and Workabout employ ASIC9 whereas the HC, MC and S3 use ASIC2.

Hardware Interface

As indicated above, the SIBO serial protocol consists of two lines that switch at 5V CMOS voltage levels:

Clock Line

This line is used to synchronously clock data between the controller and slaves. It is always output from the controller circuitry that resides in ASIC2/ASIC9. The clock should only be active during the transfer of data or when the serial channel is continuous clocking mode (used by ASIC5). At all other times it is tri-state pulled low.

Clock Timing Parameters

Symbol	Parameter	Min	Typ	Max	Units
Tckh	Width of Clock High	65	130	-	nSec
Tckl	Width of Clock Low	130	130	-	nSec
Tcyc	Cycle time of clock	195	260	-	nSec
Fck	Clock Frequency		3.84	5.12	MHz

Data Line

This is a bi-directional line used to transfer data synchronously between the controller and slaves. The direction of the data line is not determined by the physical layer but by the control information in the transport layer. This is described in the next section. When no data transfers are in progress the data line is always set to input on both the controller and slaves. This line is pulled low. Data is changed on the falling edge of clock by the transmit device and latched into the receiving device on the rising edge of the clock.

The Physical layer

This section specifies the low level protocol of the SIBO serial protocol. The physical layer protocol consists of a series of 12 bit frames. There are four types of frames:-
Null frames - Transmitted by controller to synchronise slaves.
Control frames - Control information transmitted by controller to slaves.
Data output frames - Data frame transmitted by controller to slaves.
Data input frames - Data frame received by controller from a slave.

Frame structure

All 12 bit frames have the following structure:-

Bit	0	1	2	3	4	5	6	7	8	9	10	11
Name	ST	CTL	I1	D0	D1	D2	D3	D4	D5	D6	D7	I2

ST	Start bit. This bit goes high to indicate the start of a valid frame.
CTL	Control bit. When low indicates this is a control frame. High indicates a data frame.
I1	Idle bit. Used to turn around direction of data line. Normally Low.
D0-D7	Data bits.
I2	Idle bit. Used to turn around direction of data line. Normally low.

Null Frame

This is a special frame transmitted by the controller to ensure all slaves are synchronised. It is generated by transmitting 12 clock pulses with the data line set to input. Since the data line is pulled low this results in 12 zeroes being transmitted.

Control frame

This frame is transmitted from the controller to one or more slaves. The data line is an output from the controller throughout the whole frame. The bits in the frame have the following value in a control frame:

ST Start bit.	This bit goes high to indicate the start of a valid frame.
CTL Control bit.	Low to indicate this is a control frame.
I1 Idle bit	Set low.
D0-D7 Data bits.	8 bits of control information.
I2 Idle bit	Set low.

Data Output Frame

This frame is transmitted from the controller to one or more slaves. The data line is an output from the controller throughout the whole frame. The bits in the frame have the following value in a data output frame:

ST Start bit.	This bit goes high to indicate the start of a valid frame.
CTL Control bit.	High to indicate this is a data frame.
I1 Idle bit	Set low.
D0-D7 Data bits.	8 bits of transmitted data.
I2 Idle bit	Set low.

Data Input Frame

This frame is received by the controller from a slave. The data line is an output from the controller for cycles 1 and 2 and input to the controller for cycles 4 to 11. The bits in the frame have the following value in a data input frame:

ST Start bit.	Output from controller. This bit goes high to indicate the start of a valid frame.
CTL Control bit.	Output from controller. High to indicate this is a data frame.
I1 Idle bit.	Used to turn around direction of data line. Both controller and slave
should	tri-state the data line during this bit. This bit should be low due to pull
down	resistor on data line. The controller changes the data line from output to

	input at the end of cycle 2. The slave changes the data line from input to output at the start of cycle 4.
D0-D7 Data bits.	Output from slave 8 bits of data transmitted by slave. Controller sets data line to input during these bits.
I2 Idle bit. should resistor	Used to turn around direction of data line. Both controller and slave tri-state the data line during this bit. Should be low due to pull down on data line. The slave changes data line from output to input at the end of cycle 11.

Data line direction

The following table summarises the direction of the data line.

	CONTROLLER		SLAVE	
Condition	CK	DATA	CK	DATA
Outside Frame	T	I	I	I
Null frame	O	I	I	I
Control Frame	O	O	I	I
Data output from controller	O	O	I	I
Data input to controller:-				
Cycles 1-2	O	O	I	I
Cycle 3	O	I	I	I
Cycles 4-11	O	I	I	O
Cycle 12	O	I	I	I

Key	T	Tri-state
	I	Input
	O	Output

The Transport layer

This section specifies the transport level protocol that operates above the SIBO serial communication physical layer. The transport layer protocol controls the serial communication between the SIBO Protocol Controller (SPC) and a number of SIBO Protocol Slave (SPS) devices. The following rules apply:-
1) The interface is controlled by the writing of control bytes from the controller to the slaves. Control bytes cannot be written by the slaves.
Unsolicited data cannot be sent from the slave to the controller.
2) The controlling device contains two registers to communicate to the slaves. These are the control register (byte, write only) and the Data register (byte or word, read/write).
Control bytes are transmitted to the slaves by writing to the control register.
The format of the control byte is as follows:-

Bit	7	6	5	4	3	2	1	0
Name	S	x	x	x	x	x	x	x

The control word can have 2 distinct formats depending on the setting of bit 7 the Select (S) bit:-
Select = 0 This is the slave select mode. This mode is for selecting, deselecting and resetting slaves.
Select = 1 This is the slave control mode. This mode is for communicating with a slave which has been previously selected using the select slave command.

Slave select mode

The format of the slave select byte is as follows:-

Bit	7	6	5	4	3	2	1	0
Name	0	R	I	I	I	I	I	I

Key:-

R	single reset bit.
IIIIII	6 bit ID field.

The 6 bit ID field is a property only of the slave. No slave may have an ID of zero, hence there can be 63 different slaves connected to one controller. The reset bit (R) controls whether the slave(s) are selected or reset. If R = 0 slave(s) are reset, R = 1 slave(s) are selected. Slave select control bytes can be summarised by the following table:-

S	R	ID	Description
0	0	0	Reset all slaves
0	0	xx<>0	Reset specific slave with ID = xx
0	1	0	Deselect slave (does not reset slave)
0	1	xx<>0	Select slave with ID=xx and read slave info (see below).

The Reset function is dependant on the slave. It would normally put the slave into a known passive reset state.
Select Slave with ID=xx (S=0,R=1)
This is a special command that causes a slave with ID=xx to transmit to the controller an 8 bit information field. This field depends entirely on the slave but must be non zero. A reply of 0 indicates that there is no slave of the requested ID present.

Slave control mode

This mode is for communicating with a slave which has been previously selected using the select slave command described above.
The format of the control word in slave select mode is as follows:-

Bit	7	6	5	4	3	2	1	0
Name	1	R/W	B/W	S/M	X	X	X	X

Key

R/W	Read/write select. 0 = write, 1 = read
B/W	Data transfer size. 0 = 1 byte transfer, 1 = word (2 byte transfer).
S/M	Single/Multi transfer mode. 0 = single, 1 = multibyte.

XXXX = 4 bits of data to slave.

Note the meaning of the 4 bits of data (XXXX) is entirely dependent on the slave.
The settings of R/W,B/W,S/M bits in the control word determine the size, type and direction of subsequent data transfers in the following manner:-

R/W	B/W	S/M	
0	0	0	write a single byte to slave
0	0	1	write a number of single bytes to slave
0	1	0	write a byte pair to slave (not implemented)
0	1	1	write a number of byte pairs to slave (not implemented)
1	0	0	read a single byte from slave
1	0	1	read a number of single bytes from slave
1	1	0	read a byte pair from slave (not implemented)
1	1	1	read a number of byte pairs from slave (not implemented)

Write a single byte

This command readies the currently selected slave to receive a byte of data and sets up the controller so that the next byte (or the LSB of a word) written to its data register will be transmitted to that slave. Anything further written to the controller's data register will have no effect.

Write a number of single bytes

This command readies the currently selected slave to receive a number of sequential bytes of data. The slave will expect to receive data bytes until another control byte is received. The controller is set up so that the next byte (or the LSB of a word) written to its data register will be transmitted to that slave. All subsequent bytes written to the controller's data register will be transmitted to the slave. This will continue until another byte is written to the controller's control register.

Write a byte pair

This command readies the currently selected slave to receive two bytes of data and sets up the controller so that the next word written to its data register will be transmitted to that slave (LSB first). Anything further written to the controller's data register will have no effect.

Write a number of byte pairs

This command readies the currently selected slave to receive a number of sequential byte pairs of data. The slave will expect to receive byte pairs until another control byte is received. The controller is set up so that the next word written to its data register will be transmitted to that slave (LSB first). All subsequent words written to the controller's data register will be transmitted to the slave. This will continue until another byte is written to the controller's control register.

Read a single byte

This command triggers a byte to be transmitted from the selected slave to the controller. This byte can then be read from the LSB of the controller's data register. Further reads of the controller's data register will return the same data but have no effect on the protocol.

Read a number of single bytes

This command triggers a byte to be transmitted from the selected slave to the controller. This byte can then be read from the LSB of the data register. This read will trigger the next byte to be transmitted to the data register of the controller. All subsequent reads of the controller's data register will trigger further bytes to be transmitted to the controller. This will continue until another byte is written to the controller's control register.

Read a byte pair

This command triggers a byte pair to be transmitted from the selected slave to the controller. This word can then be read from the controller's data register. Further reads of the controller's data register will return the same data but have no effect on the protocol.

Read a number of byte pairs

This command triggers a byte pair to be transmitted from the selected slave to the controller. This word can then be read from the controller's data register. This read will trigger the next byte pair to be transmitted to the data register of the controller. All subsequent reads of the controller's data register will trigger further byte pairs to be transmitted to the controller. This will continue until another byte is written to the controller's control register.

Timing

The time taken for commands to be processed and data sent is shown below. The time is given in SIBO pack protocol clock cycles. The length of a clock cycle is nominally 260 nanoseconds for a clock frequency of 3.84 MHz.

Receive and process the control byte	12 cycles
Byte transfer to or from slave	12 cycles
Byte pair transfer to or from slave	24 cycles

When writing to the controller's data and control registers the following rules apply:-

- After writing to the control register there must be a delay of at least 12 cycles before the data register is accessed or another control word is written.
- To read a word from the data register after the command to read byte pair is issued there must a delay of at least 12 (for control byte)+24 (for the byte pair transfer)= 36 cycles.
- To perform a multiple byte pair write there must be a delay of at least 12 cycles after the command is written to the control register before the first word can be written to the data register and a delay of at least 24 cycles between subsequent writes to the data register.

States

A slave can be in one of 5 states. Note a control byte can be received and interpreted at any time.

1) Waiting to receive a data byte or control byte
2) Waiting to receive a data byte pair or control byte
3) Waiting to transmit a data byte or control byte
4) Waiting to transmit a data byte pair or control byte
5) Waiting to receive control byte only

5. Mechanical Overview

This section will contain information regarding mechanical and plastic moulding for Series 3/3a, Workabout and HC machines that is deemed to be of especial importance to developers who are considering producing peripherals for these particular Psion platforms.

The Psion Series 3/3a range

S3a/Series 3 Reduced External Expansion Port

The Psion Series 3/S3a personal digital assistants have two SSD slots and provide access to external peripheral units through a single reduced internal expansion port, port C, on the left edge of the machine. The reduced external serial interface expansion port from the Series 3/3a forms six wires. The purpose of each is described in the table below. In addition to data, clock and power an active high interrupt line is provided. This allows the peripheral device to generate an interrupt within the host series 3/3a. The level of interrupt that is generated depends on both the machine and the expansion port that is used. Either ASIC4 or ASIC5 can act as the other end of the Psion Serial Interface. With exception of the interrupt line all used signals should be connected directly to the appropriate pins on ASIC4/5.

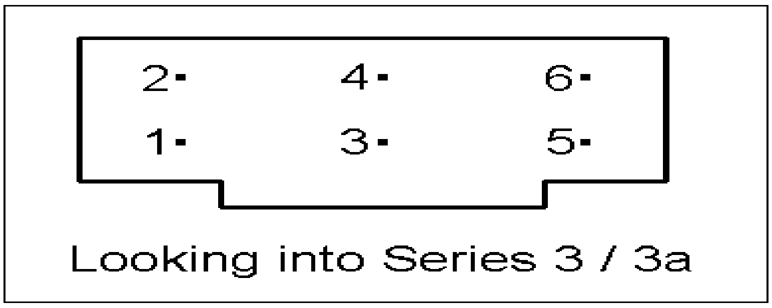
Pin	Name	Description	Connect to
1	MSD	Data line	ASIC4/5 SDAT
2	MCLK	Serial clock	ASIC4/5 SCLK
3	Vcc	+5 volt supply	Vcc
4	GND	Signal ground	GND
5	SSD/INT	Interrupt line	Interrupt source/GND
6	SCK/EXON	Not used in this scenario	Do not Connect

Signal Definition

MSD and MCLK form a single master SIBO serial protocol channel. This is normally channel 7 on a Series 3 and channel 5 on an S3a. The serial channel clock can be continuously enabled to provide a free running clock for expansion devices. The frequency is fixed at 1.536MHz regardless of the system clock frequency. SDKs/INT and SCK/EXON are both dual function pins. SDKS and SCK form a single slave SIBO serial protocol channel. This can be combined with MSD and MCLK to form a bi-directional high speed data link. SDS/INT can also be used to as an active high interrupt input. The function of SDS/INT can be programmed in ASIC2 or ASIC9. A rising edge on the SCK/EXON input will bring the system out of the standby state into the operating state. VCC is a +5 volt supply that is switched off when the system is in the standby state and is switched on when the system is in the operating or idle state. The maximum current that can be drawn is 25mA. Opening the pack doors on either an S3a or a Workabout will cut power to external peripherals.

Physical Connector

The reduced expansion port is made up of a 6-way two row connector spaced on a 2x3 way 0.1 inch pitch. The diagram below shows the physical connector numbering:



The male plug is connected to a 0.5m long plastic moulded 3-link cable assembly (part no. 25020013) which is terminated in a six-pin in-line connector which connects to a 6-way 1.5mm pitch transition header (part no. 47000106).

The Psion Workabout

The Workabout Expansion Interfaces

The Psion Workabout provides a rugged and easy-to-use computer system for a wide range of mobile corporate needs. The machine can be readily adapted to support various peripheral units such as barcode scanners and modems attached to the expansion ports. The Workabout has a 26-way extended internal expansion interface and a special 11-pin reduced external expansion interface.

Workabout Extended Internal Expansion Interface

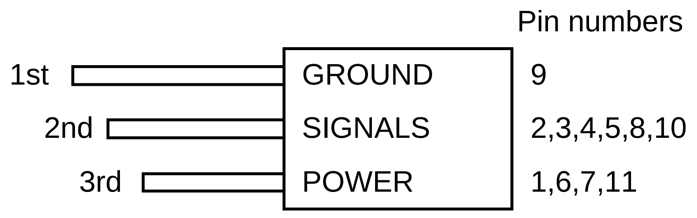
The pin-out of the Workabout 26-way internal Torson connector is outlined below:

Torson 26 way connector pin	Workabout Signal name
1	**GND**
2	*NICD (not used)*
3	**RUN**
4	*Vh (not used)*
5	**Vcc1**
6	**Vcc2**
7	*CODEN (not used)*
8	*AMPEN (not used)*
9	*VOL0 (not used)*
10	*VOL1 (not used)*
11	*SCK (not used)*
12	*SYNC (not used)*
13	*SIN (not used)*
14	*SOUT (not used)*
15	*ESDOE (not used)*
16	**SCK2**
17	**SD2**
18	**EINT1 (active low)**
19	**SCK3**
20	**SD3**
21	**EINT2 (active low)**
22	*THM (not used)*
23	*VIN (not used)*
24	**EXON**
25	*N/C*
26	**GND**

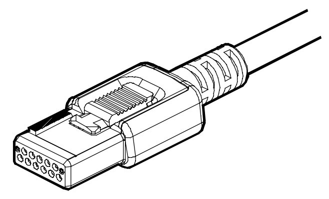
- Vcc1 is 3.0V nominal power supply. Current available from Workabout is limited to 100mA.
- Vcc2 is 5V nominal power supply. Current available from Workabout is limited to 200mA.
- RUN is low when powered down and high when powered up. It is used to power down or reset the peripheral module.
- SCK2 and SCK3 are serial data clocks. The clocks are left running continuously at 1.536MHz when the serial port is in use. They are used to clock the UART in ASIC5 in the RS232 AT/TTL and AT/Barcode modules respectively.
- SD2 and SD3 are bi-directional serial data lines used in the RS232 AT/TTL and AT/Barcode modules respectively.
- EINT1 and EINT2 are active low signals for interrupt input.
- EXON is an active high signal used to turn on the Workabout.
- All of the above logic signals are at 3.0V or 3.3V levels, depending upon the logic supply Vcc1.
- Lines currently described as unused relate to a yet unspecified codec interface.

Workabout Reduced External Expansion Interface

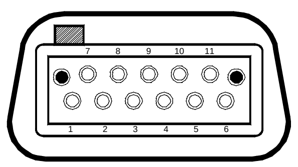
For the Workabout reduced external expansion interface, a new 11-pin Low Insertion Force (LIF) connector has been designed for connecting the computer to the Cradle System. The computer mounted male LIF may be weather proofed, the cable mounted female LIF cannot. Currently the LIF connector cover can be moulded with a polarising pin in one of two positions. The facility exists to manufacture the cover with the polarising pin in two more positions, giving four possible variants. If more than four versions are required it is possible to have the cover and the socket bezel moulded in a range of colours to differentiate between variants. The polarising options are presented below:



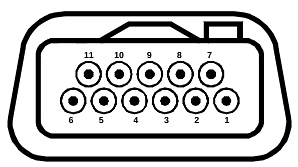
The step arrangement of the LIF Connector pins



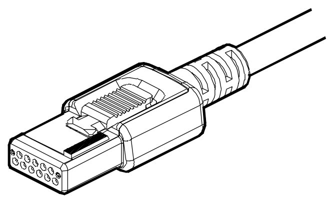
Polarising Pin A - Vehicle Interface Box for the HC only



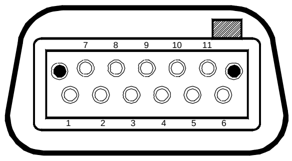
Cable mounted LIF (Female plug)



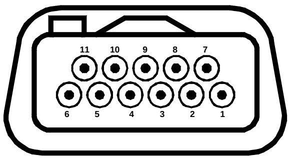
Computer mounted LIF (Male socket)



Polarising Pin B - HC and Workabout LIF



Cable mounted LIF (Female plug)



Computer mounted LIF (Male socket)

Type A and Type B polarisation of the LIF Connector

Pin Definition for LIF - PFS Connector

LIF Connector Polarisation Type B

Pin No	Pin Name	Wire Gauge	Colour	Contact	Direction (Cradle's perspective)	Standard Function	Cradle usage
1	LCA	7/0.1	Brown	third	Input	Local[1] Computer Active. High when the computer is on. (The Workabout can source 100mA from this pin and the HC/HCDOS 5mA to power remote circuitry)	Used as an enable for the cradle resident Xmod 5V supply.
2	EXON	7/0.1	Blue	second	Output	EXternal switch ON, active high (+5V). Asserted by a remote[2] device to switch on the computer.	May be asserted by a cradle resident Xmod.
3	INT	7/0.1	Orange	second	Output	INTerrupt to computer, active high (+5V).	May be asserted by a cradle resident Xmod.
4	THM	7/0.1	Yellow	second	Input	Battery thermistor terminal. Allows remote[2] sensing of the battery temperature.	Standard function
5	DLA	7/0.1	Green	second	Output	Disconnect Local[3] ASIC, active high (+5V). (does not apply to Workabout). When this signal is asserted the serial channel is disconnected from the local[3] ASIC4/5 in the HC resident Xmod (if present) and instead connected to a remote ASIC4/5 (if present).	Asserted by the Cradle ASIC, connects the cradle resident Xmod to the serial channel.
6	BAT	28 SWG	Red	third	Output	+ve battery terminal (1 amp)	Standard function
7	Vin	28 SWG	Black	third	Output	Power supply to computer (+10V)	Standard function
8	SCLK	7/0.1	Grey	second	Input	Serial channel CLocK.	Standard function
9	GND	28 SWG	White	first	-	Power, signal ground and -ve battery terminal (1 amp)	Standard function
10	SDATA	7/0.1	Violet	second	Bi-directional	Serial channel DATA.	Standard function
11	STATUS	7/0.1	Pink	third	Output	STATUS. Connected to a pull-up resistor to allow connection to an open-collector/drain driver. Normal usage is: low indicates the presence of a remote[2] device.	Driven low by an open collector driver when LCA is high and the cradle is powered-up to allow the computer to sense whether or not the cradle is connected.

LIF Connector Polarisation Type A

Pin No	Pin Name	Wire Gauge	Colour	Contact	Direction (Computer's perspective)	Function
1	DCD	7/0.1	Brown	third	Input	RS232 signal
2	RX	7/0.1	Blue	second	Input	RS232 signal
3	TX	7/0.1	Orange	second	Output	RS232 signal
4	THERM	7/0.1	Yellow	second	-	Battery thermistor terminal
5	DTR	7/0.1	Green	second	Output	RS232 signal
6	VBAT	28 SWG	Red	third	-	+ve battery terminal
7	VIN	28 SWG	Black	third	Input	Power supply to computer
8	DSR	7/0.1	Grey	second	Input	RS232 signal
9	GND	28 SWG	White	first	-	Power, signal ground and -ve battery terminal
10	RTS	7/0.1	Violet	second	Output	RS232 signal
11	CTS	7/0.1	Pink	third	Input	RS232 signal

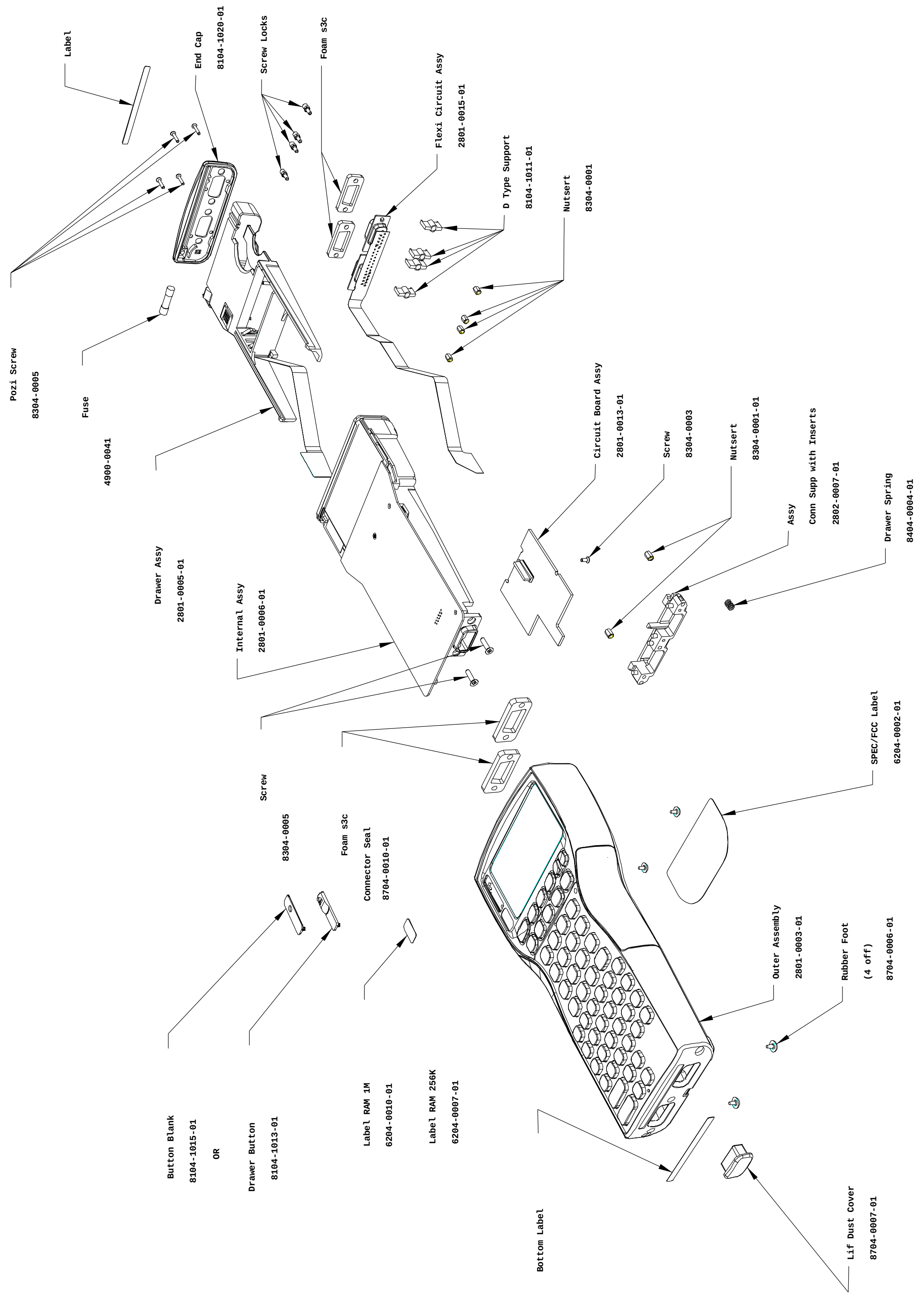
Definitions

Computer	HC, HCDOS or Workabout
Cradle resident Xmod	Expansion module fitted to the cradle, may or may not be present.
HC resident Xmod	Expansion module fitted to the HC, which contains the cradle interface and possibly another peripheral.
HC peripheral	A peripheral, located in the HC resident Xmod which is connected to the same serial channel as the cradle.
Cradle ASIC	An ASIC5 located on the main cradle PCB which remains connected to the serial channel irrespective of the state of DLA.

1. The term "local computer" implies the computer local to the LIF connector, i.e. the HC, HCDOS or Workabout, as opposed to a "remote" computer which might be connected via a cradle resident Xmod for example.
2. The term "remote" implies something on the other side of the LIF connector to the computer.
3. The term "local" implies something on the computer side of the LIF connector including devices on an HC resident Xmod.

Exploded view of the Psion Workabout

Below is an exploded view of a complete Psion Workabout showing part numbers for various components and illustrating the positioning of the expansion interfaces.


Label
End Cap 8104-1020-01
Screw Locks
Foam s3c
Flexi Circuit Assy 2801-0015-01
D Type Support 8104-1011-01
Nutsert 8304-0001
Pozi Screw 8304-0005
Fuse 4900-0041
Drawer Assy 2801-0005-01
Internal Assy 2801-0006-01
Circuit Board Assy 2801-0013-01
Screw 8304-0003
Nutsert 8304-0001-01
Assy Conn Supp with Inserts 2802-0007-01
Drawer Spring 8404-0004-01
Screw 8304-0005
Foam s3c Connector Seal 8704-0010-01
SPEC/FCC Label 6204-0002-01
Outer Assembly 2801-0003-01
Rubber Foot (4 off) 8704-0006-01
Button Blank 8104-1015-01
OR
Drawer Button 8104-1013-01
Label RAM 1M 6204-0010-01
Label RAM 256K 6204-0007-01
Bottom Label
Lif Dust Cover 8704-0007-01

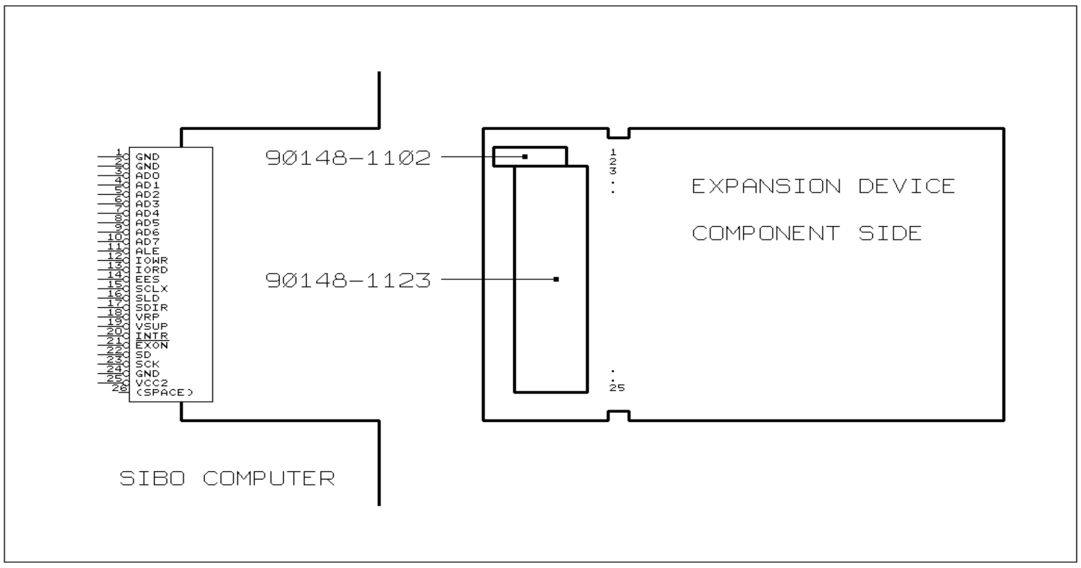
The Psion HC range

HC Extended Internal Expansion Port

The Psion HC range of computers are intended to provide a rugged and powerful mobile computer system for a wide variety of demanding application requirements. As part of its adaptability, every element of the hardware is configurable from the plug-in SSDs to the expansion ports for peripheral devices such as bar code scanners, modems and magnetic card readers. There are two independent extended internal expansion ports at either end of an HC unit. The top port is termed the "A" port and the bottom one the "B" port. Each expansion port provides direct I/O with the processor, a SIBO serial channel, and connection to the power supply. It allows for higher powered expansion devices to be added by including a direct connection to the main 7.2 volt battery.

Physical Connector

The expansion port is made up of a 25 way single row connector spaced on a 0.1 inch pitch. The 26th position is a polarising key and should be left blank. The correct mating connector on the expansion device is made up from a number of Molex C-Grid series 90148 connectors. Pins 1 and 2 are ground and should have there own connector placed nearer the board edge to ensure the ground connection is made first when the expansion device is inserted. The required connectors are Molex 90148-1102 for the GND contacts and Molex 90148-1123 for the signal contacts. The diagram below shows the physical position of the connectors.



Signal definition

The following table defines the 25 way expansion connector:-

Pin No.	Name	Sig. type	Comments
1	GND	Power	Should mate first when device inserted
2	GND	Power	Should mate first when device inserted
3	AD0	B CMOS	8 bit multiplexed address and data bus pulled low with 100 K resistors
4	AD1	B CMOS	
5	AD2	B CMOS	
6	AD3	B CMOS	
7	AD4	B CMOS	
8	AD5	B CMOS	
9	AD6	B CMOS	
10	AD7	B CMOS	
11	ALE	O CMOS	Address latch enable - high when valid address on AD0-AD7
12	IOWR	O CMOS	I/O write strobe - active high, data valid on falling edge of IOWR
13	IORD	O CMOS	I/O read strobe - high when device can place valid data on AD0-AD7
14	EES	O CMOS	External Expansion Select - high during I/O cycles to expansion device
15	SCLX	T CMOS	512 KHz SCL signal for SLD bus - usually Hi-Z and pulled low
16	DNC	N/A	For future expansion - do not use.
17	THERM	Resistor	Connected to thermistor (bottom slot only - top slot DNC)
18	VB1	Battery	Connected to the internal NiCd battery (bottom slot only - top slot DNC)
19	Vsup	Power	Unregulated battery voltage - present all the time
20	INTR	I CMOS	Active high interrupt input
21	_EXON	I CMOS	Active low input pulled up to Vcc1 - pull low to switch machine on
22	SD	B CMOS	SIBO serial protocol data line - pulled low
23	SCLK	T CMOS	SIBO serial protocol clock line - Hi-Z in standby needs a pull down
24	GND	Power	
25	Vcc2	Power	+5 volt supply, switched off in standby. Max current available = 50 mA

DNC (Do Not Connect) indicates that the pin should not be connected. The signal types are:-

O CMOS	CMOS output to the expansion device.
B CMOS	CMOS bi-directional line to the expansion device.
T CMOS	CMOS tri-state output to the expansion device.
I CMOS	CMOS input from the expansion device.

All the CMOS signals including the AD0-AD7 bus are buffered from the main system busses and so present a load of one HC series logic gate.

Direct I/O

Expansion devices can be connected to direct processor I/O space using the following signals; AD0-AD7, ALE, IOWR, IORD, EES and INTR.

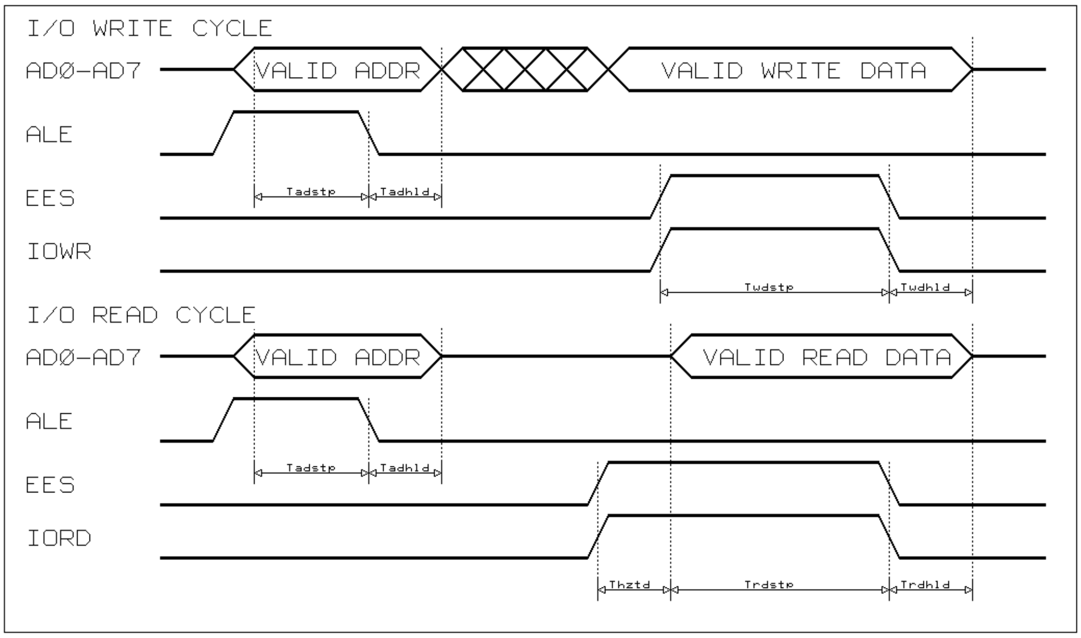
AD0-AD7 is the least significant half of the multiplexed address and data bus, this means that up to 128 I/O addresses are available for each expansion device. As only the least significant half of the bus is available and no bus conversion is done only even addresses can be used.

ALE must be used to latch the address from AD0-AD7 for devices that require a stable address. The address is valid on the falling edge of ALE. Note that A0 will always be low for valid writes to the expansion device and as such should not be used as an address line, A1 should be used as the lowest order address line. A0 can be used as an additional enable signal to stop odd I/O accesses disturbing the expansion device.

EES is the External Expansion Select and is high during all I/O accesses to the expansion device, i.e. for I/O reads and writes to address range 100 to 1FF hex. for expansion port 1, and 200 to 2FF for expansion port 2.

IOWR is an active high signal which is high during all I/O write bus cycles. The data on AD0-AD7 is guaranteed to be stable before the rising edge of IOWR and after the falling edge of IOWR. IORD is an active high signal which is high during all I/O read bus cycles. The AD0-AD7 bus is guaranteed to be tri-state before the rising edge of IORD and after the falling edge of IORD. The expansion device must present valid data on the bus when IORD is high, see the timing details below.

INTR is an active high interrupt input to ASIC1. T his can be used as a directly readable bit or as a dedicated interrupt input. It must not be driven high when the system is in the standby state as this input is pulled down and will cause excessive standby current consumption. The diagram below shows the timing of the I/O write and read cycles.



Symbol	Parameter	Min	Typ	Max	Units
Tadstp	Address set up time	60	100	370	nSec
Tadhld	Address hold time	50	80	-	nSec
Twdstp	Write data set up time	260	390	-	nSec
Twrcyc	Write cycle pulse width	260	390	780	nSec
Twdhld	Write data hold time	80	150	-	nSec
Thztd	Time from Hi-Z to data active	0	-	240	nSec
Trdcyc	Read cycle pulse width	340	520	1040	nSec
Trdstp	Read data set up time	130	-	-	nSec
Trdhld	Read data hold time	0	-	-	nSec

Note the typical values given are for an HC with a system oscillator of 7.68 MHz.

The SIBO serial channel

A single SIBO serial protocol channel is provided on each expansion port. Expansion port 1 is connected to serial channel 5, expansion port 2 is connected to serial channel 6. The serial channel clock can be continuously enabled to provide a free running clock for expansion devices. The frequency is fixed at 1.536 MHz regardless of the system clock frequency. This frequency is a multiple of the SLD clock rate and of all normal RS232 baud rates.

High Speed Side Port

Port C on an HC houses an 8-pin in-line connector that allows access to a high speed SIBO serial interface. This interface is currently used only by the HC cradle peripheral. Its pin-out is outlined in the table overleaf:

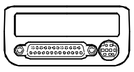
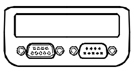
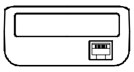
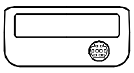
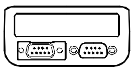
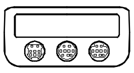
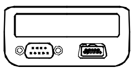
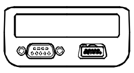
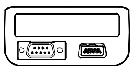
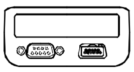
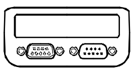
Pin	**Signal**	**Type**	**Description**
1	SDS/INT	In/Out	Fast Serial Data (Slave)/External interrupt
2	GND	Power	GND connection
3	SCKS/EXON	Output	Fast Serial Clock (Slave)/External Power-on
4	SCK7	Output	Serial Data Clock
5	VSLED	Power	External DC supply to HC
6	SD7	In/Out	Bi-directional Serial Data line - Side port C
7	VFC	Power	Battery VB2 (for charging battery pack)
8	TSEN	Input	Battery temperature sensor

Power supplies

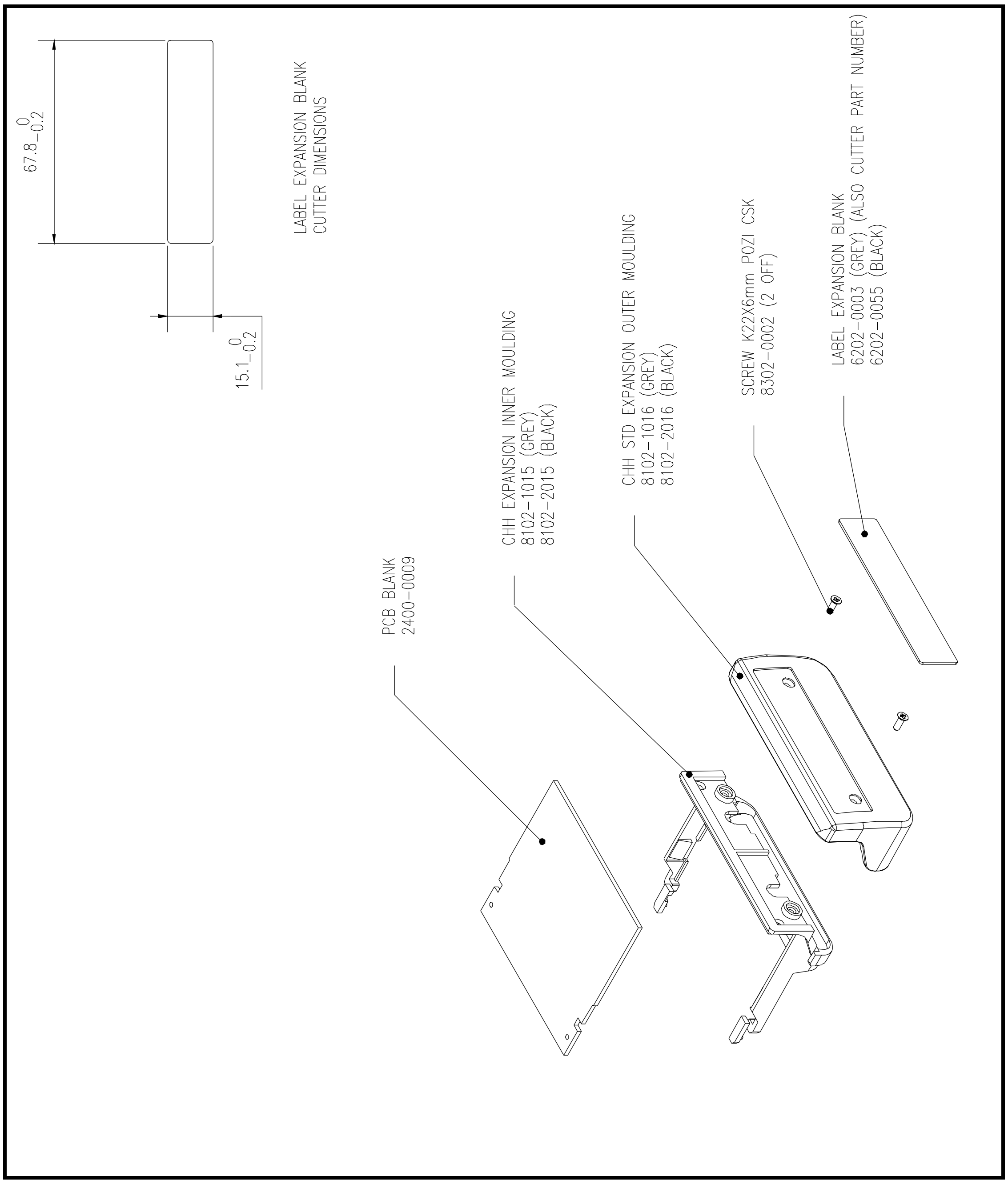
Two power supplies are available for expansion devices these are Vcc2 and Vsup. Vcc2 is a +5 volt supply that is derived from Vsup. Vcc2 is switched off when the system is in the standby state and is switched on when the operating or idle state is entered. Each expansion device can draw up to 50 mA from Vcc2. If an expansion device requires more than 50 mA or cannot be powered down when the system is in the standby state the Vsup power supply must be used. Vsup is the unregulated supply directly from the main system batteries or from the DC jack input. It will be in the range 5.5 to 12 volts under normal conditions. To use Vsup the expansion device must regulate Vsup to 5 volts with a low drop-out linear regulator. Care must be taken to not be active and driving any signals high when the system is in the standby state as Vsup is always present. This can be achieved either in software or by using Vcc2 as a signal indicating the active state.

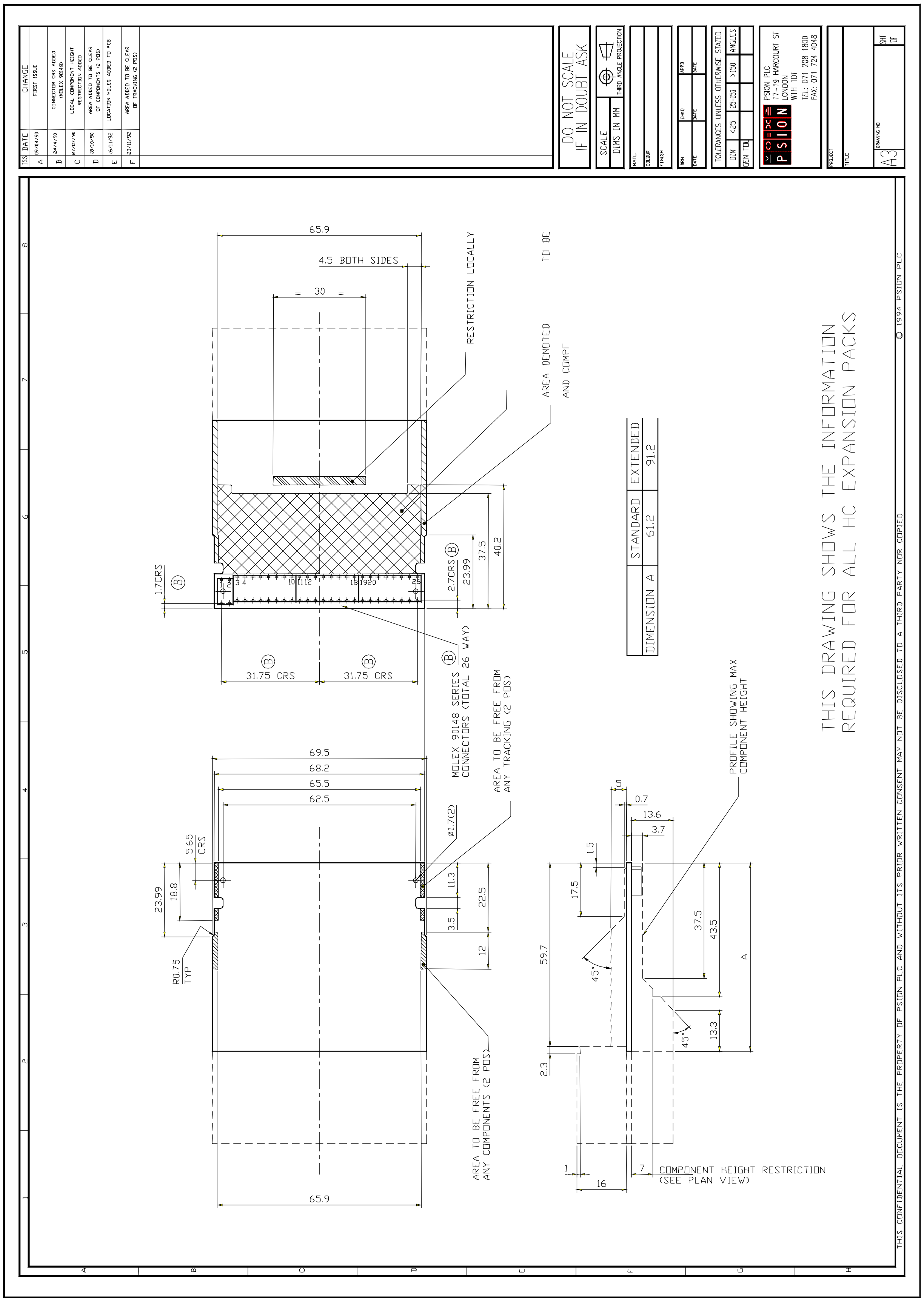
Mechanical Information

Mechanical details regarding the numerous build variants and accessories that currently exist for the HC are presented overleaf. The peripheral expansion boards for these build variants are housed in a special plastic casing that can be machined to hold the requisite connectors. In the matrix, a • indicates that the relevant combination of HC and accessory are compatible and an X indicates that the combination is not compatible. Following the build variant table are two diagrams that display exploded views of the HC expansion connector and the HC Expansion Board complete with part numbers and dimensions.

	Build Variants	**HC 100**	**HC 110**	**HC 120**	**HC DOS**	**HCR 400**	**FAST CHARGER**
	With EL Backlighting	●	●	●	●	●	
	Without EL Backlighting	●	●	●	X	X	
	Industrial	X	●	●	●	●	
	Non-Industrial	●	●	●	X	X	
	Keypad Variants						
	53 Key A/N UK 2401-0026	●	●	●	X	X	
	A/N European 2401-0051	●	●	●	X	●	
	A/N Scandinavian 2401-0050	●	●	●	X	X	
	Numeric only UK 2401-0046	●	●	●	X	X	
	DOS Keypad 2401-0147	X	X	X	●	X	
	53 Key A/N USA 2400-0026	X	X	X	X	●	
	HC Expansion Modules						
	**RS232 / Parallel (Printer) 1502-0001** 25 way D type (F) + 9 way Mini DIN Certified FCC Class B / Passed VDE Class B	●	●	●	●	●	●
	**RS232 / TTL** 1502-0039 (IP64), 1502-0040 (NON IP64) 9 way D type (F) + 9 way D type (M) Passed FCC Class B / Passed EN55022 Class B	●	●	●	●	●	●
	**UK Modem (Asic 8) 1502-0010** RJ 11 connector BABT Approved in UK, BS6301 (Safety)	●	●	●	X	●	●
	**Barcode Only** HP Wand HBCS-A207 + Plug + EXMOD 1502-0020 Wand Welch Allen + Plug + EXMOD 1502-0021 FCC Class A / Passed VDE Class B	●	●	●	X	●	X
	**RS232 / Barcode 1502-0044** 9 way D type Quick Loc(F) + 9 way D type (M) Complies with FCC Class A	●	●	●	●	●	●
	**MCR / Scanner / RS232 1502-0003** Scanner NipDenso + Plug 1502-0022 Scanner DigVision + Plug 1502-0023 Magnetic Card Reader + Plug 1502-0024 Certified FCC Class B / Passed VDE Class B	●	●	●	X	●	●
	**LIF / RS232 (Under development)** 9 way D type (M) + 9 way LIF- PFS (M)	●	●	●	●	X	X
	**LIF / TTL (Under development)** 9 way D type (F) + 9 way LIF- PFS (M)	●	●	●	●	X	X
	**LIF / BARCODE 1502-0043** 9 way D type Quick Loc(F) + 9 way LIF- PFS (M)	●	●	●	●	●	X
	**Vehicle / TTL** 9 way D type (F) + 9 way LIF- RS232 (M)	●	●	●	●	●	X
	**16550 RS232 / TTL (Under development)** 9 way D type (F) + 9 way D type (M) Complies with FCC Class A	●	●	●	●	●	X
	Printer (High Res.) 1502-0037	●	●	●	●	X	●
	Laser Scanner 1503-0012	●	●	●	●	X	
	Fast Charger Variants						
	Fast Charger with Holster (Due Jan 95)	●	●	●	●	X	
	Fast Charger without Holster (Due Jan 95)	●	●	●	●	X	
	Additional Accessories						
	Nicad Battery Pack 500 mA 1503-0005	●	●	●	●	X	

Corporate Hand Held (CHH) Expansion Module Exploded Assembly Drawing

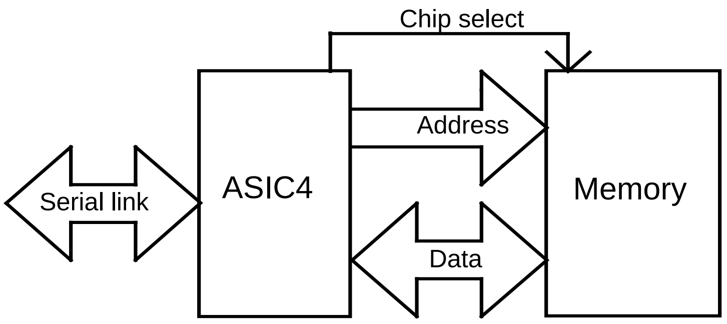

67.8 0 -0.2
LABEL EXPANSION BLANK CUTTER DIMENSIONS
15.1 0 -0.2
PCB BLANK 2400-0009
CHH EXPANSION INNER MOULDING 8102-1015 (GREY) 8102-2015 (BLACK)
CHH STD EXPANSION OUTER MOULDING 8102-1016 (GREY) 8102-2016 (BLACK)
SCREW K22X6mm POZI CSK 8302-0002 (2 OFF)
LABEL EXPANSION BLANK 6202-0003 (GREY) (ALSO CUTTER PART NUMBER) 6202-0055 (BLACK)


ISS DATE CHANGE
A 09/04/90 FIRST ISSUE
B 24/4/90 CONNECTOR CRS ADDED (MOLEX 90148)
C 27/07/90 LOCAL COMPONENT HEIGHT RESTRICTION ADDED
D 18/10/90 AREA ADDED TO BE CLEAR OF COMPONENTS (2 POS)
E 16/11/92 LOCATION HOLES ADDED TO PCB
F 23/11/92 AREA ADDED TO BE CLEAR OF TRACKING (2 POS)
DO NOT SCALE IF IN DOUBT ASK
SCALE
DIMS IN MM
THIRD ANGLE PROJECTION
TOLERANCES UNLESS OTHERWISE STATED
DIM <25 25-150 >150 ANGLES
GEN TOL
PSION PLC 17-19 HARCOURT ST LONDON W1H 1DT
TEL: 071 208 1800 FAX: 071 724 4048
PSION
A3
65.9
4.5 BOTH SIDES
30
RESTRICTION LOCALLY
AREA DENOTED TO BE
AND COMP
1.7CRS
2.7CRS
23.99
37.5
40.2
31.75 CRS
31.75 CRS
MOLEX 90148 SERIES CONNECTORS (TOTAL 26 WAY)
AREA TO BE FREE FROM ANY TRACKING (2 POS)
69.5
68.2
65.5
62.5
Ø1.7(2)
5.65 CRS
23.99
18.8
11.3
22.5
3.5
12
R0.75 TYP
AREA TO BE FREE FROM ANY COMPONENTS (2 POS)
65.9
DIMENSION A
STANDARD 61.2
EXTENDED 91.2
PROFILE SHOWING MAX COMPONENT HEIGHT
5
0.7
13.6
3.7
1.5
17.5
59.7
45°
37.5
43.5
A
45°
13.3
2.3
1
16
7
COMPONENT HEIGHT RESTRICTION (SEE PLAN VIEW)
THIS DRAWING SHOWS THE INFORMATION REQUIRED FOR ALL HC EXPANSION PACKS
© 1994 PSION PLC
THIS CONFIDENTIAL DOCUMENT IS THE PROPERTY OF PSION PLC AND WITHOUT ITS PRIOR WRITTEN CONSENT MAY NOT BE DISCLOSED TO A THIRD PARTY NOR COPIED

6. ASIC 4

What is ASIC 4?

ASIC4 is custom integrated circuit designed for use in Memory Packs (called SSDs) and peripheral devices. Its primary purpose is to convert the PSION Serial protocol into the signals required to address memory and memory-mapped peripherals. A typical Series 3/3a ASIC4 peripheral will consist of an ASIC4 connected to port C of the host machine and a peripheral chip/device mapped into ASIC4's addressing space. An example could be an ASIC4 connected to a simple 1k memory device:



To write a value, `<v>`, to address `<addr>`, the appropriate control codes must be sent along the port C Psion Serial link to assert `<addr>` on ASIC4's address outputs and `<v>` on ASIC4's data outputs.

ASIC4 Addressing and Modes

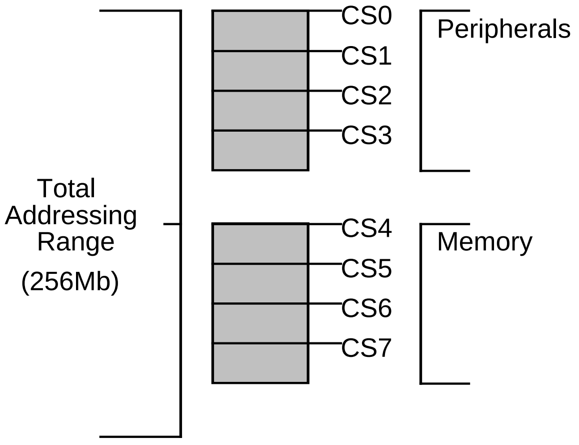
ASIC4 has an 8 bit data bus and a 28 bit address bus. Also provided are eight chip select lines that form selectable addressing blocks each of a size defined by software. The default is 32Kbyte/block. The filing system will set this to the appropriate size while accessing memory in the upper portion of the address space.

ASIC4 has two basic modes of operation, namely ASIC5 Compatibility (or SSD) mode and ASIC4 Extended mode. To select the mode, ASIC4 must be selected with an appropriate ID. This is achieved by writing a SIBO serial protocol slave control frame, as detailed in chapter four, to the ASIC prior to sending any read or write requests. After sending this frame, the so-called Info Byte is read off the data bus. Details concerning the meaning of the various Info Byte bits are provided in the following section. In the case of SSD mode, the ID is 2 and for ASIC4 Extended mode, the ID is 6.

Putting ASIC4 into **SSD mode** makes the chip compatible with all current versions of existing SSD software in production by Psion including all HC, series 3/3a and MC software. In this mode, ASIC4 mimics an ASIC5 in pack mode (see next chapter). This is because ASIC4 was originally designed to be a cut-down version of ASIC5 and so from the outset it was necessary to make previously existing ASIC5 software run on the new chip. The maximum address space in SSD mode is 21 address bits and 4 chip selects (which comes to 4 x 2Mb). In **ASIC4 Extended mode**, ASIC4 is capable of addressing up to 28 address bits (256Mb). In this mode, in addition to the Info Byte, a further 4 bits of information can be elicited from the state of the address lines A27-A24 during reset. Of these bits, the state of A27 (bit M) determines whether ASIC4 is going to be used as a standard SSD (M=0) or in a mixed mode (M=1) comprising of memory devices and peripherals. It is only the latter case which is of interest to the potential developer since this is the mode intended specifically for peripheral type expansion. In mixed mode ASIC4's address space is split into two equal halves. The lower half of the addressing range is set aside for memory-mapped peripherals and can be used for any purpose. The upper portion of the address space is reserved for pure memory. The Series 3/3a, Workabout and

HC filing system is able to use this memory (which does not have to be present) as an additional storage medium. Typically it will be a ROM containing the software that controls the peripheral. On reset, configuration data is supplied to ASIC4 on its data bus lines which the filing system can read in order to determine what form of and how much memory it has available in this upper region.

In mixed mode chip selects are split into four selectable peripheral blocks and four selectable memory blocks. CS0-CS3 are for peripheral access. CS4-CS7 select memory devices one to four. This set-up is illustrated below:



Reset and configuration

As indicated earlier, in mixed mode, following a reset or power up ASIC4 will read the form in which to configure itself from the data on data lines D0-D7 (the Info Byte) and address lines A24, A25, A26, A27 (most significant nibble of the Extended Info Byte). The table below shows the meaning of each of these lines during reset.

A27	**A26**	**A25**	**A24**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**

M	Peripheral Id Code	Memory Type	No. devices	Memory Size
1	Don't care, some codes are reserved. 0 0 0 No peripheral 0 0 1 T3Link 0 1 0 3Fax Contact Psion for an official code.	0 0 0 RAM 0 0 1 Type 1 Flash 0 1 0 Type 2 Flash 1 1 0 ROM	0 0 1 device 0 1 2 1 0 3 1 1 4	0 0 0 No memory 0 0 1 32Kbyte 0 1 0 64Kbyte 0 1 1 128Kbyte 1 0 0 256Kbyte 1 0 1 512Kbyte 1 1 0 1Mbyte

Pull up, pull down resisters would usually be used to place these lines in the desired state on reset. High value resisters of typically 100k would be used to allow ASIC4 and bus devices to drive these lines to other levels during normal operation.

ASIC4 should be powered from its host Series 3/3a/HC. High current peripheral chips or volatile memories should have their own supply. Whenever the Series 3/3a/HC is powered down, has its batteries removed, or has it's pack doors open, any attached ASIC4 will be powered down and reset upon resumption of power. Taking the 3Fax as an example with one Read Only Memory device of 512k the required configuration is:

1	0	1	0	1	1	0	0	0	1	0	1

ASIC4 Pin-out

Pin No	Pin Name	Direction	Pin Description
50	D0	I/O	Databus
49	D1	I/O	Databus
48	D2	I/O	Databus
47	D3	I/O	Databus
46	D4	I/O	Databus
45	D5	I/O	Databus
44	D6	I/O	Databus
43	D7	I/O	Databus
13	A0	O	Address bus - register AT0
12	A1	O	Address bus - register AT0
11	A2	O	Address bus - register AT0
8	A3	O	Address bus - register AT0
7	A4	O	Address bus - register AT0
6	A5	O	Address bus - register AT0
5	A6	O	Address bus - register AT0
4	A7	O	Address bus - register AT0
3	A8	O	Address bus - register AT1
2	A9	O	Address bus - register AT1
1	A10	O	Address bus - register AT1
64	A11	O	Address bus - register AT1
63	A12	O	Address bus - register AT1
62	A13	O	Address bus - register AT1
61	A14	O	Address bus - register AT1
60	A15	O	Address bus - register AT1
29	A16	O	Address bus - register AT2
28	A17	O	Address bus - register AT2
27	A18	O	Address bus - register AT2
25	A19	O	Address bus - register AT2
24	A20	O	Address bus - register AT2
23	A21	O	Address bus - register AT2
22	A22	O	Address bus - register AT2
21	A23	O	Address bus - register AT2/Oscillator Output in PSRAM Mode
20	A24	I/O	Address bus - register AT3 Inputs to set device size on reset
19	A25	I/O	Address bus - register AT3 Inputs to set device size on reset
18	A26	I/O	Address bus - register AT3 Inputs to set device size on reset
17	A27	I/O	Address bus - register AT3 Inputs to set device size on reset
37	CS0	O	Device chip selects
36	CS1	O	Device chip selects
35	CS2	O	Device chip selects
34	CS3	O	Device chip selects
33	CS4	O	Device chip selects
32	CS5	O	Device chip selects
31	CS6	O	Device chip selects
30	CS7	O	Device chip selects
16	OE	O	Output Enable/Refresh in PSRAM Mode
14	WR	O	Write pulse
15	VPS	O	VPP control
39	POR	I	Reset input
40	SCLK	I	Serial clock input
38	SDAT	I/O	Serial Data input
59	SDIR	O	Protocol Direction indication Bit
51	LBO	O	Low Battery detect driver output (Open drain)
56	MCSD	I	PSRAM Mode Select
52	IN0	I	General Purpose inputs
53	IN1	I	General Purpose inputs
54	IN2	I	General Purpose inputs/Refresh Disable in PSRAM Mode
55	X2D2	I	Oscillator Input for PSRAM Mode
57	ATST	I	Test input (pull high to put device into address test mode)
10	VDD	PWR	Power inputs
42	VDD	PWR	Power inputs
9	GND	PWR	Ground
26	GND	PWR	Ground
41	GND	PWR	Ground
58	GND	PWR	Ground

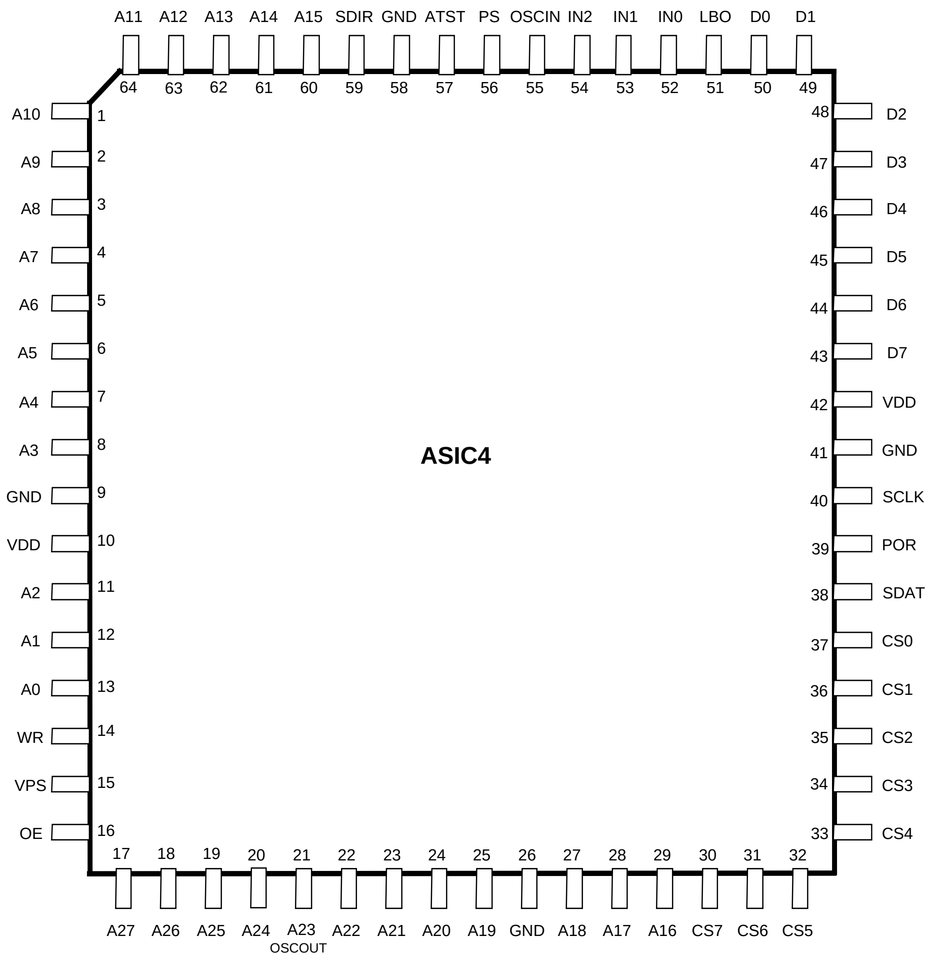
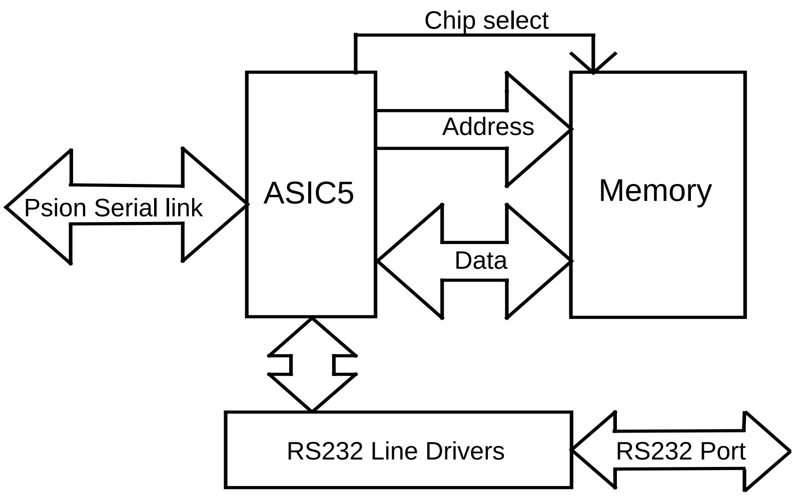


Diagram of ASIC4 Pin-out (NEC)

7. ASIC 5

What is ASIC 5?

ASIC5 is custom integrated circuit designed for use in Memory Packs (called SSDs) and peripheral devices. ASIC5 provides three primary functions. A built in UART provides for serial communication at baud rates of up to 48000 baud. General purpose I/O pins provide for a wide range of control and communication applications. Finally ASIC5 provides address and data lines for access to memory devices and memory mapped peripherals. A typical ASIC5 peripheral such as the 3Link consists of an ASIC5 connected to port C of a series 3/3a, a ROM memory device mapped into ASIC5's addressing space and line drivers to convert the UART signals from ASIC5 into standard RS232 levels.



ASIC5 Modes

ASIC5 can operate in two modes. In pack mode ASIC5 generates all the address, data, and control signals necessary to access memory devices. No peripheral functions are available in this mode. In peripheral mode ASIC5 has only limited memory address capabilities as some or all address and control lines are reused for I/O purposes. ASIC5 is placed into peripheral mode by setting the peripheral bit in ASIC5's PBMODE register.

ASIC5 as a UART

ASIC5 contains a full function UART which supports baud rates of up to 48000 bits per second. In order to use ASIC5 as a UART, ASIC5 must be placed into peripheral mode. In this mode the input signals PA0, PA1, PA2, PA3 become the UART inputs RX, CTS, DSR, and DCD respectively. The output signals PD0, PD1, PD2 become the UART signals TX, RTS, and DTR. Serial data is transmitted from the TX line. Incoming serial data is received by the RX line. RTS and DTR can be used for handshaking or as general purpose outputs and need to be set high or low explicitly by software. CTS, DSR, and DCD can be used for handshaking or as general purpose inputs.

The Psion SIBO serial link which connects ASIC5 to a host computer is a two wire interface consisting of a data and a clock line. ASIC5 generates it's baud rate clocks from this clock line. Under normal operation, clock pulses along a Psion SIBO serial link only accompany data frames. To be able to generate baud rate clocks ASIC5 requires a steady clock from the Hosts Psion port. To facilitate this, Psion serial links can be put into a mode called continuous clocking where clock pulses are generated regardless of whether there is any actual data to be transferred. Continuous clocking increases power consumption and Psion serial links should not be left in this mode unnecessarily. The UART portion of ASIC5 is capable of generating interrupts when characters are received, when the transmitter is awaiting a character to send and when there is a change of state on the handshaking lines. ASIC5 has only one, active high, interrupt line. This line is shared between these interrupt sources. When an interrupt is generated it is the responsibility of software to determine the cause of the interrupt.

A character to be transmitted should be written to the Transmitter Holding Register where ASIC5 will convert it to serial form for transmission. The register will be emptied once the character has been transmitted. ASIC5 contains no internal buffering. The Transmitter Holding Register must be empty before writing a character to it. The state of the Transmitter Holding Register is reflected in the Transmitter Empty bit in the UART status register. Enabling the Transmitting Holding Register interrupt will cause ASIC5 to generate an interrupt every time that the Transmitting Holding Register becomes empty. Reading the UART Status Register will clear the interrupt. Received characters are copied into the Receive Character Register. If the Receive Character Interrupt is enabled, ASIC5 generates an interrupt on each character received. If the character is received in error due either parity, framing or overrun errors, appropriate bits in the UART Status Register are set to reflect this.

ASIC5 for parallel I/O

Lines PA0-PA7 form a general purpose, non latched, 8 bit input/output port. All access to this port take a total of twelve clock cycles. The clock is generated from the Psion Serial Link. Actual memory access cycles only last for one clock cycle. Twelve cycles are required because data being sent or received needs to be converted to or from the Psion serial format. Because of the conversion read accesses occur on the third cycle, write cycles on the twelfth. This is usually of no real consequence to the peripheral designer

In peripheral mode CS0 will be taken low for one clock period each time Port A is accessed. Data outputted from this port will remain valid for only the period that CS0 is low. In pack mode Port A forms the data bus in memory mapped systems. A read from or write to Port A in this mode will result in one of the lines CS0-CS3, being taken low for one clock period. The line which will be taken low will depend on the address being generated for the access. During read cycles OE line will be taken low and remain so for 10 cycles. Data present on PA0-PA7 must remain stable for the last nine cycles. During write cycles the WR_B line will be taken low for the second half of the cycle over which one of CS0-CS3 is low. If port B is set to counter mode, accessing port A will result in the counter being incremented upon completion of the access.

ASIC5 can be programmed to generate an interrupt whenever the state of line PA4 changes. Reading port A will clear this interrupt. One use of PA4 is as the BUSY line in a Centronics port implementation.

The Lines PB0-PB7 can be programmed to operate in four different modes. Two of these modes are for testing purposes and will not be discussed further. In latched mode, data written to the Port B resister is latched onto the Lines PB0-PB7 and will remain there until a following write to Port B or a reset condition occurs. In counter mode, the binary value on lines PB0-PB7 is incremented following any access to port A. With ASIC5 in pack mode lines PB0-PB7 form the address lines A0-A7. Placing port B into counter mode allows 256 consecutive memory locations to be read without need to set-up the address of each access.

Lines PD0-PD7 are general purpose outputs. In pack mode these form the address lines A8-A15. In peripheral mode lines PD0, PD1, PD2 become UART outputs.

Lines PC0-PC4 in pack mode form the address lines A16-A20. In peripheral mode lines PC4 and PC7 become inverted inputs and can be used as edge triggered interrupt lines. PC5 becomes the interrupt output line. PC6 becomes a general purpose latched output. PC0-PC3 become a dual synchronous serial port for use in magnetic card systems.

ASIC5 for Barcodes

Psion barcode peripherals use the UART functionality of ASIC5 to receive data from a dedicated barcode scanner IC.

ASIC5 for Card readers

Magnetic card readers generate clocked serial data. ASIC5 contains two synchronous serial ports for connection to card readers or other peripherals which generate clocked serial data.

Reset and configuration

Following a reset or power up ASIC5 will read the form in which to configure itself from the data on data lines PA0-PA7. Line PC6 is used to select whether ASIC5 is to operate in pack or peripheral mode. The table below shows the meaning of each of these lines during reset. In peripheral mode lines PA0-PA7 should be set to give an indication of the type peripheral the ASIC5 is forming. combinations of types can be used.

PC6	**PA7**	**PA6**	**PA5**	**PA4**	**PA3**	**PA2**	**PA1**	**PA0**

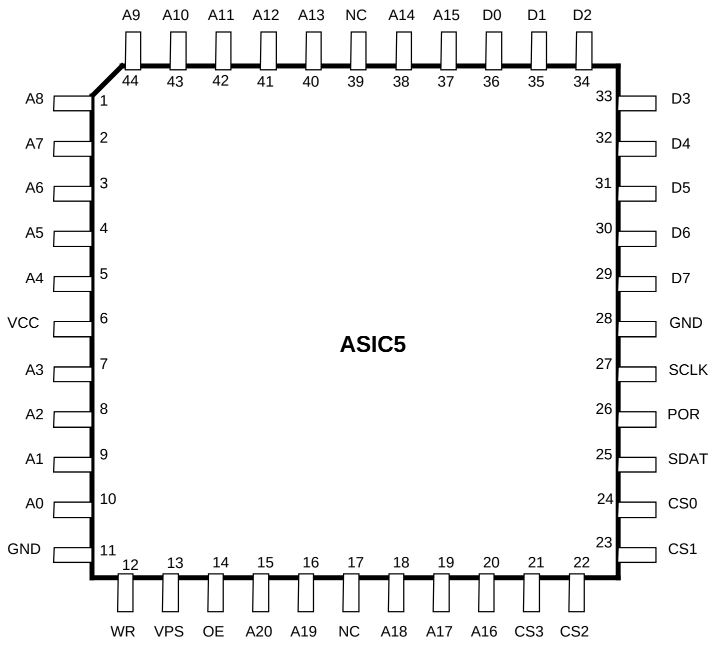
Pack Mode	Memory Type	No. devices	Memory Size
1	0 0 0 RAM	0 0 1 device	0 0 0 No memory
	0 0 1 Type 1 Flash	0 1 2	0 0 1 32Kbyte
	0 1 0 Type 2 Flash	1 0 3	0 1 0 64Kbyte
	1 1 0 ROM	1 1 4	0 1 1 128Kbyte
	1 1 1 Write protect		1 0 0 256Kbyte
			1 0 1 512Kbyte
			1 1 0 1Mbyte

Peripheral Mode	Type
0	X X 0 X X X X 1 RS232 port
	X X 0 X X X 1 X Centronics (Parallel) port
	X X 0 X X 1 X X
	X X 0 X 1 X X X
	X X 0 0 X X X X Barcode reader
	X X 0 1 X X X X USA modem
	X 1 0 X X X X X Modem
	1 X 0 X X X X X RS232 TTL

Pull up, pull down resisters would usually be used to place these lines in the desired state on reset. High value resisters of typically 100k would be used to allow ASIC5 and bus devices to drive these lines to other levels during normal operation.

ASIC5 should be powered from its host Series 3/3a, Workabout or HC. High current peripheral chips or volatile memories should have their own supply. Whenever the Series 3/3a/HC is powered down, has its batteries removed or has its pack doors open, any attached ASIC5 will be powered down and will be reset upon resumption of power.

ASIC5 Pin-out



ASIC 5 current pin-out (TI version CF30179)

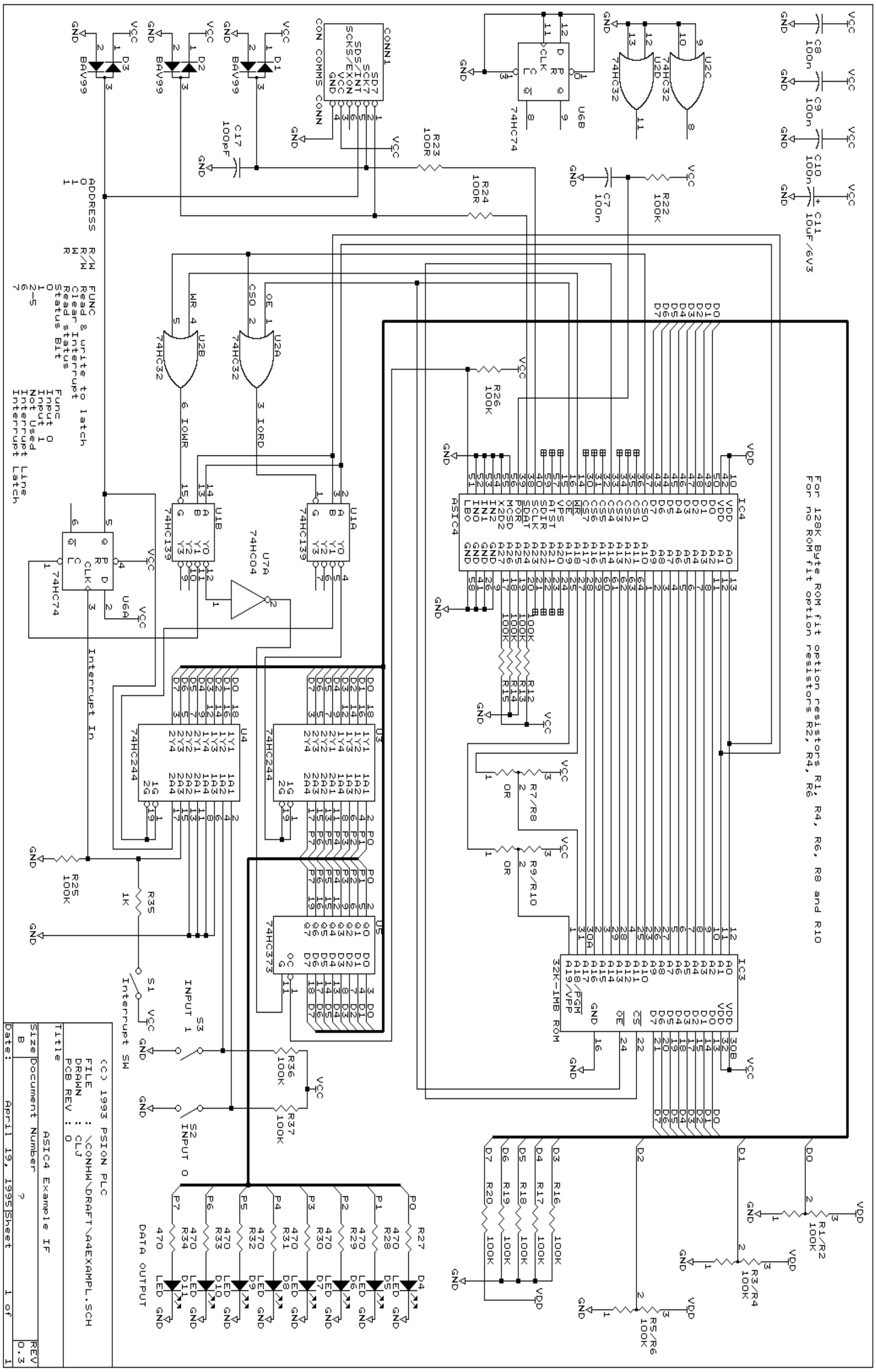
8. EXAMPLE PERIPHERALS

The ASIC4 Example Interface Board

The ASIC4 Example Interface Board detailed in this chapter is intended to provide the developer with a simple example of a Psion ASIC4 peripheral. To this end, the actual practical usefulness of the hardware is of secondary importance. In fact, the board essentially consists of eight LEDs connected via some latches to an ASIC4. Using a device driver, the host machine is able to control the status of the LEDs. Specifically, the board translates SIBO serial protocol signals into a parallel 8-way data bus format that can be used to set the various 74HC series latches and gates. It is intended that the board can be readily adapted to run on all currently available Psion machines. The only physical change that need be made concerns connection to the host machine's external expansion interface.

In the following circuit, an eight-bit tri-state data buffer, U3, and an eight-bit output data latch, U5, are commoned together to the eight LEDs thereby enabling their status to be sensed and set. In addition, a facility for generating hardware interrupts is provided by means of a suitably connected switch, S1, and a third eight-way buffer, U4. U4 is the interrupt switch status buffer. It holds the values of the two input switches S2 and S3 which are read as part of the interrupt service routine. Depending on the value of the switch positions, a different response can be output to the LEDs from the set buffer, U5. One of the D-type flip-flops in U6 is used to latch hardware interrupt signals into the INT line (pin 5) of the reduced external expansion interface connector. On completion of the interrupt routine code, it is necessary to reset this flip-flop and hence the interrupt hardware by means of a write to address A1. Address decoding is provided by two 2-to-4 decoders on chip U1 paralleled to address lines A0 and A1. A circuit diagram of the ASIC4 Example Interface Board is presented overleaf for the case of a host S3a.

Note that it is possible to construct this circuit with or without the compiled driver code in an on-board ROM. The developer merely has to set the resistors on the three data bus lines D0-D2 such that the corresponding info byte conveys the appropriate information. The meaning of the various bits in an ASIC4 info byte was discussed earlier in chapter 6. If a ROM is to be used, then the info byte read off D0-D7 on reset should include bits 001 on lines D2-D0. This requires the R1/R2 optional resistor to be connected to Vcc. If a ROM is *not* used, as was the case with the constructed test circuit, the data bus lines D2-D0 should be set to 000. This is done by choosing the resistors connected to ground from the three R1/R2, R3/R4 and R5/R6 pairs.


For 128K Byte ROM fit option resistors R1, R4, R6, R8 and R10
For no ROM fit option resistors R2, R4, R6
ADDRESS R/W FUNC
0 R/W Read & write to latch
1 W Clear Interrupt
1 R Read status
Status Bit Func
0 Input 0
1 Input 1
2-5 Not Used
6 Interrupt Line
7 Interrupt Latch
ASIC4
32K-1MB ROM
74HC244
74HC373
74HC139
74HC32
74HC74
74HC04
CONN1
CON COMMS CONN
DATA OUTPUT
INPUT 0
INPUT 1
Interrupt SW
Interrupt In
(C) 1993 PSION PLC
FILE : \CONNHW\DRAFT\A4EXAMPL.SCH
DRAWN : CLJ
PCB REV : 0
Title
ASIC4 Example IF
Size B
Document Number ?
REV 0.3
Date: April 19, 1995
Sheet 1 of 1

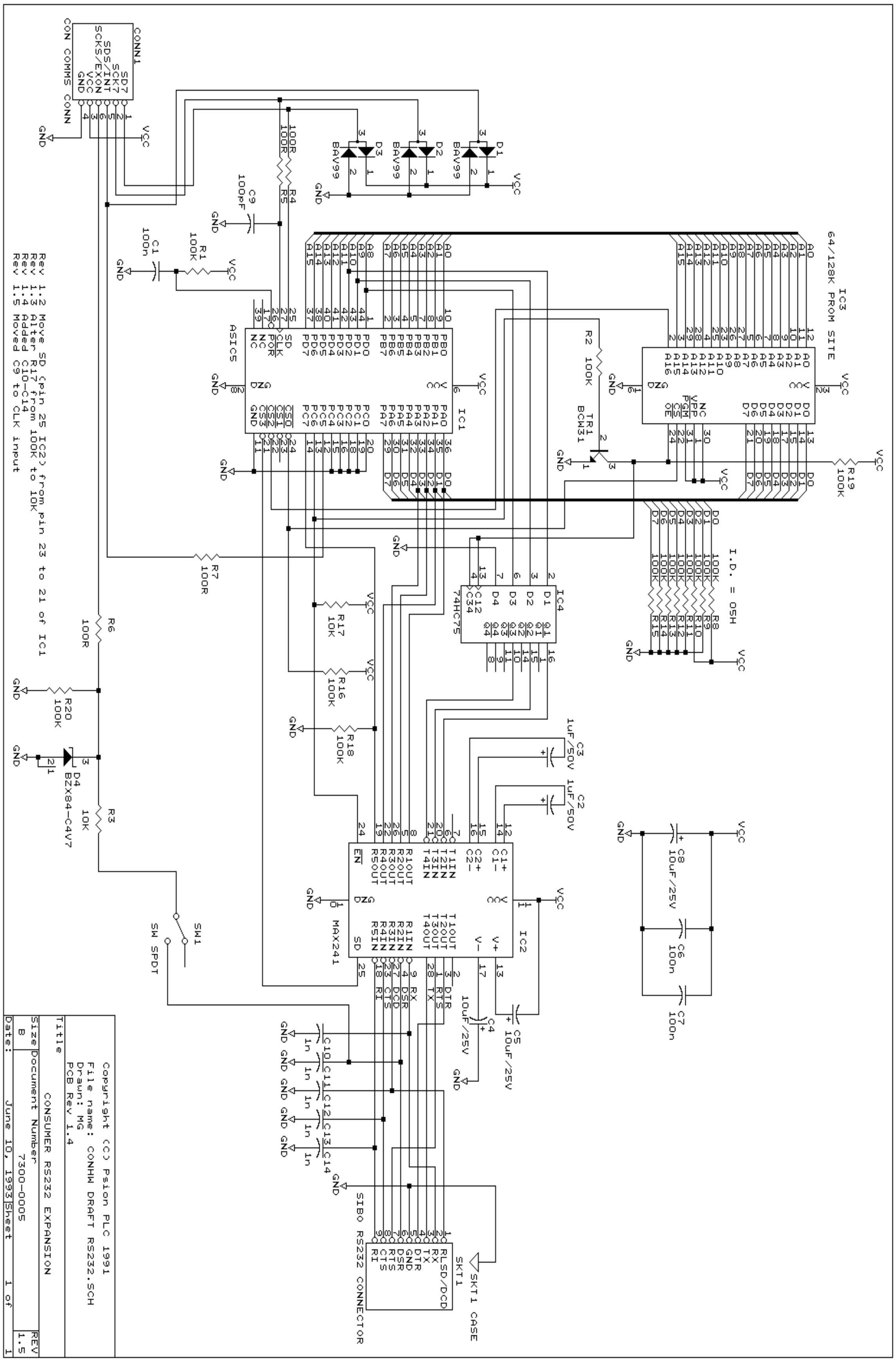
The Psion 3-Link

The Psion 3-Link is an ASIC5-based peripheral that enables the user to transform SIBO serial protocol signals into RS232 format data so that a host Psion machine can communicate with a PC or printer. The diagram below shows the schematic of a TTL level RS232 based expansion device. The interface is based around ASIC5 and allows the standard RS232 device driver contained in EPOC to be used.

Transistor Q1 provides a switched power rail for any other expansion device. This rail will automatically be switched off when the RS232 port is closed. D1 is required to isolate the supply so that the external device does not back power the SIBO computer when it enters the standby state. If the additional device does not require a supply or only uses a few microamps then Q1 and D1 can be omitted. Vcc and the supply for the expansion device can be directly connected to Vcc2.

If CMOS level RS232 signals are required then IC1 can be replaced with a 74HC244 device. If inverted sense RS232 signals are required, IC1 can be replaced with a 74HC241 device.

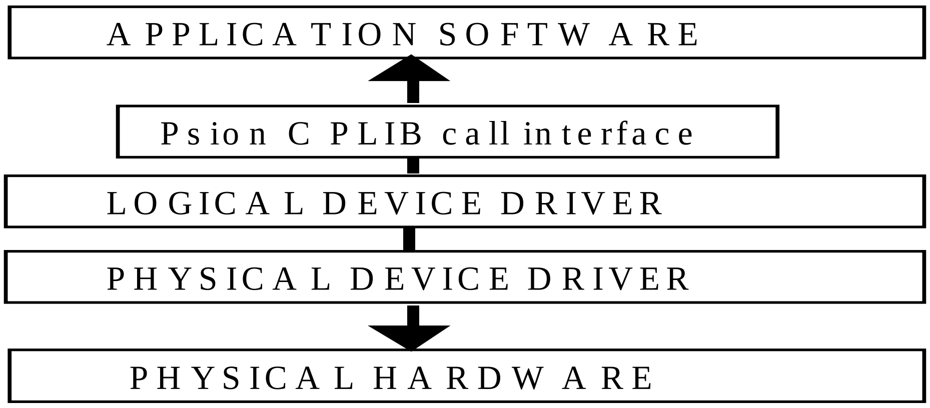
The diagram overleaf shows the circuitry for an ASIC5-based 3-link with ROM.


CONN1
CON COMMS CONN
64/128K PROM SITE
IC3
IC1
ASIC5
IC4
74HC75
IC2
MAX241
SKT1
SIBO RS232 CONNECTOR
SKT1 CASE
SW1
SW SPDT
I.D. = 05H
Rev 1.2 Move SD (pin 25 IC2) from pin 23 to 21 of IC1
Rev 1.3 Alter R17 from 100K to 10K
Rev 1.4 Added C10-C14
Rev 1.5 Moved C9 to CLK input
Copyright (C) Psion PLC 1991
File name: CONHM DRAFT RS232.SCH
Drawn: MG
PCB Rev 1.4
Title
CONSUMER RS232 EXPANSION
Size B
Document Number 7300-0005
REV 1.5
Date: June 10, 1993
Sheet 1 of 1

9. DEVICE DRIVER OVERVIEW

Introduction

Once a piece of peripheral hardware has been designed the appropriate software must be written to control it. All hardware on Series 3/3a, Workabout and HC machines is controlled by logical and physical device drivers. These act as the logical low-level software interface between a piece of hardware and an application that uses it. The remainder of this document is concerned with the methods by which a device driver is able to communicate with and thereby control an ASIC4 or ASIC5-based peripheral.



This chapter is intended to guide the programmer through the central issues involved in writing device drivers for peripherals that attach to the family of Psion host machines based around the proprietary SIxteen-Bit Organiser (SIBO) architecture. The purpose of a device driver is to abstract away the hardware details required to conduct communication between a peripheral and a software application that uses that peripheral. The device driver therefore performs the logical processing required to translate low level hardware instructions into high level application services. Psion device drivers are written in 8086 assembler and following convention are divided into a logical layer residing over a physical layer. A physical device driver (PDD) contains the code required for talking directly with the hardware device and provides a set of low level hardware specific services. A logical device driver (LDD) performs the logical processing that transforms these low level services into the high level services used by an application. This two-layer nature of device drivers at Psion can be illustrated by the following example. An application using the serial driver decides that it requires RTS/CTS handshaking. It calls an LDD which decides whether or not a line should be driven. If the answer is yes, the LDD calls the appropriate PDD and asks for a particular line to be driven to a specific state. The PDD duly carries out the requested service. Psion SIBO machines often use the same LDD with a PDD written specifically for each version of the hardware device. In such a situation, splitting the device driver is highly desirable. In the example given above, however, the LDD could have talked directly with the hardware negating the requirement for a separate PDD. Similarly, most external peripherals would normally use an LDD.

An LDD must provide a minimum of eight functions for use by the operating system. The functions are passed to the OS via a table of function offsets (referred to as the vector function table). These functions are mandatory. Similarly, a PDD must provide two functions for use by the operating system and may provide more if required. An LDD will usually provide further services/functions for use by an application. The form these take is dependent on the LDD requirements and the functions supplied by the associated PDD(s).

Psion SIBO machines are supplied with a set of resident device drivers built into the ROM each of which can be replaced with an installable device driver having the same name. Installable device drivers can also be added to increase the number of available device drivers. Installing a device driver is carried out dynamically without resetting the machine (this is not the case with many operating systems).

Device Names and Channels

The name of a device driver is the mechanism by which an application can obtain a channel to that device driver. A logical device driver name always has three characters followed by a colon. For example, "`TTY:`" is the serial LDD. This name is required to uniquely identify the LDD to the EPOC OS when attempting to open a channel on it. A physical device driver name always has the three characters of its owning LDD followed by a period, a further three characters and a colon. For example "`TTY.UAR`" is the ASIC5 UART driver and "`TTY.SRX`" is the 16450/16550 driver. The first three characters of a PDD name are the name of the LDD to which the PDD belongs. The second set of three characters uniquely identify the PDD. Thus in the above examples, both PDDs belong to the "`TTY:`" LDD.

A channel can be opened on an LDD by calling the PLIB library function `p_open`. EPOC uses the driver name passed through an application `p_open` call to invoke the `IoOpen` operating system service. This service in turn calls the associated driver 'open vector' which can decide whether or not to open a channel on the driver. The assembled ASIC4 Example Interface Board logical device driver, A4EXIF.LDD, for example, has the three-character device name "`LED`" so a channel with its handle in `pcb` may be obtained on it by means of the following call:

```
p_open(&pcb, "LED:", -1)
```

In this call, the third argument refers to the open mode and a value of -1 indicates that the mode parameter is to be ignored. To obtain a channel on a PDD, an application should call the `DevOpenPDD` OS service. Typically, only LDDs open PDDs though the `p_open` library function can be used to open a PDD indirectly as illustrated in the following example:

```
p_open(&pcb, "TTY.UAR:", -1)
```

For a device driver configuration consisting of an LDD and a PDD, the application will usually open a channel to the LDD only: the LDD as part of its initialisation would open a channel to the required PDD. If an LDD requires a PDD and none is specified, it is up to the LDD to either fail the open request or hunt for a loaded PDD that it can use. An LDD uses the `DevFind` OS service to search for a PDD as for instance in the case of the "`TTY:`" device.

A device driver may be capable of supporting more than one open expansion channel at a time. In order to distinguish the channels, a qualifier can be added to the open request as part of the device name. It is then up to the device driver to specify the format of the qualifier. By convention, channels are allocated a single character sequentially from the character 'A'. For example, the parallel port driver can support two open channel, 'A' and 'B'. The LDD requires one of these qualifiers in order to open a parallel driver channel:

```
p_open(&pcb, "PAR:A", -1)
p_open(&pcb, "PAR:B", -1)
```

The number of channels that can be thus supported will in general be dependent on the host SIBO hardware. In the case of the serial port on the S3a, for instance, only one SIBO channel can legitimately be opened corresponding to expansion port C. With the Workabout and HC, however, it is possible to open up to three separate SIBO channels on ports A through to C where A refers to the top port of the host machine, B to the bottom port and C to the side (or cradle) port.

LDDs have been designed to be accessed via the I/O system. I/O requests on the opened channel will reach the 'strategy vector' of the device driver. PDDs have been designed to be accessed by an LDD either via far calls or the `DevVector` OS system service. Once a channel has been obtained on a device driver, the operating system can send it events not sent to other applications. Examples are events generated by the machine being switched on or off, memory segments being moved about and the owning application being panicked. The EPOC OS can handle a maximum of 32 device drivers on a Series3 machine and 48 on other machines.

Loadable Logical Device Driver Structure

Common features of the structure of loadable LDDs

All loadable LDDs must conform to the following rules:

- There must be a single code segment and no data segments. The code segment is encapsulated in the assembler .asm file by calls to the `CodeSeg` and `EndCodeSeg` defines respectively.
- The code segment must begin with a LibEnt structure which indicates the LDD name or signature which is used to identify the driver when trying to open and close channels.
- There must be at least eight supported functions which are listed in the LibEnt struc.

LibEnt Structure

The first field of the LibEnt structure consists of a two-byte signature containing the define '`LDDSignature`' or '`PDDSignature`' in the case of a PDD. The remaining fields consist of an eight byte name which holds the device driver name stored as a zero terminated string (note that the trailing colon is omitted), a two byte vector count which must be at least eight and a vector table listing the supported device driver functions. The relevant code in the case of a hypothetical DevFunc LDD (DEVFUNC.LDD) with the device name of "`TES:`" is listed below:

```
        CodeSeg

        ProcBegin@ DevFuncLDD

        dw      LDDSignature
        db      'TES',0,0,0,0,0
        dw      (VectorEnd-Vector)/2
Vector:
        dw      DevFuncInstall
        dw      DevFuncRemove
        dw      DevFuncHold
        dw      DevFuncResume              ; Mandatory LDD vectors
        dw      DevFuncReset               ; must be in this order
        dw      DevFuncUnits
        dw      DevFuncOpen
        dw      DevFuncStrategy
VectorHandler:
        dw      DevFuncHandler             ; Optional LDD vectors
InterruptVectors:
        dw      DevFuncTickInt
VectorEnd:
        ProcEnd noret
```

The vector table contains the offsets within the device drivers code segment for the functions required by the EPOC operating system which must be entered in the order shown. Note that in this document, the terms vector and function are used interchangeably.

Mandatory LDD Functions

All LDDs must support the following eight functions:

• `DevFuncInstall`	called on device installation
• `DevFuncRemove`	called on device removal
• `DevFuncHold`	called to temporarily disable the driver
• `DevFuncResume`	called to enable the driver after it has been temporarily disabled

• `DevFuncReset`	called when an application terminates without closing the channel
• `DevFuncUnits`	called to query the number of supported units (i.e. channels)
• `DevFuncOpen`	called to open a channel to an LDD
• `DevFuncStrategy`	called to access the driver's functionality from the I/O system

All of the eight mandatory routines pointed at by the function vector table will be called *FAR* by the operating system and should therefore use a *FAR* return machine code instruction to return control back to the OS. Since the *FAR* return address is to the OS it does not matter if the OS moves memory whilst code in the LDD is being executed: the OS cannot move its own code.

DevFuncInstall

This function is called by the operating system when the device driver is loaded in order to initialise any internal variables. It should not be called directly by an application process. The `DevInstall` operating system service will cause this function to be called. Applications should not call this service directly and should call instead the `DevLoadLDD` service.

An installable device driver may have the same name as a resident device driver. When the operating system loads a device driver, it places it at the end of the device driver table. The operating system will search this table for the appropriate device driver when it wishes to establish a channel. The search starts at the end and thus will locate the most recently installed device driver (if any) or if not, the resident driver. By this mechanism an installable driver can replace any resident driver.
When called, the DS and ES segment registers are in an unknown state. The device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register or more commonly just using the CS override. The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored. All operating system services may be called, except those concerning file or device access.
PASSED
No values are passed to the install vector.
RETURN
If the installation was successful, return with the carry flag clear.
If the installation failed, return with the carry flag set and the error number in the AL register.
PANIC
The install vector must not panic: it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the install function.

DevFuncRemove

This function will be called by the operating system when the device driver is requested to be unloaded. It should not be called directly by an application process. The `DevRemove` operating system service will cause this function to be called. Applications should not call this directly, they should use the `DevDelete` service. Before the remove function is requested, the device driver will have received a hold request. Thus devices will only ever be removed when in a held state. If the device driver is currently busy serving a client, the remove request should return an error. Note that all resident device drivers will return an error since there is no mechanism by which they can be re-installed.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register. The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored. All operating system services may be called inside the remove vector, except those concerning file or device access.
PASSED
No values are passed to the remove vector.
RETURN
If the remove was successful, return with the carry flag clear.
If the remove failed, return with the carry flag set and the error number in the AL register.

PANIC
The remove vector must not panic: it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the remove vector.

DevFuncHold

This vector will be called by the operating system when a logical device driver is requested to be held. The hold vector is called in the context of the operating system so DS and ES are not available. The `DevHold` operating system service will cause this vector to be called. Applications should not call this service. Physical device drivers cannot invoke holds and resumes. The operating system will call the hold vector under three conditions:

- Device memory segments are about to be moved.
- The machine is about to switch off due to the auto switch off time-out or user request, it enters the standby state.
- The machine is about to switch off due to the power source being removed.

It should be noted that holds and resumes are called on a per driver basis and so the corresponding driver code must deal with all the currently open channels. In all cases the device driver must respond to the request as quickly as possible. It must also ensure that ALL interrupts from the hardware device that it is driving are disabled. Device memory segments can only be moved if an installable device driver is being installed or removed. If the LDD uses an attached PDD and uses the faster *FAR* call mechanism to call the PDD strategy vector, the PDD strategy vector address will potentially move, thus the *FAR* address will be wrong. This address can be resolved in the resume vector. The LDD must not call the PDD between a hold and resume. Typically, the device driver only needs to disable its interrupts. When a resume occurs, the device driver should continue as though nothing had happened.

If the machine is about to switch off due to the auto switch off or user request mechanisms (enter the standby state), the device driver should make an orderly shut down of the device such that the state before the shut down can be recovered when the system powers up again. The device driver should also attempt to ensure that no data is lost. For example, in the serial driver the current state of the hardware handshaking lines should be noted so that each state can be restored on power up. For this type of power down the hold vector is allowed to take a significant length of time to shut down a device. For example in a serial driver the hold vector should wait until the remote end stops transmitting data after any hardware handshaking has been applied. Of course, the time taken should be kept to a minimum: in the case of the serial driver above the time is roughly equivalent to 3 character transmission times. When a resume occurs the device driver should continue as though nothing had happened.

If the machine is about to switch off due to the power source being removed, the device driver should reset the device in the minimum possible time: no attempt should be made to perform an orderly shut-down. The device driver is not expected to be able to recover the hardware state. When a resume occurs, the device driver would typically fail any outstanding application requests. If the hold vector takes too long the voltage will fall below the threshold to hold the state of the internal RAM. If this occurs the machine will perform a warm re-boot when powering up, all data in the internal memory of the machine will be lost including the device driver code! On power fail there is about 2ms available to power down all devices.

On a power failure hold, the operating system will already have sent a 'reset' to all the SIBO serial channels. Any device drivers using these channels need only record the hold reason for the resume vector. Any other peripherals should be designed to allow a power fail mechanism with the minimum amount of code. It *must* be noted that the power fail type hold can occur whilst the device driver is in the memory move hold state. In this case, the device driver will receive two hold requests before seeing a resume request. A device driver must be capable of handling this. In this case, the device driver will also receive two resume requests. A device driver will not get a power fail hold whilst in power down hold. A call to the hold vector will always be followed by a call to the resume vector (except when a device is requested to be removed).

Note that with the Series 3a and Workabout, there is an additional case when hold and resume must be invoked and that is on opening/closing of the pack doors. In this case, the LDD must generate its own Hold and Resume. This situation is examined in more depth in the context of the specific example drivers presented later in this document.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register. The device driver should not call any operating system services in the hold vector code due to the time taken, especially on power failure.
PASSED
The AH register takes one of the following

• `DevHoldNormal`	Device memory is about to be moved.
• `DevHoldPowerDown`	The system is about to enter the standby state.
• `DevHoldPowerFail`	The system has lost its power supply.

RETURN
None.
PANIC
The hold vector must not panic: it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the hold vector.

DevFuncResume

This vector will be called by the operating system when the device driver is requested to be resumed. The resume vector is called in the context of the operating system. The `DevResume` operating system service will cause this vector to be called. Applications should not call this service. The resume vector will be called either when memory has finished being moved or when the machine powers back up. In both cases the hold vector will have been called before this vector is called. The device driver is expected to recover from the previous hold request (except power fail) and resume any I/O that was suspended. If the device driver has an interrupt service routine, it should reset the interrupt service routine's address since the device driver may have moved in memory; its absolute segment address will be different.

If the hold was a device memory segment move type hold, interrupts should be re-enabled. If the LDD uses an attached PDD and uses the *FAR* call mechanism to access the PDD strategy vector, the address of the PDD should be reset by using the `DevGetPDDAddress` operating system service before enabling interrupts. Typically, the PDD will have a call back to the LDD and it needs to be informed of the change of address of the LDD call back function, the LDD-PDD interface definition should allow such a function request.

If the hold was a power down type hold, the resume vector needs to power up the peripheral and set it to the state that it was in before the power down occurred. If this is not possible or data has been lost, the device driver should inform any outstanding requests of this fact. It is also possible that the hardware device that the driver is associated with has been removed. The driver should be able to handle this properly. If the device driver is expected to generate events due to an external state change, the driver should check the external state and generate appropriate events. For example, the serial driver may be requested to inform an application when the DTR line changes state. The remote end may have changed the state of DTR whilst the driver is held.

If the hold was a power failure type hold, the resume vector should power up the peripheral and put it into a known state, preferably the state that the application software thinks that the device is in and fail any outstanding requests as data is quite likely to have been lost.

When called, the DS and ES segment registers are in an unknown state. The device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register. All operating system services may be called, except those concerning file or device access.
PASSED
None
RETURN
None.
PANIC
The resume vector must not panic, it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the resume vector.

DevFuncReset

This function will be called by the operating system when the device driver is requested to reset a channel. The reset function is called in the context of the operating system so DS and ES are not available. The device driver must request that the operating system call the reset function. This is achieved by calling the `IoRequestReset` system service, usually in the open vector. To cancel this request, the device driver should call the `IoRequestResetCancel` system service. The cancel service is usually called as part of the close functionality in the strategy vector. The reset vector will be called when the operating system is tidying up resources owned by a process that has terminated. If a process terminated before it closed the device driver channel and no reset service is requested, that channel would remain allocated; no process will ever close the channel. The reset vector allows a device driver to reset itself and allow the channel to be opened again. Any data required to perform the reset must be stored in the device driver. The data space belonging to the process that originally opened the channel has been returned to the operating system memory pool and is no longer valid. If a device driver can handle multiple channels then the data passed to the `IoRequestReset` system service should identify the channel. This data will be passed in the CX register to the reset vector. The device driver should only have a reset request outstanding with the operating system while a process has a channel open.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register or using CS override. All operating system services may be called, except those concerning file or device access.
PASSED
This function is passed data in the CX register that the device driver requested it be sent to determine which channel should be reset.
RETURN
None.
PANIC
The reset vector must not panic; it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the reset vector.

DevFuncUnits

This function will be called by the operating system when the device driver is requested to report the number of units (i.e. channels) the device driver can support. This function is called in the context of the operating system. The operating system places no significance on the number of channels a device driver can support. It is primarily used for informational purposes. An application may use the number of units to attempt to open any available channel on that device driver.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register. All operating system services may be called, except those concerning file or device access.
PASSED

None.

RETURN
The AX register should contain the number of channels supported. If a device driver can support multiple channels (limited only by memory constraints) then the driver may return -1. A serial device driver, for example, might only support two channels (`TTY:A` and `TTY:B`) whereas the file device driver can open an unlimited number of files.
PANIC
The channels units vector must not panic; it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the units vector.

DevFuncOpen

This function will be called by the operating system when a channel to the device driver is required to be opened. This function is called in the context of the process that called the `IoOpen` system service. This means that DS and ES point to the application data space. The device driver is passed two parameters, its device handle and a pointer to an `OpenEnt` structure. The device handle is the entry in the system device table of this device driver. The device driver is required to place this handle in the `ChanLibHandle` field of the `ChanEnt` structure which must be allocated in the user's data space. The operating system uses the device handle to route any I/O requests on the opened channel to the correct device driver.

The `OpenEnt` structure contains three fields, `OpenNamePtr`, `OpenMode` and `OpenChan`.
The `OpenNamePtr` field contains a pointer to the character that exists after the device name as passed to the `IoOpen` system service. For example, if the `IoOpen` service was passed a name of `PAR:A`, the `OpenNamePtr` field would point to the colon. If the `IoOpen` service was passed a name of `TTY.AS5:B` the `OpenNamePtr` field would point to the full stop. The device driver should process the name appropriately, opening the correct PDD as required.
The `OpenMode` field contains the mode for opening the device driver. The available modes are specified by the device driver writers. For example, a combined Xmodem and Ymodem device driver could use the mode to specify whether the Xmodem or the Ymodem protocol is to be used.
The `OpenChan` field contains the I/O channel handle of the device that this driver is required to 'attach' to. Attached device drivers are dealt with later in the chapter.
The code in a device driver open vector tends to follow a very similar pattern. This is demonstrated by the following code fragments and associated comments. The first stage is to allocate some data space in the calling process' heap space. This will contain the I/O channel control block:
```
    mov       cx, (size DeviceEnt)
    HeapAllocateCell
    jc        noMemory
    mov       bx, ax                      ; cell handle
```

If the device driver requires a WaitHandler (described later):
```
    mov       al, (VectorHandler-Vector)/2
    IoAddHandler
    jc        endFreeMemory
    mov       [bx].DriverHandler, ax
```

If the device driver's `DevFuncReset` vector is required to be called:
```
    push              bx
    mov       cx, ChannelIndicator        ; unique per channel
    mov       bx, dx                      ; the device handle
    IoRequestReset
    pop       bx                          ; restore alloc cell
```

The `ChanEnt` field of the `DriverEnt` structure must be initialised:
```
    mov       [bx].DriverIo.ChanNext, bx
    mov       [bx].DriverIo.ChanSignature, IoChanSignature
    mov       [bx].DriverIo.ChanLibHandle, dx
```

The `ChanNext` field is used by attached drivers and will usually be set to be the allocated cell handle of the device driver being opened. The `IoFuncAttach` and `IoFuncDetach` functions manipulate these fields. The I/O system uses this field to direct the I/O request to the correct driver.

The `ChanSignature` field is checked by the operating system during any I/O requests for the value `IoChanSignature`. If it does not contain that value, the process calling the I/O service will be panicked for having passed an invalid I/O channel handle.
The `ChanLibHandle` field is used by the operating system to route an application's I/O request to this device. The I/O request will call the `DevFuncStrategy` vector of the device driver.
If the driver is an attached driver the following is required:

```
    mov cx, bx              ; allocated channel
    mov bx, [si].OpenChan   ; channel attaching to
    mov al, IoFuncAttach            ; return in BX the
    IoWithWait              ; channel attached to
```

Finally, if the channel has been successfully opened:

```
    clc                     ; Opened Ok
    ret                     ; return BX and DX
```

The error recovery code typically follows the following pattern:

```
    endFreeReset:
        push   ax
        push   bx
        mov    cx, ChannelIndicator
        mov    bx, dx
        IoRequestResetCancel
        pop    bx
        pop    ax
    endFreeHandler:
        push   ax
        push   bx
        mov    bx, [bx].DriverHandler
        IoRemoveHandler
        pop    bx
        pop    ax
    endFreeMemory:
        push   ax
        HeapFreeCell
        pop    ax
        stc
    noMemory:
        ret
```

If a device driver supports a fixed number of channels, it typically contains static control blocks. In order to determine if a requested channel is currently open, a field should be interrogated. The device driver should ensure that interrupts are disabled during this sort of check since a context switch could occur and another process request the opening of the same channel. This is the classic 'test and set' problem encountered in multi-tasking environments.

When called, the DS and ES segment registers point to the data segment of the application process attempting to open a device channel. The application should ensure that the DS and ES segment registers do in fact point to its data segment. The device driver must obey the normal rules concerning segment register manipulation. The DS and ES segment registers can be reloaded if required from the `IntEnt` structure pointed at by the BP register. All operating system services may be called.

PASSED

DX contains the device handle of the device driver.
SI is a pointer to the `OpenEnt` structure
BP is a pointer to the `IntEnt` structure.

RETURN

If the channel open was successful, return with the carry flag clear and the BX register containing the open channel.
If the open failed, return with the carry flag set and the error number in the AL register.

PANIC

The open vector can panic; it will cause the process requesting the device open to terminate. It is however more usual to return an error to the calling process.

PRESERVE

The DS, ES, SS, SP, BP and DX registers must be preserved by the open vector.

DevFuncStrategy

When an application makes an I/O request on the opened device driver channel the request is routed to this vector by the operating system. A device driver defines the set of functions that it supports. These typically include `IoFuncSet`, `IoFuncSense`, `IoFuncRead`, `IoFuncWrite` and `IoFuncClose`. A device driver does not have to support any particular function, as it is a matter of design between a device driver writer and application writer as to what functions and associated parameters are provided. To obtain the power of attached device drivers, however, it is recommended that the device driver use the system defines with their appropriate functionality, for example, the `IoFuncWrite` function number should always be associated with writing data.

The strategy function is passed the channel handle as allocated in the open vector in the BX register. This typically contains control information concerning the current state of the I/O channel.
The SI register contains a pointer to a `RqEnt` structure. This structure contains four fields, `RqFunction`, `RqStatusPtr`, `RqA1Ptr` and `RqA2Ptr`.
The `RqFunction` field contains the function number as passed to the `IoWithWait` (or `IoAsynchronous`) I/O request by the application. If a device driver does not support the specified function, it should pass the request on to its 'parent' device driver.
The `RqStatus` pointer contains a pointer to a memory location in the application process's data space that receives the I/O requests completion status. The device driver must set this memory location to the value `PendingErr` whilst the I/O request is outstanding and a completion code when the I/O request completes. An I/O request may complete within the strategy vector or it may complete some time in the future, presumably from some interrupt.
The `RqA1Ptr` and `RqA2Ptr` fields contain the argument 1 and 2 parameters as passed to the `IoWithWait` (or `IoAsynchronous`) system services. The device driver is free to specify what these parameters are (if any).

The operating system defines a set of common function numbers used by device drivers referred to as the `IoFuncXXX` set of defines. By convention, a device driver should select from this list, particularly if some of the more advanced features of the I/O system are to be used, such as attached device drivers. The more common defines are listed below:

• `IoFuncRead`	; read from the device.
• `IoFuncWrite`	; write to the device.
• `IoFuncClose`	; close device channel.
• `IoFuncCancel`	; cancel an I/O request.
• `IoFuncSet`	; set driver characteristics.
• `IoFuncSense`	; sense driver characteristics.
• `IoFuncFlush`	; flush any buffers.

The PLIB library functions `p_read`, `p_write` and `p_close` will call the device driver with the `IoFuncRead`, `IoFuncWrite` and `IoFuncClose` function numbers. Thus, if the device driver chooses an alternative function number set, an application will not be able to use the supplied library functions. All resident device drivers obey the following conventions:

- A cancel request will cancel any outstanding requests. A cancel request will not return any error.
- A close request will ensure that any outstanding requests are completed before closing the channel. A close request will not return any error.
- Only one request of a particular type can be outstanding at any one time. If a second request is made the device driver will panic the calling application.

Any functions that the strategy function does not support should be passed on to the next driver down the driver hierarchy. If the driver is a root driver (attached driver), this is achieved using the `IoRoot`

(`IoSuper`)system service. If the requested function is not supported by any driver, the operating system will return a `NotSupported` error.

When called, the DS and ES segment registers point to the data segment of the application process making the I/O function request. The application should ensure that the DS and ES segment registers do in fact point to its data segment. The device driver must obey the normal rules concerning segment register manipulation. The DS and ES segment registers can be reloaded if required from the `IntEnt` structure pointed at by the BP register. All operating system services may be called.

PASSED

BX contains the allocated channel control block.
DX contains the device handle of the device driver.
SI is a pointer to the `RqEnt` structure.
BP is a pointer to the `IntEnt` structure.

RETURN

If the function request is successful, the strategy vector should return with carry clear. A request typically causes some I/O. If the I/O is completed by the strategy vector (i.e. the request is for a synchronous function such as close), the completion status should be written back to the `RqStatusPtr` location and the I/O semaphore signalled (using the `IoSignal` system service). If the request has not yet completed (i.e. the request is for an asynchronous function), the `RqStatusPtr` location should contain the value `PendingErr` and the I/O semaphore should *not* be signalled.
If the function request failed the strategy vector should return with carry set and the error code in AL. In this case, typically no I/O requests will be completed.

PANIC

The strategy vector can panic; it will cause the process making the I/O request to terminate. In most cases it is usual to return an error to the calling process. A major exception to this is if the calling process makes an I/O request of the same type as one that is currently outstanding and the device driver only supports one I/O request of a particular type at a time; by convention the device driver should panic the calling process with the `PanicIoPending` panic code.

PRESERVE

The DS, ES, SS, SP and BP registers must be preserved by the strategy vector.

Interrupts and Interrupt Service Routines

Device drivers that talk to hardware tend to have interrupt service routines associated with them, especially if they are receiving data from an external source. The EPOC operating system provides a framework within which an interrupt service routine can be written relatively easily. An interrupt service routine is a code section that is called by the OS in response to a particular hardware event. As indicated in the Hardware Overview, the SIBO architecture allows for eight independent hardware interrupt sources, some of which are pre-allocated to system components. The operating system provides the `GenSetRevector` service to allow a device driver to install an interrupt service routine for any of the eight hardware interrupt sources. This call passes the interrupt vector (the address of the interrupt service routine) and the interrupt number (which is dependent on the host hardware) to the OS so that it knows where to jump to when the interrupt occurs. A device driver should use this system service and not poke directly into the 8086 interrupt vector table. The address passed to the `GenSetRevector` service is not written into the interrupt vector table but to an internal table. After invoking this service, the desired interrupts must be masked in by writing the appropriate mask to the mask register. Initially, of the eight interrupt sources, only the tick interrupt is masked in.

When an interrupt occurs, the microprocessor could be running any currently active process. The OS handles the servicing of interrupts by building a mandatory operating system call frame. All CPU registers are preserved on route. The interrupt service routine is then called as a *FAR* routine. Since the operating system preserves all registers the interrupt service routine is free to use any register. As with all interrupt service routines various rules apply:

- Interrupt service routines should execute as fast as possible. Operating system interrupt service routines are tuned to last no longer than one millisecond.

- Typically, interrupt service routines do not enable interrupts unless the routine can handle re-entrancy.
- Interrupt service routines run in the context of whatever process is running at the time of the interrupt. An interrupt service routine should not attempt to obtain admissibility to the process that opened the channel but access the internal driver space only which in general is its own code space.
- An interrupt service routine must not directly cause the OS to reschedule the running process as this would significantly delay its completion. It must use the IoSignalByPidNoReSched system service in order to indicate to the handler that an event has occurred to the owning process. The handler function of the device driver must pick up the event and inform the owning process.
- An interrupt service routine should return with the carry flag clear if it requires a reschedule to occur (it has called IoSignalByPidNoReSched) otherwise return with the carry flag set. This will cause the operating system to reschedule if the internal state allows such an action otherwise the reschedule request is effectively queued until such time that the operating system can reschedule.

At some stage during the course of an interrupt service routine, it is necessary to clear the interrupt line with some hardware-specific action. Then the interrupt controller inside the host ASIC1 or ASIC9 has to be cleared with a write to the NonSpecificEoi location. To remove the interrupt service routine address, the operating system service `GenResetRevector` should be used. This will reset the internal table entry to the default held in the ROM. Additionally, the interrupt mask should be reset to the original value.

Device Driver I/O Semaphore Wait Handlers

An LDD may nominate one of its functions to be called by the operating system every time the I/O semaphore of the process that opened the channel is signalled. The nominated function, known as the wait handler, will only be called if the application is waiting for an outstanding I/O request to complete. For well written applications this is practically all the time. By convention the vector table entry after the mandatory vectors contains the handler vector. A handler routine is similar to an interrupt service routine in that it appears to run 'from nowhere'. Comparing handlers and interrupt services routines shows that:

- A handler will always run in the context of the process that has opened a channel. An interrupt service routine will run in the context of whatever process happens to be running at the time of the interrupt.
- A handler can access the data space of the process that opened the channel. The interrupt service routine must not. An interrupt service routine should only access the data space in the driver which is usually its own CS space.
- A handler can cause a reschedule. An interrupt service routine must not cause a reschedule. If it did, the interrupt would not be fully serviced (the rest of the interrupt service routine would not be executed until a reschedule back to the process running at the time of the interrupt, which may not happen for a significant length of time). The interrupt service routine must only use the `IoSignalByPidNoReSched` to signal the channel owner.

The handler is the mechanism by which hardware interrupt events can be filtered through to the process using the I/O channel. Typically, it is in the handler code that the `IoSignal` signifying completion of an asynchronous I/O request is invoked.

Loadable Physical Device Driver Structure

A loadable PDD must obey the following rules:

- There must be a single code segment and no data segments.

- The code segment must start with a `LibEnt` structure.
- There must be at least two supported functions, with typically a further two defined.

Single Code Segment

A PDD must be written to contain any internal variables within its own code segment. Typically, these variables are only concerned with unit (i.e. channel) allocation and hardware state. Data space for a particular open channel can be allocated in the heap space of the process that opens the device. This data space will however disappear if the process terminates, thus any variables required for 'freeing' the hardware after a process terminates must exist in the code space of the device driver.

The LibEnt Structure

A `LibEnt` structure has the following format:

- A two byte signature
- An eight byte name
- A two byte vector count
- A vector table

The two byte signature should contain the 'PDDSignature' define. The eight byte name contains a zero terminated name, being that of the device driver. Note that there is no trailing colon. The two byte vector count contains the number of vectors that follow immediately after the count. There should be at least two. For example:

```
       dw     PDDSignature         ; Its an PDD driver
       db     'DVR.HW1',0          ; Name of the driver
       dw     (VectorEnd-Vector)/2 ; Number of vectors
   Vector:
       dw     DvrInstall           ; Install vector
       dw     DvrRemove            ; Remove vector
   VectorEnd:
```

Most PDDs also define a further two vectors:

```
       dw     DvrOpen              ; Open Vector
       dw     DvrStrategy          ; Strategy vector
```

The table of vectors is a table of offsets within the device drivers code segment of the routines that implement the required functionality. The vector table must have the entries in the order shown in the example.

Mandatory PDD functions

All PDDs must support the following two functions:

- `DevFuncInstallPDD` called on device installation.
- `DevFuncRemovePDD` called on device removal.

Most PDDs will support the following two additional functions:

- `DevFuncOpenPDD` called to open a PDD.
- `DevFuncStrategyPDD` called to provide PDD functionality.

All of the routines pointed at by the function vector table will be called *FAR* by the operating system and should consequently use a *FAR* return machine code instruction to return back to the operating system. Since the *FAR* return address is to the operating system, it does not matter if the operating system moves memory whilst code in the LDD is being executed; the operating system cannot move.

DevFuncInstallPDD

This vector will be called by the operating system when the device driver is loaded to initialise any of its internal variables. The install vector is called in the context of the operating system and not the process that is loading the device driver. The `DevInstall` operating system service will cause this vector to be called. Applications should not call this directly, they should use the `DevLoadPDD` service.

An installable device driver may have the same name as a currently installed device driver. When installed, the driver is added to the end of the device driver table. When a channel to a device driver is being established by the operating system, it searches the device table from the end first, thus the latest installed device driver with the required name will be asked first for a channel. So as with LDDs by this mechanism installable device drivers can replace any of the resident drivers.

When called, the DS and ES segment registers are in an unknown state. The device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register. The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored. All operating system services may be called except those concerning file or device access.
PASSED
No values are passed to the install vector.
RETURN
If the installation was successful, return with the carry flag clear.
If the installation failed, return with the carry flag set and the error number in the AL register.
PANIC
The install vector must not panic; it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the install vector.

DevFuncRemovePDD

This vector will be called by the operating system when the device driver is requested to be unloaded. The remove vector is called in the context of the operating system and not the process that requests the unload. The `DevRemove` operating system service will cause this vector to be called. Applications should not call this service directly; instead, they should call the `DevDelete` service. Before the remove function is requested, the operating system will send a `DevFuncHold` request to all LDDs. The LDD is responsible for ensuring that no activity will occur during the remove. Note that any device driver that handles hardware interrupts must contain an LDD since only LDDs receive a hold request. If the device driver is currently busy serving a client, the remove request should return an error. All resident device drivers will return an error since there is no mechanism by which they can be re-installed.

When called, the DS and ES segment registers are in an unknown state; the device driver should take whatever steps necessary to obtain direct addressability to its data. For loadable device drivers this involves setting the DS and ES registers to the CS register. The operating system will not move memory whilst in this function, thus the normal rules governing DS and ES may be ignored. All operating system services may be called except those concerning file or device access.
PASSED
No values are passed to the remove vector.
RETURN
If the remove was successful, return with the carry flag clear.
If the remove failed, return with the carry flag set and the error number in the AL register.
PANIC
The remove vector must not panic; it will cause an operating system kernel fault if it does.
PRESERVE
The SS, SP and BP registers must be preserved by the remove vector.

DevFuncOpenPDD

This function is defined as a convenience function for the LDD-PDD interface. When an application opens a channel to an LDD, it normally uses the `IoOpen` system service. If the name specifies, or the LDD requires, a PDD then it needs to open a channel to a PDD. The `DevOpenPDD` system service will call this PDD vector to establish a channel. The LDD now has a choice of calling a PDD vector using the `DevVector` system service or calling the fourth vector in the vector table directly. The fourth vector is assumed to be a strategy vector to which any parameters as required by the LDD-PDD interface can be passed. The *FAR* address of the strategy vector is returned by the `DevGetPDDAddress`. When an LDD receives a `DevFuncResume` it should call `DevGetPDDAddress` again to ensure that if the PDD has moved the LDD still has its correct address. As a design, a PDD could provide many vectors, one for each required function. The LDD would then use the `DevVector` system service to access each of these functions. The `DevGetPDDAddress` will only return the *FAR* address of the fourth vector.

When called, the DS and ES segment registers point to the data segment of the application process making the open function request. The application should ensure that this is indeed the case. The device driver must obey the normal rules concerning segment register manipulation. All operating system services may be called.

PASSED

The BX register contains a pointer to the PDD unit name. The pointer passed to the `DevOpenPDD` service is used to find the PDD device to open. The BX register is loaded with a pointer to the trailing colon (if any) in the PDD unit name. For example if the name "`TTY.AS5:A`" was passed to the `DevOpenPDD` service, BX would contain a pointer to `:A` upon calling the open vector.

RETURN

If the open was successful, return with the carry flag clear.
If the open failed, return with the carry flag set and the error number in the AL register.

PANIC

The open vector can panic; it will cause the process requesting the device open to terminate. It is however more usual to return an error to the calling process.

PRESERVE

The SS, SP and BP registers must be preserved by the open vector.

DevFuncStrategyPDD

This function is defined as a convenience function for the LDD-PDD interface. Typically, all application function requests are routed through the strategy vector. To speed the calling interface, the `DevGetPDDAddress` operating system function will return a *FAR* address of this vector. The device driver writer defines all the functions and return values as required.

When called, the DS and ES segment registers point to the data segment of the application process making the function request. The application should ensure that the DS and ES segment registers do in fact point to its data segment. The device driver must obey the normal rules concerning segment register manipulation. All operating system services may be called.

PASSED

The parameters passed are defined by the device driver write.

RETURN

All returns are defined by the device driver writer.

PANIC

The strategy vector can panic; it will cause the process requesting the function to terminate. It is however more usual to return an error to the calling process.

PRESERVE

Which registers are preserved is defined by the device driver writer.

10. ASIC4/ASIC5 BASED DEVICE DRIVERS

Introduction

This chapter describes the rules and problems involved in writing a device driver that controls some form of ASIC4 or ASIC5 based hardware. Details concerning the internal organisation of these ASICs were presented earlier in chapters 6 and 7. The emphasis is on their use for Series 3a applications, though most issues that will be discussed also apply to the Series 3, MC and HC range of Psion products.

In addition to the eight mandatory functions already detailed in the Device Driver Overview chapter, an LDD will usually provide additional functionality in the form of further vector table entries dependent upon the requirements of its application. Wherever possible, these functions should make use of the various pre-defined system defines for services such as reading (`p_read`) and writing (`p_write`) etc. It should be noted that there is no particular requirement to implement a device driver as a separate PDD and LDD and in the case of the example logical device driver A4EXIF.LDD, the approach taken is to incorporate both aspects into the one LDD.

SIBO Hardware Expansion Channels

The number of SIBO expansion channels supported by the host machine will vary according to the particular Psion hardware present. So that a single device driver may be compiled for the different host machine possibilities, a number of build flags can be used to set various constants within the include files. The hardware options are outlined in the table below:

SIBO Machine Flags ===============	Number of SIBO serial channels supported ===	Machine Build Flag ===============
Consumer (S3a)	Refers to S3a with ASIC9 only. Supports 1 SIBO channel (A)	`BUILDSB`
Corporate (HC)	ASIC1/2 only. Supports 3 SIBO channels (A, B, C)	`BUILDCH`
Workabout	ASIC9 only. Supports 3 SIBO channels (A, B, C)	`BUILDSC`
s3	Refers to s3 with ASIC1 only. Supports 1 SIBO channel (A)	`BUILDHH`
Other SIBO machines	ASIC9 or ASIC1/2 they support 2 SIBO channels (A, B)	machine-dependent

It should be noted that Psion device drivers should be designed to be easily adapted from machine to machine. For a well-written driver, the only change that needs to be made in adapting it for use on another Psion machine is the alteration of the build flag at the start of the code. This flag indicates to the compiler which SIBO machine flags as well as other variables should be set for the host machine. The four most important SIBO machine variables are the channel interrupt mask, the channel interrupt number, the channel interrupt vector and the hardware SIBO channel. The **channel interrupt mask** is an eight bit value or'd with the contents of either `A1InterruptMask` or `A9BInterruptMask` (the **mask registers**) to initiate interrupts on the relevant channel depending on whether the interrupt controller resides in ASIC1 or ASIC9. If this mask is then used in a `HwGetChannel` call, any hardware interrupts on the selected expansion channel will be directed to the appropriately coded interrupt service routine. The channel interrupt mask is also required in the subsequent `HwFreeChannel` OS service call and when stopping hardware interrupts. The **channel interrupt number** is a sixteen-bit quantity required by the `GenSetRevector` and `GenResetRevector` OS system services to indicate to the OS which default interrupt service routine is to be replaced by the suitably coded device driver interrupt vector. The **channel interrupt vector** is a sixteen-bit pointer to the location of that interrupt vector in the device driver code. Finally, the **hardware SIBO channel** is used by the `HwSelectChannel` OS service to direct any SIBO serial control or data frames along the appropriate channel. The situation regarding the value of these variables for the expansion ports on all current SIBO platforms is illustrated in the table overleaf:

SIBO Flags	**Consumer (S3a)**	**Corporate or Workabout**	**Corporate (HC)**	**MC**	**Consumer (Series3)**
Expansion Channels	1	3	3	2	1
Controller ASIC	ASIC9	ASIC9	ASIC2	ASIC2	ASIC2
Mask Register	A9BInterruptMaskRW	A9BInterruptMaskRW	A1InterruptMask	A1InterruptMask	A1InterruptMask
Channel Interrupt Masks	mask A9MSlave (Port C)	mask A9MExpIntA (Expansion Port A) mask A9MExpIntB (Expansion Port B) mask A9MSlave (Expansion Port C)	mask ExpIntLeftA (Expansion Port A) mask ExpIntRightB (Expansion Port B) mask Asic2Int (Expansion Port C)	mask ExpIntLeftA (Expansion Port A) mask ExpIntRightB (Expansion Port B)	mask Asic2Int (Expansion Port C)
Channel Interrupt Numbers	HwIrq2Revector	HwIrq4Revector HwIrq5Revector HwIrq2Revector	HwIrq3Revector HwIrq2Revector HwIrq4Revector	HwIrq3Revector HwIrq2Revector	HwIrq4Revector
Channel Interrupt Vector Ptr	IntVec0 (Int. Routine code at this label)	IntVec0 IntVec1 IntVec2	IntVec0 IntVec1 IntVec2	IntVec0 IntVec1	IntVec0
Channel Hardware Select	SelectChannel5	SelectChannel3 SelectChannel4 SelectChannel5	ExpChannelLeftA ExpChannelRightB SelectChannel7	ExpChannelLeftA ExpChannelRightB	SelectChannel7

Talking to ASIC4

All communication to an ASIC4 is via a Psion Serial Link. As explained in the chapter on the SIBO serial protocol, two forms of data can be sent and received along this channel. These are control and data bytes. Control bytes give specific instructions to ASIC4 and data bytes can either be data sent to or from ASIC4 or data given to or taken from peripheral chips in ASIC4's address space. Sending and receiving control and data frames down a Psion Serial Channel from a Series 3/HC host is simply a matter of IN and OUT instructions to various fixed I/O addresses. Various assembler macros have been set up to ease this task and provide machine independence and they are detailed in the appendix. These include:

`SCONTOUT`	Output the control byte held in the AL register
`SDATAIN`	Input a byte of data and place it in AL
`SDATAOUT`	Output the data byte in the AL register
`XNOP`	Wait a short while
`SBUSY`	Wait while the Psion Serial Link is busy

Once a piece of peripheral hardware has been designed the appropriate software must be written to control it. All hardware devices on Series 3/3a, Workabout and HC machines are controlled by device drivers. These act as an interface between a piece of hardware and an application that uses it. The following section goes into further detail regarding the methods by which a device driver is able to communicate with and thereby control an ASIC4 and ASIC5 based peripheral.

ASIC4 Registers

ASIC4 has eight registers and their functions in mixed (i.e. peripheral) mode are outlined below:

Register 0: This register is the read/write **Data Register** that controls the data lines D0-D7 which are normally in tri-state mode. Data written to the Data Register is output on D0-D7 during a write cycle and data input to the D0-D7 may be read from the register during a read cycle. On reset, this register holds the Info Byte and its bits have the following meanings:

Bit No.	D7	D6	D5	D4	D3	D2	D1	D0
	D	D	D	N	N	S	S	S

D	**D**	**D**	**Device type**
0	0	0	RAM SSD
0	0	1	Intel Flash type 1
0	1	0	Intel Flash type 2
0	1	1	TBS
1	0	0	TBS
1	0	1	TBS
1	1	0	Read only SSD (ROM OTP etc.)
1	1	1	Hardware Write protected SSD

N	**N**	**Number of devices**
0	0	1
0	1	2
1	0	3
1	1	4

S	**S**	**S**	**Device Size**
0	0	0	Illegal (Indicates no SSD present)
0	0	1	32Kbyte
0	1	0	64Kbyte
0	1	1	128Kbyte
1	0	0	256Kbyte
1	0	1	512Kbyte
1	1	0	1Mbyte
1	1	1	2Mbyte

Register 1: This register is both a read and write register. In read mode, it is termed the **Input Register**. The eight bits of this register are then defined as follows:

7	6	5	4	3	2	1	0
M	De	Ne	Se	X2	In2	In1	In0

The Se, De and Ne bits hold, on reset, the Extended Info Byte which is used in ASIC4 Extended mode to define the type of peripheral device as explained later in the section on ASIC4 reset and configuration and outlined below:

M	**De**	**Ne**	**Se**	
1	0	0	0	No peripheral devices
1	0	0	1	Turbo RS232 serial (16550)
1	0	1	0	3Fax
1	0	1	1	T.B.S.
1	1	0	0	T.B.S.
1	1	0	1	T.B.S.
1	1	1	0	T.B.S.
1	1	1	1	Extended info contained in ROM.

The X2 bit sets the state of the X2D2 input which is used to indicate whether the device size is correct. In M=0 mixed mode, the X2 bit must be low. Inputs In0-In2 hold the current status of the three correspondingly named general input lines to ASIC4.

In write mode, Register 1 is termed the **Device Size Register** where bits 3-0 (S3-S0) map to the settings of the decoder inputs (and hence the peripheral device size) as follows overleaf:

Size Register				**Decoder Inputs**			**Device Size (bytes)**
S3	**S2**	**S1**	**S0**	**DC0**	**DC1**	**DC2**	
0	0	0	0	A15	A16	A17	32k
0	0	0	1	A15	A16	A17	32k
0	0	1	0	A16	A17	A18	64k
0	0	1	1	A17	A18	A19	128k
0	1	0	0	A18	A19	A20	256k
0	1	0	1	A19	A20	A21	512k
0	1	1	0	A20	A21	A22	1M
0	1	1	1	A21	A22	A23	2M
1	0	0	0	A22	A23	A24	4M
1	0	0	1	A23	A24	A25	8M
1	0	1	0	A24	A25	A26	16M
1	0	1	1	A25	A26	A27	32M
1	1	0	0	A26	A27	0	64M
1	1	0	1	A27	0	0	128M
1	1	1	0	0	0	0	256M
1	1	1	1	0	0	0	Not Used

This register defaults to 0x0f on reset (i.e. not used).

Register 2: A write to this register , the **Address Increment Register**, will increment the addresses A0-A3.

Register3: This register is the write-only **Address Register** which controls all 28 address lines (A0-A27) directly and the eight chip selects (CS0-CS7) indirectly. The address register is written to in multi-transfer mode LSByte (AT0) first. There can be up to four bytes written:

Byte	Address lines
AT0	A0 - A7
AT1	A8 - A15
AT2	A16 - A23
AT3	A24 - A27

When the first byte is written, all the higher address lines (A8-A27) are reset to 0. Bits 4-6 of AT3 may be used by the internal address decoder to control the CS outputs CS0-CS3 if appropriate. On reset, all bits of this register are cleared.

Register 4, 5 and 6: Not implemented.

Register 7: This register is the write only ASIC4 **Control Register** and holds the following bits:

Bit:	7	6	5	4	3	2	1	0
Label:	LBO	TSTA	LTM	VPS	EDA	CSS	WRS	OES

Setting the LBO and VPS bits causes the corresponding lines on ASIC4 to be set high enabling them to be used as general purpose outputs for peripheral development. OES and WRS control R/W accesses.

Additional ASIC4 Signals

In0, In1, In2 are general purpose digital inputs. LBO, VPS are general purpose digital outputs. OE, WR control Read/Write bus accesses. MCSD, X2D2 should be tied to ground via a 100k resister. POR is the reset line.

Talking to ASIC5

The Psion serial protocol was designed to allow many different peripheral ASICs to lie on the same serial channel. Before software can communicate with an ASIC5 it must first be selected. ASIC5 must be selected in different ways depending on whether it has been configured (in hardware) to operate in either Pack or Peripheral mode. In addition to selecting ASIC5 when it is first used, ASIC5 must be re-selected whenever a hold and resume is generated due to power down, pack doors opening or when the peripheral is inserted or removed. ASIC5 will always respond to a select with a byte indicating what type of peripheral or pack it is. If there is no ASIC5 connected to the Psion serial link there will obviously be no response (zero is returned) and software must then take the appropriate action. To select a peripheral mode ASIC5 the following code can be used:

```
HwNullFrame
mov    al,(SerialSelect or Asic5NormalId)
SBUSY
SCONTOUT
XNOP
SBUSY
SDATAIN
test   al,al
je     NoASIC5PeripheralOutThere
```

To select a pack mode ASIC5 the following code can be used.

```
HwNullFrame
mov    al,(SerialSelect or Asic5PackId)
SBUSY
SCONTOUT
XNOP
SBUSY
SDATAIN
test   al,al
je     NoASIC5PackOutThere
```

Note that an ASIC4 can pretend to be an ASIC5 in pack mode and will respond appropriately.

ASIC5 Registers

ASIC5 has sixteen internal registers. To read or write from an ASIC5 register takes two steps. First a control byte must be sent along the Psion serial link to select which register. This should be followed by a read or write of the appropriate data value. The sixteen registers are listed in the table below. Some can be read and written to and some are read or write only.

Register	Read/Write	Function
0	R/W	Port A read and write data
1	R/W	Port B read and write data
2	R/W	Port B control
3	W	Port D and C write data
4	?	Not used
5	?	Not used
6	R/W	Interrupt Mask read and write
7	R/W	Interrupt/Control resister
8	R/W	UART Status/UART Control register
9	R/W	UART Receive/UART transmit holding register
10	W	UART Baud rate LSB
11	W	UART Baud rate MSB
12	R/W	Synchronous Port1 read/Port1 and 2 reset
13	R	Barcode read data
14	R	Synchronous Port2 read
15	?	Not used

Reading from the Port A register causes ASIC5 to generate a memory access cycle. The value read from port A will be the value retrieved from any attached memory or memory mapped peripheral. Writing to Port A will cause ASIC5 to generate a write cycle and write the supplied value to attached memory. In both cases the address used for the access will depend on ASIC5's mode and the configuration of ports B, C, and D. If the UART is enabled (Bit 0 of the port B mode register) memory cannot be accessed because port A lines are reused.

Writing to port B will cause the value written to be latched onto the output lines PB0-PB7 (which form address lines A0-A7 for memory access cycles). Reading from this register will return the last value written.

Writing to the port B mode controls various aspects of ASIC5 behaviour. The table below indicates the meanings of each bit in the register. When set to operate in counter mode the value output on lines PB0-PB7 can be incremented by reading from the port B mode register, post-incremented by reading or writing to the port A register or cleared to zero by writing to the Port D and C register.

Bit	Function
0	0, Memory mode. 1, Peripheral mode -enables UART
1	Port B mode, see table below
2	Port B mode, see table below
3	0, Normal mode. 1, Test mode
4	Not Used
5	Not Used
6	Not Used
7	Not Used

Bit2	Bit1	Mode
0	0	Counter mode
0	1	Latch mode
1	0	Baud rate out on port B
1	1	Test bus output on port B

When ASIC5 is in pack mode the first write to the port D and C register will be latched onto lines PD0-PD7. In multiwrite mode the second and subsequent data writes will be latched onto port C with the following bits in the register forming the following functions.

Data bit	Pin	Function
0	PC0	A16
1	PC1	A17
2	PC3	A18
3	PC4	A19
4	PC5	A20
5	?	Not used
6	-	SEL0
7	-	SEL1

The bit SEL0 and SEL1 determine which chip select is used when a memory access cycle is generated by accessing port A. CS0-CS3 are selected as follows.

SEL1	SEL0	CS
0	0	CS0
0	1	CS1
1	0	CS2
1	1	CS3

In peripheral mode a write to the port D and C register will latch the value written onto lines PD1-PD7. In peripheral mode the line PD0 forms the UART TX line. Setting bit zero in the port C and D register therefore has no effect.

ASIC5 is capable of generating an interrupt due to various external events. The interrupt mask register is used to select which events will generate an interrupt. Writing to the interrupt mask register will set the mask to the value written. Reading the interrupt mask register will return the current mask. The meanings of each bit in this register is given in the table below. Setting an appropriate bit to 1 will enable that events interrupt, clearing to 0 will stop that event from causing an interrupt. On reset all events are disabled.

Bit	Source	Event
0	UART	UART character received
1	UART	UART transmitter empty
2	UART, change on PA1-PA3	UART error or modem line status change
3	PC7	Barcode switch/general interrupt
4	SR	Synchronous port1 character received
5	SR	Synchronous port2 character received
6	PC4	Barcode data/general interrupt
7	PA4	Centronics busy low/general interrupt

When an interrupt occurs reading the Interrupt Status register will indicate the source of the interrupt. The meaning of each bit corresponds directly to each bit in the interrupt mask (see table above). A high bit indicates an active interrupt event.

Reading the Barcode data register returns a byte representing the states of the following lines.

Bit	Line
0	PC4
1	PC7
2	PB2
3	PB3
4	PB4
5	PB5
6	PB6
7	PB7

Communicating with ASIC4

ASIC4 is capable of operating in several modes. Attached peripherals can contain an ASIC5 chip instead of an ASIC4. Before talking to an ASIC4 you must first ensure that there is an ASIC4 at the end of your serial link and you must set ASIC4 into the correct operating mode. You must do this when the channel is first obtained to ensure you have the right peripheral to start with, and also whenever you get a Hold and Resume due to a power down, Pack doors opening, or the peripheral being removed. You should also check the peripherals ID to check what peripheral it is. The user is more than likely to pull out your I/O port and replace it with a 3Fax in mid operation. You select ASIC4 in the correct mode by sending a special control code. If there is an ASIC4 out there it will respond by sending back a non zero value. The peripherals ID can be read back from ASIC4's info register. This register also returns the state of the input lines IN0, IN1, IN2. The coding example below shows how ASIC4 can be checked for, selected and the peripheral ID checked:

```
HwNullFrame
mov    al,(SerialSelect or Asic4Id)        ; First look for an
SBUSY                                      ; ASIC4 at the other
SCONTOUT                                           ; end of the link
XNOP
```

```
       SBUSY
       SDATAIN
       test   al,al
       je     NoAnASIC4IsItAnASIC5
       mov    al,(SerialReadSingle or A4InfoR)          ; Now see if we
       SBUSY                                     ; have the right
       SCONTOUT                                         ; peripheral
       XNOP
       SBUSY
       SDATAIN                                   ; Mask out the unwanted
       and    al,0f0h                            ; bits
       cmp    al,PERIPHERAL_ID                          ; check for correct ID
       jne    IsASIC4ButNotRightPeripheral       ; Got our peripheral
       clc                                       ; Connection Okay!
       ret
IsASIC4ButNotRightPeripheral:                           ; Is an ASIC4 but not
       popf                                      ; the right peripheral
       stc                                       ; Connection Failed!
       ret
NoAnASIC4IsItAnASIC5:                            ; No ASIC4
       mov    al,(SerialSelect or Asic5NormalId)
       SBUSY                                     ; Could have been an
       SCONTOUT                                         ; ASIC5 so try to put
       XNOP          stc                                       ; back in the
right mode
       ret                                       ; Connection Failed
```

Bits in ASIC4's info register have the following definitions:

1	ID msb	ID	ID lsb		IN2	IN1	IN0

Sending and Receiving data using ASIC4

All ASIC4 peripherals are memory mapped into that ASIC's address space. To access a peripheral or a particular peripheral's register the ASIC must first be told to set its address bus to the appropriate location. Once it has been informed of the address which it is to access that location can be read from or written to as many times as required. The value to be written is held in the data register. To read or write to another location the address that the ASIC is accessing must be changed. To read from a random address within the ASIC's address space the sequence would be:
1) Send a control code to inform the ASIC that it is to set its address bus to the following address.
2) Send the address to position to.
3) Send a control frame to inform the ASIC that we wish to read from this address.
4) Read back the byte.

To write a random value to and address within the ASIC's address space the sequence would be:

1) Send a control code to inform the ASIC that it is to set its address bus to the following address.
2) Send the address to position to.
3) Send a control frame to inform the ASIC that we wish to write to this address.
4) Send the byte.

Overleaf are two 8086 assembler functions Input and Output which read from or write to an address within the ASIC's address space. These functions only give access to the bottom 256 addresses. For peripheral devices this is usually more than adequate. Interrupts should always be off during calls to these functions to prevent the S3/S3a/HC from multitasking. Note that in the following routines, `A4Address` and `A4Data` refer to the value of the respective ASIC4 registers. `SerialWriteSingle` has the value corresponding to a single frame write in the SIBO serial protocol format. These system defines can be found in the important include files ospack.inc and ossibo.inc.

```
       ProcBegin@ Output
;      ================
;
;      Output byte to ASIC4-based peripheral
;      IN: DL holds the hardware address ASIC4 is to write to
;          AL holds the byte to be output to that address

       push   ax
       mov    al,(SerialWriteSingle or A4Address)        ;CTRL=Write to A4 address
reg
       SBUSY
       SCONTOUT                                                 ;Send this control
frame
       mov    al,dl                                      ;DATA=Hardware address
       SBUSY
       SDATAOUT                                                 ;Send this data
frame
       mov    al,(SerialWriteSingle or A4Data)                  ;CTRL=Writing data
now
       SBUSY
       SCONTOUT                                                 ;Send this control
frame
       pop    ax                                         ;DATA=Data to write
       SBUSY
       SDATAOUT                                                 ;Send this data
frame
       ret
       ProcEnd noret

       ProcBegin@ Input
;      ================
;
;      Input byte from ASIC4-based peripheral
;      IN: DL holds the hardware address ASIC4 is to read from
;      OUT:AL holds the value read from that address

       mov    al,(SerialWriteSingle or A4Address)        ;CTRL=Write to A4 address
reg
       SBUSY
       SCONTOUT                                                 ;Send this control
frame
       mov    al,dl                                      ;DATA=Hardware address
       SBUSY
       SDATAOUT                                                 ;Send this data
frame
       mov    al,(SerialReadSingle or A4Data)                   ;CTRL=Reading data
now
       SBUSY
       SCONTOUT                                                 ;Send above
control frame
       SBUSY
       XNOP
       SDATAIN                                           ;Receive data frame
       ret                                               ;AL holds the data
       ProcEnd noret
```

Obtaining and using a channel

The s3/S3a/HC have three Psion serial links. Only one of these channels can be selected for communication at any one time. Two of these channels form the SSD slots. The third forms the expansion port. Before a device driver is able to talk to a peripheral it must both own and have this third channel selected. Many device drivers can exist in memory and more than one may access the peripheral. To prevent multiple access to a single peripheral a device driver must first own a serial channel before it is free to communicate to the ASIC4/5 peripheral at the end of it. To check if a channel is free and then reserve it call the operating system function `HwGetChannel`. This will return with carry clear if the attempt to capture the channel was successful, carry set otherwise.

```
       mov    al,InterruptMaskForDesiredChannel
       HwGetChannel
       jc     CouldntGetTheChannel
```

Obtaining the desired channel will usually be performed by a device drivers open vector. Open should fail if the channel is unavailable. If a device drivers takes possession of a channel it is its duty to free it again once it is finished. This would normally be done when the device drivers is closed and is performed by the operating system function `HwFreeChannel`.

```
        mov     al,InterruptMaskForDesiredChannel
        HwFreeChannel
```

As illustrated in the above table, on a series 3a the interrupt mask for the expansion port (Port C) is defined by `A9MSlave`, on a series 3 `Asic2Int`. Interrupts are IRQ2 and IRQ4 respectively. Only one serial link can ever be selected at one moment in time. Selecting a particular channel means that you are unable to speak to another without selecting it instead. To select a channel the operating system call `HwSelectChannel` is used.

```
        mov     al,SelectForDesiredChannel
        HwSelectChannel
        push    ax
```

`HwSelectChannel` will return in al the channel that was previously selected. Device drivers should select the correct channel on entry to any vector or interrupt service routine that needs to communicate down that channel. On exit the previously selected channel should be restored. Between the time when the channel is first selected and when the old channel is restored multitasking should be disabled. this will usually mean switching off interrupts. Because of the watchdog timer, interrupts cannot be left off for an indeterminate length of time. Communications down channels therefore will usually be restricted to short bursts. For the expansion port the channel to select on an S3a is defined by `SelectChannel5` on an S3 `SelectChannel7`.

```
        pushf
        cli
        mov     al,SelectChannel5
        hwSelectChannel
        push    ax

        mov     dl,AddressOfInputBuffer
        call    Input
        mov     dl,AddressOfOutputBuffer
        call    Output

        pop     ax
        hwSelectChannel
        popf
```

Controlling ASIC5's UART

Before data can be sent or received from ASIC5's UART continuous clocking from the host must be enabled and a baud rate selected. The UART must be enabled by setting bit 0 in the port B mode register. The selected baud rate value is related to the result of dividing the required baud rate into the input clock frequency from the host of 1.536 MHz. The value generated from the equation given below forms a sixteen bit word. ASIC5 has two registers for selecting the baud rate. Register 10 should contain the least significant byte and register 11 the most significant byte of the calculated word value. The table below lists the divisor values for some of the more commonly required baud rates:

Divisor = 1-(96000/Desired Baud Rate)

Baud Rate	Divisor -Decimal	Divisor -Hexadecimal
48000	-1	ffff
32000	-2	fffe
19200	-4	fffc
9600	-9	fff7
7200	-12	fff4
4800	-19	ffed

3600	-26	ffe6
2400	-39	ffd9
2000	-48	ffd0
1800	-52	ffcc
1200	-79	ffb1
600	-159	ff61
300	-319	fec1
200	-480	fe20
150	-639	fd81
134	-715	fd35
110	-872	fc98
75	-1279	fb01
50	-1919	f881

The format of data communicated, stop bits, data bits, parity checking, is controlled by writing to the UART status/control Register. Bits in the UART status/control register have the following meanings:

Bit	Function: Read	Function: Write
0	State of the CTS line (PA1)	Generate break character
1	State of the DSR line (PA2)	Character length 1 (see table below)
2	State of the DCD line (PA3)	Character length 2 (see table below)
3	Transmitter buffer empty	Parity enabled if set
4	Transmitter Busy	Odd parity if set, even if clear
5	Receive data waiting	Set for two stop bits, clear for one
6	Overrun or framing error	Not used
7	Parity error	Not used

A character to be transmitted should be written to the Transmitter Holding Register where ASIC5 will convert it to serial form for transmission. The register will be emptied once the character has been transmitted. ASIC5 contains no internal buffering. The Transmitter Holding Register must be empty before writing a character to it. The state of the Transmitter Holding Register is reflected in the Transmitter Empty bit in the UART status register. Enabling the Transmitting Holding Register interrupt will cause ASIC5 to generate an interrupt every time that the Transmitting Holding Register becomes empty. Reading the UART Status Register will clear the interrupt. Received characters are copied into the Receive Character Register. If the Receive Character Interrupt is enabled ASIC5 generates an interrupt on each character received. If the character is received in error due either parity, framing or overrun errors, appropriate bits in the UART Status Register are set to reflect this.

Hold and Resumes

There are three types of hold and resume. The first occurs when memory is being moved. An LDDs hold and resume vectors will always be called when this occurs. The second is a On/Off Hold and resume. An LDD will always be held and resumed when this occurs although it is not guaranteed that power will have been restored to the expansion ports when the resume is issued. The third type occurs when one of the pack doors is opened or a peripheral inserted. Only certain internal LDDs receive a Hold and Resume under these circumstances. Any loaded LDD will definitely not receive a Hold or Resume. To get around these problem, the `TCK:` device driver is used to provide a regular call to a routine within your driver. This call monitors the power to the expansion port and issues its own holds and resumes as appropriate. Holds and resumes can become nested and the whole situation can become rather complicated. The correct procedure for dealing with holds and resumes for an ASIC4 peripheral device driver (LDD) is given below.

When a hold is received through the normal route the channel is marked as being under a normal hold. If the polling routine sees that the power has vanished and the channel is not already under a normal hold then it holds the channel and marks it as being under a special hold. If a normal resume is issued channel can be resumed but only if power is present. If power is not present then the channel is marked as being under a special hold and the resume is put off. If the polling routine sees that the power is present and that the channel is under a special hold then it should resume the channel.

Alternatively of course each time you read and write to your peripheral you could do a quick pre-check to see if the hardware is available and set-up correctly. This would dispense with the need to handle power up/down or pack door holds and resumes. You would still have to start and stop interrupts though when appropriate.

Example Device Drivers

An example device driver which will control the ASIC4 Example Interface Board described earlier can be found in the file A4EXIF.ASM. The device driver is a fully comprehensive multi-channel implementation. It supports all the mandatory logical device driver functions. Open will fail if the Interface Board hardware is not present. It has an interrupt service routine and handles Holds and Resumes correctly in all the distinct cases outlined earlier. In addition, asynchronous reads of the LED status are possible so a special routine to handle this case, the Wait Handler, is included. The driver has been made as fully comprehensive as possible both in terms of functionality and the choice of host machine. To this end, the code given can be compiled for the entire range of Psion machines simply by changing the appropriate machine Build Flag at the head of the file which then causes the correct interrupt masks to be selected for the machine. Chapter 11 of this document details the structure of A4EXIF.LDD a logical device driver constructed for the ASIC4 Example Interface Board.

Chapter 12 details the functionality of a physical device driver for the ASIC5-based Psion 3-link peripheral that enables the host hardware to communicate with a PC by converting SIBO serial protocol signals into RS232 signals. Once again, this driver contains various build flags to facilitate conditional compilation. Common interrupt routine code is included and the LDD-PDD interface is specified.

The source code for both A4EXIF.LDD and SYS$AS5.PDD is presented in the appendix to this document.

11. An Example Device Driver for ASIC4: A4EXIF.LDD

Introduction

In this chapter, the functionality and code structure an example installable logical device driver, A4EXIF.LDD, is presented in some depth. Circuit details of the corresponding Psion peripheral for this driver, the ASIC4 Example Interface board, were outlined in chapter 8. A4EXIF.LDD enables software written using the Psion SDK to communicate with the prototype ASIC4 Example Interface board through means of standard PLIB calls such as '`p_open`' and '`p_write`'. The description of A4EXIF.LDD is intended to provide a clear insight into the generalised structure of logical device drivers for peripherals based around the SIBO architecture. To this end the actual usefulness of the combined hardware-driver interface is of secondary importance. As shown earlier, the ASIC4 Interface Board translates SIBO serial protocol signals into a parallel 8-way data bus format which can be used to set various 74HC series latches and gates. An eight-bit buffer and latch are commoned to eight LEDs. The latch is write-only and is used for setting the state of the LEDs. The buffer is read-only and is used to sense the state of the LEDs. In addition, a facility for generating hardware interrupts is provided by means of a suitably connected switch and third 8-way buffer (the status buffer). Address decoding is provided by two 2-to-4 decoders attached to address lines A0 and A1.

Code Structure

Loading A4EXIF.LDD and Device Names

A4EXIF.LDD is loaded into the RAM of the host machine by means of the following PLIB call in the application code:

```
p_loadldd("A4EXIF.LDD");
```

The name of a device driver is the mechanism by which an application can obtain a channel to that device driver. LDDs all have three character names which are stored in the second field of the driver's LibEnt structure followed by a colon. This name is required to uniquely identify the LDD to the OS when attempting to open a channel on it. The A4EXIF LDD has the three character name "`LED`" so a channel with its handle in `pcb` may be obtained on it by means of the following call:

```
p_open(&pcb,"LED:",-1);
```

EPOC used the driver name in the `p_open` call to invoke the `IoOpen` system service which in turn invokes the Open vector on the associated device driver.

The Single Code Segment and Data Storage

As indicated earlier, loadable Psion device drivers have a single code segment and no data segment. Data associated with the driver is stored in one of two distinct ways dependent on its nature. Global data associated with the driver is stored in its code segment. Examples of such data are the channel interrupt masks and numbers have to be visible to all processes that may be using A4EXIF.LDD. Data local to the application invoking the driver, however, is stored in the heap space of that application. An example in A4EXIF.LDD would be the pointers to the open channel strategy vector parameters.
The overall structure of the code segment is typical of Psion logical device drivers. The code segment begins with a `CodeSeg` directive followed by the LibEnt structure which defines the mandatory device driver functions as well as two non-mandatory ones. These are the Wait Handler and the replacement

Tick Interrupt Vector. Before entering the code for these functions, all global variables are declared in what is termed the device driver's internal data space. Following the code for the Lib Ent functions comes the code for all the local driver functions. After these are the `EndCodeSeg` and `end A4ExifLDD` directives.

Channel Status Ent and Open Channel Control Block

At the head of the driver's assembler source file A4EXIF.ASM, various constants, compiler defines and type definitions are listed. It is here that templates are declared for the global structures that hold the key driver variables. In view of the above discussion, most device drivers employ at least one globally defined structure to hold the various important flags and masks that relate to the status of each separate channel on that driver. In the case of A4EXIF.LDD, two different structure types are employed, namely A4ExifStatusEnt and A4ExifEnt. The former is referred to in this chapter as the **StatusEnt** struc and is instantiated later on in the device driver code segment with each separate permissible channel of the main Psion host machines being assigned its own StatusEnt struc. A channel's StatusEnt struc is generally accessed through CS:DI and holds important channel-related information such as the `A4ExifChannelOpen` and `A4ExifChannelRunning` flags and the channel interrupt masks and interrupt numbers. The StatusEnt structure resides in the driver code segment whereas the A4ExifEnt struc is held in the heap space of the application invoking the driver and so is accessed through DS:BX. The A4ExifEnt structure contains a number of variables that are logically associated with a successfully opened channel and as such should be distinguished from the StatusEnt variables. An A4ExifEnt struc instantiated by invoking the device driver's open vector is referred to as the **open channel control block** and it is generally accessed through DS:BX.

The layout of both the StatusEnt struc and the A4ExifEnt open channel control block struc are presented later in this chapter along with various other important pre-defined structs that were used in the construction of A4EXIF.LDD. Both of these structures were extensively used in the coding of the LDD functions described in the following pages.

SIBO hardware and conditional compilation

The number of expansion channels supported by a host SIBO machine is dependent on the hardware. In the case of the S3a, only one SIBO channel can legitimately be opened corresponding to expansion port A. With the HC and **Workabout**, it is possible to open up to three separate SIBO channels on ports A through to C. In A4EXIF.LDD, a global constant `numberofchannels` is set at the start of the device driver code segment to indicate the number of serial SIBO channels supported by the different machines. This will vary according to the Psion hardware present and in order to aid conditional compilation of the A4EXIF driver for the different host machine possibilities, a number of build flags can be used to set various constants within the include files. The hardware options with regard to these flags were outlined in the previous chapter.

It should be noted that Psion device drivers should be designed to be easily adapted from machine to machine. With the A4EXIF driver, for instance, the only change that needs to be made in adapting it for use on another Psion machine is the alteration of the build flag at the start of the code segment. This flag indicates to the compiler which SIBO machine flags as well as other variables should be set. The SIBO machine flags are in turn used later in the code segment header to set the global `numberofchannels` variable mentioned above. For example, in A4EXIF.LDD, we have the following:

```
if Corporate or Workabout
       numberofchannels            equ    3
else
       .......etc.
endif
```

This type of code is used extensively in the conditional instantiation of A4ExifStatusEnts in the device driver's internal data space. Here different interrupt masks and SIBO channels are invoked for the different machines according to the host controller ASIC. For instance, in the case of a consumer (S3a), the hardware interrupt mask corresponding to expansion port C on an ASIC9 is mask A9MSlave and the interrupt number is HwIrq2Revector. On an Workabout, the masks for expansion ports A, B and C are A9MExpIntA, A9MExpIntB and A9MSlave respectively. The corresponding interrupt numbers are HwIrq4Revector, HwIrq5Revector and HwIrq2Revector. This conditional compilation code in A4EXIF.LDD outlined here is worthy of some study because it encapsulates all the information regarding masks and SIBO hardware channels required by the prospective developer interested in constructing a multi-platform Psion peripheral and driver.

Mandatory LDD Functions

The eight mandatory LDD functions are the Install, Remove, Hold, Resume, Reset, Units, Open and Strategy and they are all discussed in the context of the example device driver A4EXIF.LDD below.

A4ExifInstall

IN:	No values are passed to the install vector.
OUT:	If successful return with carry clear.
	If installation unsuccessful, return with the carry flag set and error number in the
AL	register.
PRESERVE:	SS, SP and BP

The install vector is called by the operating system when the device driver is loaded in order to initialise any internal variables. It should not be called directly by an application process but invoked indirectly by a '`p_loadldd("A4EXIF.LDD")`' call. Typically at this stage the various open channel flags would be initialised to zero but in the case of the A4EXIF.LDD, this has already been done in the appropriate A4ExifStatusEnt struc headers. As a result, the install vector merely clears the carry flag and returns.

A4ExifRemove

IN:	No values are passed to the remove vector
OUT:	If successful return with carry clear.
	If installation unsuccessful, return with carry set and error number in the AL
register.	
PRESERVE:	SS, SP and BP

The remove vector is called by the operating system whenever the device driver is requested to be unloaded which is usually indirectly as the result of a call to '`p_devdel("LED",E_LDD)`'. ROM-resident drivers cannot be deleted so any attempt to invoke this vector on them will result in an error being returned. The code for the remove vector checks to see that all the device driver channels are closed and returns carry clear if this is the case. In the case of A4EXIF.LDD, a closed channel will hold a zero in its corresponding StatusEnt open channel flag. If any of the open channel flags is non-zero then the carry flag is set before returning indicating that an attempt has been made to unload a device driver that still holds at least one open channel. In the case of A4EXIF.LDD, the `numberofchannels` flag is used as a loop counter for the different hardware channel possibilities in a similar manner to the corresponding code in the open vector.

A4ExifHold

IN:	AH register holds one of the following values:
	`DevHoldNormal`, `DevHoldPowerDown` or `DevHoldPowerFail`
OUT:	None
PRESERVE:	SS, SP and BP

The hold vector is called by the operating system whenever the device driver is requested to be held. This can occur as a result of three conditions which result in the three different possibilities for the value held in AH:

1. Device memory segments about to be moved. AH holds `DevHoldNormal`.

2. Machine about to switch off due to power-save time-out or the off switch being pressed. AH holds `DevHoldPowerDown`.
3. Machine about to switch off due to power source being removed or the batteries failing. AH holds `DevHoldPowerFail`.
The hold vector code must be able to conduct a rapid shut down of the device driver because in the case of `DevHoldPowerFail`, the driver may only have a couple of ms before internal voltages fall to an unworkable level. The EPOC OS automatically handles the three cases highlighted above but in Series 3a machines and the Workabout, in addition to these there is another situation that requires the invocation of the hold vector, namely the opening/closing of the SSD pack doors on an S3a or Workabout. This is an action which can only be detected by means of software polling which requires the setting up of the ROM-resident "`TCK:`" device driver so that the door status can be checked each system tick (i.e. 32 times a second). The section on A4ExifTickInt provides a more detailed outline of the TCK interrupt code required to handle pack door opening and closing. It is this latter routine that polls for the open or closed status of the host machine doors, setting the global `HoldFlag` accordingly. The Hold vector code thus includes a check on the status of the `HoldFlag` and is generally optimised to be as efficient as possible. An outline of the structure of A4ExifHold presented below. It follows the pattern for a typical device driver:

- Check the state of the global `HoldFlag`.
- If the flag is non-zero, then the driver is already held and the function returns without taking any further action.
- If the `HoldFlag` is zero, then the function checks the status of all the channels that the driver is permitted to open. If any of these channels have their open channel flags cleared, it is not necessary to do anything as part of the process of holding the channel. Only in the case when the flags are set is the '`StopChannelRunning`' function invoked. As with the similar loop in the Reset vector, the global `numberofchannels` is used as the loop counter as each of the possible channels has to be checked.

A4ExifResume

IN:	None
OUT:	None
PRESERVE:	SS, SP and BP

The resume vector is called by the operating system to restart a held device driver. Resume will be called after a hold caused by any of the four possibilities indicated above in the section on A4ExifHold but it should be noted that in some cases, ASIC4 will only be switched on *after* the call to resume. The device driver is expected to recover from the previous hold and resume any suspended I/O except in the case of power failure where the resume vector ought to power up the peripheral and put it into a known state indicating to the user that data may have been lost. The resume vector will also have to handle the reinstallation of any interrupt service routines by checking the status of the open channel flag.

- Check the state of the global `HoldFlag`.
- If the flag is zero, then the driver has already resumed so return without further action.
- If the flag is non-zero, then check the door status of the host machine with a call to the OS `HwGetSsdData` service. If the doors are open, then set `HoldFlag` to 2 to signal this position to the driver. If the doors are closed, then `numberofchannels` is used as a loop counter while the open channel flags of each of the permissible channels for the host hardware are checked. Only if these flags are set is the '`StartTheChannelRunning`' function invoked.

A4ExifReset

IN:	BX holds the device driver device handle
	CX holds user specified channel identification data. In this case, holds the address
of	the status struc identifying the open channel to be reset.
OUT:	None

PRESERVE: SS, SP and BP

The reset vector enables a device driver to reset itself when the application which owns the device driver terminates abnormally before closing one or more open channels on that driver. The vector is only invoked by the operating system if it has been primed by means of a call to the `IoRequestReset` operating system service in the open vector. Such a call must be balanced by a corresponding call to `IoRequestResetCancel` in the close function code of the strategy vector. Registers BX and CX are required to be set up by the user prior to invocation of either of these services. BX should contain the device driver handle which is passed through DX in the open vector. CX should hold any suitable open channel identification data that can be accessed in the driver's internal data space (i.e. through CS:DI). In the case of A4EXIF.LDD, this information is the address of the channel StatusEnt structure. If a device driver is only capable of opening one channel, then it is irrelevant what CX holds. A reset should only be outstanding while some process has a channel open on the device driver. The format of A4ExifReset is similar to that of a typical reset vector and can be summarised as follows:

- Move the address of the channel status Ent from CX into DI.
- Check the value of the corresponding cs:[di].A4ExifChannelOpen flag.
- If channel is not open to reset, then return from the vector otherwise call '`StopTheChannelRunning`' and use `HwFreeChannel` to release the SIBO serial channel. Set the ChannelOpen flag to zero.

A4ExifUnits

IN: None
OUT: **AX holds the number of channels supported.**
PRESERVE: SS, SP and BP

The A4ExifUnits function is called by the operating system when the device driver is requested to report the number of units (or channels) that it can support. This is primarily useful for purposes of information.

A4ExifOpen

IN: **DX contains the device handle of the device driver**
SI is a pointer to an OpenEnt struc
BP is a pointer to an IntEnt struc
DS, ES point to the application process's data segment
OUT: **If successful return with carry clear and the address of the open channel in BX.**
i.e. BX holds the address of an A4ExifEnt struc which contains various fields of information concerning the open channel.
If installation unsuccessful, return with carry set and error number in the AL register.
PRESERVE: DS, ES, SS, SP, BP and DX

The open vector is called by the operating system when a channel to the device driver is required to be opened. The device handle, which is passed through DX, is used by the operating system to route any I/O requests on the opened channel to the correct device driver. An LDD must place this handle in the ChanLibHandle field of the ChanEnt struc of the open channel structure allocated within the application's data segment. In the case of A4EXIF.LDD, this structure is the A4ExifEnt struc described in the Appendix to this document. The open vector of a device driver runs in the context of the process that has called `p_open` to open a serial port on that device driver.

The OpenEnt struc consists of three fields , OpenNamePtr, OpenMode and OpenChan. The OpenNamePtr points to the character immediately following the device name as passed in the `p_open` call. For instance, in the case of A4EXIF.LDD, the OpenNamePtr would point to the colon after LED in the name "`LED:`". The OpenMode field contains the mode for opening the device driver which is not used in this example. The OpenChan field holds the I/O channel handle of the device that the driver is required to 'attach' to. Attached drivers add functionality to, or replace, a service provided by an underlying device driver. The IntEnt struc pointed to by BP can be used to reload the various segment registers if their contents have been altered. The A4ExifEnt struc requires further elaboration. It consists of four fields: A4ExifIo, A4ExifHandlerPtr, A4ExifStatusPtr and A4ExifA1Ptr. A4ExifHandlerPtr contains a handle to the driver's WaitHandler, A4ExifHandler. This is the routine required to handle the completion of asynchronous I/O requests and is described in more detail later on in this chapter. Of the remaining fields, A4ExifStatusPtr holds the location of the completion status word for I/O requests and A4ExifA1Ptr is a user-defined parameter. These two values are passed to the strategy vector through the RqEnt structure. A4ExifIo is itself a ChanEnt struc with sub fields ChanSignature, ChanNext and ChanLibHandle. After initialisation of this structure, [bx].A4ExifIo.ChanSignature must hold '`IoChanSignature`' and [bx].A4ExifIo.ChanLibHandle holds the handle to the device driver (stored in DX). Both of these values are again required by the strategy vector. The [bx].A4ExifIo.ChanNext field is used by attached drivers and is set to 0 for root drivers like A4EXIF.LDD.

The code in an open vector follows a similar pattern in many LDDs and is presented in outline form below:

- Determine the channel to be opened using the pointer to the device name held in [si].OpenNamePtr.
- If OK, disable interrupts and invoke the OS `HwGetChannel` call.
- If OK, then set A4ExifChannelOpen flag to non-zero.
- In order to poll for door-opening and closing, it is necessary to open a "`TCK:`" channel at this stage and initialise it by means of using the OS `IoWithWait` service to call the TickInt vector on each system tick.
- Re-enable interrupts and set up the calling process id using the `ProcId` service.
- Check for the presence of the required hardware. If present return with carry clear.
- Allocate space in calling process' heap (i.e. the application data space) to contain the I/O channel control block (an A4ExifEnt struc in the case of A4EXIF.LDD) and set its base address to BX (thus BX holds the address of the open channel control block).
- If OK, install the device driver Wait Handler using the `IoAddHandler` system service. A Wait Handler is required only if there are any asynchronous I/O requests to be handled.
- If OK, initialise the fields inside the I/O channel control block that were outlined above.
- Invoke `IoRequestReset` system service to handle unexpected termination of main program by invoking the driver's Reset vector.
- Before leaving the Open vector, if the call succeeded, BX must hold the open channel control block handle.

A4ExifStrategy

IN: **BX holds the allocated channel control block initialised in the open vector**
DX contains device handle of the device driver in the case of an LDD
SI is pointer to RqEnt struc
BP is pointer to IntEnt struc
DS, ES point to data segment of application making the I/O function request

OUT: **If successful return with carry clear**. Furthermore, in this case, if the strategy vector is meant to complete the I/O request (as is the case with IoClose for instance) then the completion status should be written back to RqStatusPtr location and the I/O semaphore signalled by calling `IoSignal`. If the I/O request is not completed by the strategy vector (as is the case with the asynchronous functions `IoFuncRead` and `IoFuncWrite`) then `PendingErr` should be written back to the location pointed to by RqStatusPtr and the I/O semaphore should *not* be signalled.
If installation unsuccessful, return with carry set and error number in the AL register.

PRESERVE: DS, ES, SS, SP and BP

When an application makes a I/O request on the opened device driver channel, the request is routed to this vector by the operating system. A device driver defines the set of strategy functions that it supports such as `IoFuncSet`, `IoFuncSense`, `IoFuncWrite` and `IoFuncClose`. The functions of a device driver are usually dependent on its purpose and there is no requirement to support any particular function. The ordering of these functions in the strategy vector table is defined in p_file.h and presented in A4EXIF.ASM.

The SI register contains a pointer to an RqEnt struc which consists of four fields: RqFunction, RqStatusPtr, RqA1Ptr and RqA2Ptr. RqFunction contains the function number passed to the I/O request by the application. In the device driver strategy vector code, the value held in RqFunction is compared against the supported function numbers held in `IoFuncClose`, `IoFuncRead` etc. If the function number is not one of those supported then a call to `IoRoot` is necessary in the case of a root device driver such as A4EXIF.LDD. `IoRoot` chains the I/O request through to the operating system which runs some default code to handle it. RqStatusPtr is a pointer to a memory location in the application data space which holds the value of the I/O request's completion status word. While a request is outstanding, this value is set to `PendingErr` and only when it completes does a completion code get written to this location. I/O requests can complete within the strategy vector or later after an interrupt. Often as is the case with A4EXIF.LDD, only a single request of any one kind can be outstanding at any time otherwise the application is panicked. RqA1Ptr and RqA2Ptr hold the values of two parameters that are passed to the `IoAsynchronous` EPOC service. Each synchronous strategy function has a similar pattern with explicit I/O being conducted whilst interrupts are disabled and a completed request being signalled with an OS `IoSignal` call prior to exit from the function. This call signals to the OS the completion of the particular I/O request entailed by the function call. The situation with regard to the one asynchronous strategy function is somewhat more complicated and is dealt with in greater detail in a later section.

The structure of the strategy vector of A4EXIF.ASM is outlined below:

- Determine the strategy function number using [si].RqFunction and compare with the below:
- **IOFUNCREAD**: Asynchronous read of latches U4 and U3. Panic if [bx].A4ExifStatusPtr != 0. Update [bx].A4ExifA1Ptr and [bx].A4ExifStatusPtr with [si].RqA1Ptr and [si].RqStatusPtr respectively as the locations to be accessed on receipt of interrupt. Move `PendingErr` into the location pointed to by [bx].A4ExifStatusPtr to signal that the request is awaiting completion. Allow handler to be enabled through the `IoEnableHandler` OS call and exit without signalling completion of the request.
- **IOFUNCWRITE**: Identical behaviour to that of `IOFUNCSET` which it calls.
- **IOFUNCCANCEL**: Used to cancel any pending asynchronous reads. This involves disabling the wait handler, consuming any stray signal from the interrupt routine by means of an OS `IoWaitForSignal` call and then storing `CancelErr` in the channel I/O request status word.
- **IOFUNCCLOSE**: Close down the TCK channel and remove handler. Disable interrupts and `HwFreeChannel` setting A4ExifPid to 0 before re-enabling interrupts. Invoke `IoRequestResetCancel` system service and `HeapFreeCell`. Signal completion of I/O request by calling `IoSignal` system service.
- **IOFUNCSENSE**: Reads the status and LED bytes from latches U4 and U3 respectively storing the results at the addresses pointed to by RqA1Ptr and RqA2Ptr.
- **IOFUNCSET**: Sets latch U5 to contain the value pointed to by the sole argument in the call to `P_FSENSE`.
- If not supported then call `IoRoot` system service.

The Non-Mandatory LDD Functions

The eight functions described above must be supported by all device driver. Typical Psion LDDs employ at least two other LDD functions. The first of these is a pseudo-interrupt routine which is invoked by the OS on each system tick. In A4EXIF.LDD, this function is represented by A4ExifTickInt and an outline of its purpose was presented earlier in the discussion on holding and resuming device drivers. As was indicated then, the minimum functionality required by this routine is to poll the status of the host machine doors and if appropriate call the driver's Hold and Resume vectors. If in addition the LDD is intended to service any asynchronous I/O requests, a wait handler and common interrupt service routine are required. With A4EXIF.LDD, these two additional functions are represented by A4ExifHandler and ComInt respectively. The former function constitutes the second of the additional LDD functions as can be seen by examining the LibEnt structure for A4EXIF.LDD. The latter function is set up so that it replaces the default interrupt service routine code invoked by the OS on the receipt of hardware interrupts on the peripheral's SIBO channels. A4ExifHandler is invoked by the OS every time the I/O semaphore of the process that opened the channel is signalled with through an `IoSignalByPidNoResched` call indicating that an interrupt has been received and an outstanding I/O request has completed. The function of the wait handler is to deal with I/O semaphore signalling and to update the appropriate open channel control block field variables. The common interrupt service routine code in ComInt cannot do this because it is not permitted to access the application data space. This in turn is because the routine runs in the context of the process which was running when the interrupt occurred. A4ExifTickInt, the wait handler and the common interrupt service routine for A4EXIF.LDD as well further details as to how the latter two interact to handle asynchronous I/O requests are described below.

A4ExifTickInt

As was explained earlier in discussing the A4ExifHold vector, it is necessary to utilise the "`TCK:`" system tick device driver in order to handle the particular case of invoking a hold on A4EXIF.LDD when the host machine SSD pack doors are opened. Such an action automatically causes power to be removed from any peripherals. The TCK driver is opened via an OS `IoOpen` call in the A4ExifOpen vector which returns with carry clear and a non-zero handle in AX if successful. In this case, the `IoWithWait` function is called to request an synchronous I/O service from the TCK driver. Prior to this call, the service number requested is held in AL and the I/O handle procured from the previous call to `IoOpen` is loaded into BX. The interrupt number of the tick poll routine, A4ExifTickInt, is loaded into CX so that as its parent driver is invoked on each system tick, the OS enters this vector. The reason the TickInt routine is required in A4EXIF.LDD is solely to handle the problem of recognising a sudden removal of power from the host machine. The OS has no way of testing for this condition other than polling its global '`DoorStatus`' variable every system tick and taking the requisite course of action. As such, the outline of the code for the tick poll routine which is required in all Psion LDDs can be presented below. This outline should be carefully compared with the structure of the Hold and Resume vectors shown earlier in order to see how they interact:

- Check the current status of doors which is held in SI.
- If DoorIsOpen then if global `HoldFlag` is zero, force a call to A4ExifHold, set `HoldFlag` to 2 and return far
  else do a far ret
- Else if DoorIsClosed then
  if global `HoldFlag` is 2, force a call to A4ExifResume and return far
  else do a far ret

It should be emphasised that A4ExifTickInt is only a pseudo interrupt service routine in that it does not preserve the status of the registers which is something that the device driver writer must undertake to ensure is done explicitly.

The handling of synchronous and asynchronous I/O

The provision of synchronous and asynchronous I/O services are an important aspect of writing any device driver. In the case of Psion drivers, all such services are the provided through the Strategy vector which is typically preceded by a Strategy Vector Table listing the complete set of I/O services that can be invoked on the driver. The strategy vector table for A4EXIF.LDD is presented below to provide an indication as to the kind of service routines that can be written:

```
StrategyVectorTable                label            word
```

```
        dw      offset A4ExifDefault      ;StrategyPanic
        dw      offset A4ExifRead         ;StrategyRead (function define is P_FREAD)
        dw      offset A4ExifWrite                ;StrategyWrite (function define is P_FWRITE)
        dw      offset A4ExifClose                ;StrategyClose (function define is P_FCLOSE)
        dw      offset A4ExifCancel       ;StrategyCancel (function define is P_FCANCEL)
        dw      offset A4ExifDefault      ;StrategyAttach
        dw      offset A4ExifDefault      ;StrategyDetach
        dw      offset A4ExifSet          ;StrategySet (function define is P_FSET)
        dw      offset A4ExifSense                ;StrategySense (function define is P_FSENSE)
```

Both synchronous and asynchronous I/O requests can be made on the driver through means of the PLIB '`p_iow(pcb,<func>,&A1,&A2)`' call where `pcb` is the open channel handle, `<func>` is one of the above function defines and `A1` and `A2` are two optional parameters required for the servicing of the request. The routing of an I/O request to the correct service code is done by the common code at the head of the strategy vector. Each time this vector is entered, BX holds the address of the open channel control block, DX the device handle of the device driver and SI a pointer to the strategy RqEnt struc used by the OS to follow the course of the I/O request. Using this information, it is possible to access all the relevant flags in the channel StatusEnt by moving [bx].A4ExifStatusEntPtr into DI. This leaves us with the open channel control block in DS:BX and the channel StatusEnt in CS:DI.

Armed with the above information, it is possible to outline the general pattern of both synchronous and asynchronous I/O requests at an OS level. The case of synchronous requests is particularly straightforward since the servicing of the request can be completed entirely within the relevant strategy vector table function. It should be emphasised that in the outline presented below, disabling and re-enabling of interrupts only actually has to be done around any I/O.

```
        SYNCHRONOUS I/O REQUEST STRATEGY VECTOR TABLE FUNCTION
        pushf
        cli                                     ;Disable interrupts
        < request service code >                ;Relevant synchr. I/O processing
ExitWithCompletionStatusZero:
        < set I/O request status word to zero >
        IoSignal                                ;Signals completion of I/O to OS
        xor ax, ax
        clc                                     ;Tells OS that strategy exited OK
        popf                                    ;Re-enable interrupts
        ret
        ProcEnd noret
```

Note that in the above outline, the single `IoSignal` call is made after completion of the I/O request in order to signal this fact to the OS. The situation is somewhat more complicated in the case of an asynchronous request because the I/O request strategy vector table function has to interact with both a wait handler function and an interrupt service routine. Of the I/O request services outlined above for A4EXIF.LDD, only one, namely A4ExifRead, is asynchronous and it serves as a good illustration of the nature of programming for asynchroneity. The steps taken in the A4ExifRead vector are outlined below:

- Set the `ChannelReadCompleted` flag to 0 (stored in cs:[di].A4ExifChanReadCompleted)
- If the address of the I/O request status word is non-zero then panic (i.e. there is an outstanding read request)
- Update the open channel control block variables using the RqEnt struc
- Load I/O request status word with `PendingErr`
- Enable the wait handler by using the `IoEnableHandler` OS service

The read request is now in a pending state waiting for completion via receipt of a hardware interrupt. When such an interrupt is received, the OS switches into the appropriately configured common interrupt routine ComInt. Here, the `ChannelReadCompleted` flag is set to indicate to the handler that the read request has completed. An `IoSignalByPidNoReSched` OS service call is also made to indicate to the OS that an as yet unspecified I/O request has terminated. This signal is consumed by the OS which means that at this stage, an `IoSignal` for completion of the original asynchronous request has yet to be made. The interrupt service routine exits after sensing the status and LED byte values from the ASIC4 Example Interface Board and loading the appropriate response bytes into the relevant locations in the device driver code segment. More details as to the functionality of the interrupt service routine are provided in the section on interrupts. The OS now invokes all the currently active wait handlers in order to determine which application is responsible for consuming the completed read. The wait handler code includes the all-important `IoSignal` call that indicates to the OS that the original asynchronous read request has finally been accounted for. Receipt of this signal by the OS constitutes completion of the request. The outline code for the A4EXIF.LDD wait handler A4ExifHandler is outlined in the next section.

A4ExifHandler

A WaitHandler is a special function of the LDD which is nominated to be called by the operating system every time the I/O semaphore of the process that opened the channel is signalled. It will only actually be called if the application is already waiting for an outstanding I/O request to complete (i.e. while the application is hung in `p_iowait`). The handler function is the means by which hardware interrupts can be filtered through to the process which opened the I/O channel since it permits the rescheduling of processes.

The WaitHandler is invoked in the open vector by using the `IoAddHandler` system service. When an application makes an I/O request, the I/O semaphore is signalled and the operating system calls the Wait Handler function while the application is waiting for the request to complete. The wait handler returns with CLC if no outstanding request has completed. Otherwise the wait handler returns with STC and zero in AL. The wait handler can include within it synchronous I/O requests to cancel or use up signals but it is not entered recursively (i.e. re-enterently) by the operating system. A wait handler is best viewed as a necessary requirement in dealing with the I/O semaphore signalling which cannot entirely be addressed in the appropriate interrupt service routine because the latter is not able to access the open channel's control block.

Interrupts and Interrupt Service Routines

The SIBO architecture allows for eight separate hardware interrupt sources. The EPOC operating system provides a `GenSetRevector` service to enable a device driver to install an interrupt service routine for any of these eight interrupt sources. When an interrupt occurs, the operating system preserves the state of all the registers before calling the appropriately-installed interrupt service routine. As a result of this, the interrupt service routine is free to employ any register it sees fit to use. A few important points ought to be made concerning the code within an interrupt service routine:

- On an 8086 processor, interrupts cannot be nested so their is no requirement to disable interrupts whilst inside the interrupt service routine.
- Interrupt service routines should operate as fast as possible. In general, operating system service routines are tuned to be of less than 1ms duration.
- Interrupt service routines run in the context of the process that was running at the time of the interrupt and should access only the device driver data space which, as with A4EXIF.LDD, typically resides in its own CS space.
- Interrupt service routines should not cause a process rescheduling. To indicate that an event has occurred to the interrupted process, the `IoSignalByPidNoResched` system service should be called. If a call is made to this service, the interrupt service routine should exit with CLC otherwise it returns with STC.

The interrupt routines called require knowledge as to the particular hardware SIBO channel being used by the driver at the time an interrupt occurs. This information is passed through to the `GenSetRevector` function in StartInterrupts by loading the address of the appropriate `IntVec` routine from the A4ExifChannelIntVec field of the channel StatusEnt struc. As a result, when an interrupt occurs, the currently loaded `IntVec` routine is invoked. All the `IntVec` routines reload DI with the appropriate address of the channel StatusEnt in the device driver code segment before falling through to the common interrupt routine service code held in ComInt.

ComInt

The common interrupt service routine code resides in the ComInt vector. The structure of this code is presented below. It should be emphasised that no data in the application data space and hence in the open channel control block can be accessed from ComInt. This explains why it is necessary for the OS to invoke a wait handler to clean up after the interrupt service routine code for a particular asynchronous I/O request has been run.

- Check the value held in the `ChanReadCompleted` flag
- If this value is 2 then we do not have an asynchronous read completed but rather the interrupt line has been pulled low outside of an asynchronous I/O request.
- If this value is 1 then we have an asynchronous I/O request completed and so `IoSignalByPidNoResched` must be invoked to signal to the OS that a read has completed.
- In both cases, sense the status and LED bytes from the Example Interface hardware and load these values in the device driver's code segment A4ExifStatusEnt struc.
- There are four possibilities of status byte corresponding to the four combinations offered by the two input switches, S1 and S2. These are used to output one of four separate LED bytes to the LEDs.
- A write to **A1NonSpecificEoi** register (or **A9BNonSpecificEoiW** in the case of an ASIC9 machine) must be done prior to exiting to let the OS know that the installed interrupt service routine has terminated.

Other important local device driver functions

The functions presented in this section are local to A4EXIF.LDD and perform various important tasks relating to the peripheral hardware such as groping the hardware and setting interrupts running. Since these tasks have to be undertaken by most device drivers, the relevant functions are outlined in greater detail.

A4ExifCheckHardwarePresent

Every Psion device driver has a characteristic 'CheckHardwarePresent' or 'GropeHardware" function that usually involves checking for the presence of the required peripheral device by means of comparing a couple of ASIC4 identification bytes returned by the peripheral in response to a SIBO serial protocol identification message with the expected answers. The specific value of the InfoByte is unimportant as long as a "`test al, al`" call is made which will always return a non-zero answer if ASIC4 is present. In that case, the extended info byte should be tested for against what is expected from address lines A23-A27.

Psion Workabout/HC machines have three physical serial links and two of these are accessed via the SSD slots. The third link corresponds to the serial expansion port. In order for a driver to talk to a peripheral connected to this port, it must specifically select this serial link. In order to prevent multiple access to a single peripheral, it is necessary in the first instance to check whether the serial channel is free or not. A call to the operating system function `HwGetChannel` with the appropriate interrupt mask in AL returns with CLC if the channel was captured successfully. Logically this call should be made in the Open vector but since it can only sensibly be made after the hardware has been successfully located it can be placed at the end of the GropeHardware function. A corresponding call to `HwFreeChannel` must be made when closing the channel. To actually select a channel prior to communication along it, a call to `HwSelectChannel` is necessary.

A4ExifStartChannelRunning

This function is called at the end of the Open vector to set the ASIC4 Example Interface hardware running. It can also be invoked by the Hold vector. The function first checks the status of the global StatusEnt `ChannelRunning` flag. If this flag is clear, CheckHardware is called to determine whether the hardware is still connected and then StartInterrupts is called before the `ChannelRunning` flag is set to indicate that the channel hardware has been started. The function returns with CLC on successful completion or with STC if no hardware was located.

A4ExifStopChannelRunning

This function simply clears the `ChannelRunning` flag if set and calls StopInterrupts unless the channel already has its `ChannelRunning` flag clear. The StopChannelRunning function will be called by Strategy vector when trying to close a channel and can also be called by the Resume vector.

A4ExifStartInterrupts

StartInterrupts is called whenever a call is made to StartChannelRunning. At the heart of the function is a call to the OS `GenSetRevector` function which is used to install a specified interrupt service routine in the place of the default routine held in the interrupt vector table. `GenSetRevector` is called with the interrupt vector number of the service to be replaced in AX and the offset and code segment of the replacement interrupt vector service in BX and CX respectively. BX is loaded from the ChannelIntVec field of the channel StatusEnt which holds the name of the specific interrupt service routine code (i.e. IntVec0, IntVec1 etc.). The latter functions set up the appropriate channel-specific variables before falling through to the common ComInt routine. The channel interrupt mask which is machine and channel dependent is written to the appropriate register on the host ASIC. In the case of ASIC9, this register will be **A9BInterruptMaskRW**. For ASIC1 it will be **A1InterruptMask**.

A4ExifStopInterrupts

StopInterrupts reinstalls the original interrupt service routine by invoking the `GenResetRevector` OS system service and signalling the interrupt mask to the corresponding host ASIC register.

Structures and Include files

Epocdef.inc and device driver strucs

Epocdef.inc is an important header file that contains the definitions of various strucs extensively used in the construction of device drivers. It also contains the definitions of all the OS error values such as `DeviceErr` and `NameErr` that are used in A4EXIF.LDD to communicate an error to the application via AX. Listed below and overleaf are the key strucs defined in Epocdef.inc that were used in the coding of A4EXIF.LDD.

```
LibEnt struc                                Start of device driver
LibSignature         dw     ?
LibInfo              dw     ?
LibCount                    dw     ?
LibBase              dw     ?
LibEnt ends

OpenEnt struc                               Pointed to by SI in Open vector
OpenNamePtr          dw     ?
OpenMode                    dw     ?
OpenChan                    dw     ?
OpenEnt ends
ChanEnt struc                               Used in open channel control block
ChanSignature        dw     ?
ChanNext                    dw     ?
ChanLibHandle        dw     ?
ChanEnt ends

IntEnt struc                                Pointed to by BP in Open and Strategy vectors
IntFrame                    dw     ?
IntBP                dw     ?
IntES                dw     ?
IntDS                dw     ?
```

```
IntPC               dw     ?
IntCS               dw     ?
IntFLAGS                   dw     ?
IntEnt struc ends

RqEnt struc                       Pointed to by SI in strategy vector
RqFunction          dw     ?
RqA1Ptr             dw     ?
RqA2Ptr             dw     ?
RqStatusPtr         dw     ?
RqEnt struc ends
```

A4EXIF channel strucs

These are defined at the start of the A4EXIF.LDD code segment. The open channel control block is held in a structure of type A4ExifEnt and is allocated space from the heap of the process invoking the device driver's open vector. As such, it cannot be accessed from the Hold, Resume and Reset vectors or from the interrupt service routine, ComInt. The channel StatusEnt holds various key channel related flags. Each hardware channel has its own StatusEnt which is instantiated in a header in the device driver code segment following the LibEnt structure. This header block is usually held in CS:DI which is how the StatusEnt variables are accessed. The structure of both A4EXIF channel strucs is shown below:

```
A4ExifEnt struc                              OPEN CHANNEL CONTROL BLOCK
A4xifIo                ChanEnt<>
A4ExifHandlerPtr       dw     ?      Pointer to the Wait Handler
A4ExifStatus           dw     ?      Address of I/O request status word
A4ExifA1Ptr            dw     ?      Pointer to first argument
A4ExifA2Ptr            dw     ?      Pointer to second argument
A4ExifStatusEntPtr     dw     ?      Pointer to channel's StatusEnt
A4ExifTickHandle              dw     ?      Handle to the TCK device channel
A4ExifEnt ends

A4ExifStatusEnt struc                CHANNEL STATUS_ENT
A4ExifChannelPid              db     ?      Process id of process holding
channel open
A4ExifChannelOpen      db     ?      Is the channel open or not?
A4ExifChannelRunning   db     ?      Is the channel running?
A4ExifChanReadCompleted       db     ?      Flag used to indicate to handler
when
A4ExifChannelIntMask   db     ?      Contains the HW channel interrupt mask
A4ExifChannelIntNum    db     ?      Holds no. of interrupt vector to be
replaced
A4ExifChannelSelect    db     ?      SIBO channel select flag
A4ExifChannelDummy     db     ?      Spare
A4ExifChannelIntVec    dw     ?      Holds replacement int. vector routine
name
A4ExifStatusEnt ends
```

Ossibo.inc and Ospack.inc

Ossibo.inc is an important header file that contains the defines for all the ASIC2 and ASIC9 register addresses. Ospack.inc contains similar information but for ASIC4.

12. AN EXAMPLE DEVICE DRIVER FOR ASIC5: SYS$AS5.PDD

Introduction

In this chapter, the functionality and code structure an example installable physical device driver, SYS$AS5.PDD, is presented in some depth. Circuit details of the corresponding Psion peripheral for this driver, the Psion 3-link, were outlined in chapter 8. SYS$AS5.PDD enables software written using the Psion SDK to communicate with the 3-link peripheral through means of standard PLIB calls such as 'p_open' and 'p_write'. This peripheral incorporates an ASIC5 as the SIBO serial protocol slave device. As explained earlier in chapter 7, ASIC5 has an on-board 16550 UART which means that it can be used in conjunction with the standard serial LDD provided an appropriate PDD is loaded. SYS$AS5 is a PDD intended for this purpose and its description in this chapter provides a clear insight into the generalised structure and construction of physical device drivers for peripherals based around the SIBO architecture.

The LDD-PDD interface

All Psion device drivers have a LibEnt structure at the head of their code segment which includes a vector table defining the functionality of the driver. In the case of SYS$AS5, four vectors are defined. These are: Install, Remove, Open and Strategy. The structure of the first three of these are fixed for most device drivers but the fourth can be specified in any way. This fourth vector defines the LDD-PDD interface and is constructed to allow the two drivers to best communicate with each other. In the case of serial PDDs, the approach taken is modelled on the LDD strategy vector with the corresponding strategy vector table and function numbers. This is a logical choice but it is important to emphasise that the LDD-PDD interface is completely user-definable and that the approach described in this chapter is optional.

The key feature of any LDD-PDD interface is that the LDD must have no explicit concept of hardware. In the case of the serial LDD, for instance, it knows that it has a serial port that it can read data bytes from or write data bytes to but it is ignorant of the explicit implementation of the hardware at this port. That aspect is handled by the corresponding PDD which handles all the specifics of data byte I/O. In the case of an application writing a buffer to the serial port by means of a `p_iow(P_FWRITE....)` call, for instance, the LDD will first copy the data in the application's DS into a local CS buffer. It will then call the PDD to indicate that it intends to start sending data bytes when the PDD is ready to start receiving. When the PDD is ready to commence sending a byte out of the serial port, it calls a function (TransmitByte) in the LDD. This function reads the next byte to be transmitted from its local buffer and hands it to the PDD. The PDD duly transmits the byte and again calls TransmitByte until all the bytes are completed. The LDD then sends a special 'NothingToSend' signal to the PDD which indicates that transmission is finished. The LDD also sends an `IoSignalByPidNoResched` which causes its wait handler to be invoked by the OS. In this way, the `IoSignal` that signifies completion of the original write request is invoked by the LDD. Note that the LDD has no concept of interrupts, merely of sending a byte at a time and registering completion or otherwise of I/O requests. Futhermore, the PDD never takes the initiative from the LDD and merely undertakes one function at a time before returning control back to the LDD. The LDD-PDD interface is examined in greater depth later in this chapter when the structure of the PDD strategy vector is presented.

Code Structure

Device Names and Loading SYS$AS5.PDD

SYS$AS5.PDD is loaded into the RAM of the host machine by means of the following PLIB call in the application code:

```
p_loadldd("SYS$AS5.PDD");
```

The name of a device driver is the mechanism by which an application can obtain a channel to that device driver. LDDs all have three character names followed by a period, a further three characters and a colon. The first three characters of a PDD name are the name of the LDD to which the PDD belongs. The second set of three characters uniquely identify the PDD. The device name is required to uniquely identify the LDD to the OS when attempting to open a channel on it. The SYS$AS5 PDD belongs to the "TTY:" LDD. Its name as defined in its LibEnt structure is 'TTY.SR5'. Thus a channel with its handle in `pcb` may be obtained on it at the application level by means of the following call:

```
p_open(&pcb,"TTY.SR5:A",-1);
```

The qualifier after the colon indicates that the driver can support more than one channel. Channels are allocated a single character sequentially from the character 'A' up to the character 'C'. The number of channels that can be supported in this way is dependent upon the host hardware. Only one expansion port can be opened on the S3a for instance whereas three are possible on the Workabout and HC. EPOC uses the driver name in the `p_open` call to invoke the `IoOpen` system service which in turn invokes the Open vector on the associated device driver.

The Single Code Segment and Data Storage

All data associated with a physical device driver must be stored in its code segment. Examples of such data are the channel interrupt masks and numbers that have to be visible to all processes that may be using SYS$AS5.PDD.

The overall structure of the code segment is typical of Psion physical device drivers. The segment begins with a CodeSeg directive followed by the LibEnt structure which defines all the device driver functions. Before entering the code for these functions, all global variables are declared in the internal (CS) data space. Following the code for the LibEnt functions comes the code for all the local driver functions, After these are the EndCodeSeg and end OsAs5PDD directives.

The Channel struct and A5Ent struct

At the head of the driver's assembler source file, SYS$AS5.ASM, various constants, compiler defines and types are listed. It is here that templates are declared for the global structures that hold the key driver variables. in the case of SYS$AS5, one main structure, the Sr5ChannelStruct, is employed to hold the various important flags and masks that relate to the status of each separate channel on the driver. This structure is termed the **Channel struct** and its fields are filled in during the course of running the PDD install and open vectors. Whenever the serial LDD invokes one of the PDD functions, the application must ensure that CS holds the address of the PDD's code segment which is where the Channel struc resides. The channel's Channel struct is usually accessed through DI or BX depending on preference. Its layout is presented later in this chapter along with other important defines that were used in the construction of SYS$AS5.PDD.

SIBO hardware and conditional compilation

As indicated previously, the number of expansion channels supported by a host Psion machine is dependent on the hardware. In the case of the S3a, only one SIBO channel can legitimately be opened corresponding to expansion port A. With the HC and the Workabout, it is possible to open up to three separate SIBO channels on ports A through to C. SYS$AS5.PDD is constructed to enable it to run on any host Psion platform. In its internal data space, the various hardware options for the SIBO channels, interrupt masks and interrupt numbers are coded in a large if statement thereby permitting conditional compilation of the driver for the required host hardware. The only change that need be made in adapting it for use on another Psion machine is the alteration of the build flag at the start of the code segment. This flag indicates to the compiler which SIBO machine flags as well as other variables should be set. The conditional compilation table outlined here is worthy of some study because it encapsulates all the information regarding masks and SIBO hardware channels required by the prospective developer interested in constructing a multi-platform peripheral and accompanying PDD.

The PDD Functions

OsAS5Install

IN:	Nothing
OUT:	If successful, return with carry clear
	If installation unsuccessful, return with the carry flag set and error number in the
AL	register
PRESERVE:	SS, SP, BP

The install vector is called by the parent LDD whenever the PDD is required to be loaded. It cannot be called directly be an application only indirectly through the LDD in its Install vector. The install vector is called in the context of the OS with DS and ES in an unknown state. Memory will not be moved while in this function so the normal rules governing the use of ES and DS may be ignored.

OsAS5Remove

IN:	Nothing
OUT:	If successful, return with carry clear
	If installation unsuccessful, return with the carry flag set and error number in the
AL	register
PRESERVE:	SS, SP, BP

The remove vector is called by the parent LDD whenever the PDD is required to be unloaded. It cannot be called directly by an application and only indirectly through the LDD in its Remove vector. The remove vector is called in the context of the OS with DS and ES in an unknown state. Memory will not be moved while in this function so the normal rules governing the use of ES and DS may be ignored.

OsAS5Open

IN:	SS:SI points to the OpenEnt structure
	ES, DS point to the DS of the application process.
OUT:	If successful, return with carry clear and control block in BX
	If installation unsuccessful, return with the carry flag set and error number in the
AL	register
PRESERVE:	SS, SP, BP

The open vector is called by the parent LDD whenever a channel to the PDD is required to be opened. It cannot be called directly by an application and is typically invoked in the higher level LDD Open vector code by means of the `DevOpenPDD` OS system call. On invoking the open vector, SI points to the OpenEnt structure which contains three fields, namely OpenNamePtr, OpenMode and OpenChan. The OpenNamePtr points to the qualifier immediately following the device name in the call to `p_open`. In the case of SYS$AS5, for instance, the OpenNamePtr would point to the '`A`' in the name "`TTY.SR5:A`". This corresponds to an attempt to open the first hardware SIBO channel on the host machine which if successful will leave the address of the Chan0 Sr5ChannelStruct in BX. The PDD open vector finally includes a call to the OS service `HwGetChannel` to obtain the requested SIBO channel.

The order of action undertaken by the parent serial LDD's open vector is generally fairly complicated and includes various calls to the PDD strategy vector routines. The situation is outlined overleaf:

- Ensure that the LDD itself can be opened.
- Call the PDD open vector using the OS DevOpenPDD OS call.
- If successful, call the OS service `DevGetPDDAddress` which returns the full segment:offset address of the PDD's fourth strategy vector in BX:AX. These values are loaded into the dword Channel struct field SerialPDDEntry.
- Invoke the PDD strategy Open and SetHandlerCS functions to set up offsets and segments respectively to locations in the LDD above.
- Invoke strategy Set to initialise the transmission baud rate.
- Invoke strategy Start to set-up and then enable interrupts

Once the DevGetPDDAddress service has been used to load the LDD's SerialPDDEntry field, it may be used to load in the address of any of the PDD's strategy functions. The DevGetPDDAddress OS call invoked in the LDD open vector must also be invoked in the LDD resume vector code since memory may have been moved while the LDD was held. The PDD is ignorant of such activity since the serial LDD is responsible for handling all holds and resumes.

OsAS5Strategy

IN:	AX holds the vector number
	ES, DS point to the DS of the application process.
OUT:	If successful, return with carry clear and control block in BX
	If installation unsuccessful, return with the carry flag set and error number in the
AL	register
PRESERVE:	SS, SP, BP

The strategy functions are invoked directly from the various serial LDD vectors to provide hardware-specific services. For instance, in order to set up the baud rate, it is necessary to invoke strategy Set. The strategy vector table for OsAS5Strategy is presented below and then the functionality of the important component vectors is outlined:

```
AS5StrategyJumpTable label word
       dw     offset AS5Open             ;Load Handler offsets
       dw     offset AS5Close                   ;Close the channel
       dw     offset AS5Start                   ;Start the channel
       dw     offset AS5Stop             ;Stop the channel
       dw     offset AS5Set              ;Set the channel status
       dw     offset AS5Sense                   ;Return channel status
       dw     offset AS5Control          ;Drive the handshaking lines
       dw     offset AS5Enquire          ;Returns baud rate
       dw     offset AS5Enable                  ;Begin sending output
       dw     offset AS5SetHandlerCS            ;Load Handler segments
```

Open and SetHandlerCS go together and are invoked from the LDD open and resume vector code. Both are required to let the PDD know the address at which its LDD resides. The four full addresses passed through to Open and SetHandlerCS are of the LDD's control block and the LDD StatusInt, RecvInt and XmitInt vectors. Close is called in order to close an opened channel. Start and Stop are used to enable/disable interrupts and are invoked from the LDD open, hold/resume and set vectors. The order of action in the PDD Start vector is as follows:

- Initialise PDD variables
- Check H/W present
- Start the hardware running. In the case of SYS$AS5, part of this process involves starting a continuous clock from the host controller ASIC in order to trigger the UART clock on the 3-link's ASIC5. In addition to this clock, the RTS and DTR lines must be driven low.
- Start interrupts.

The common interrupt service routine code resides in the PDD ComInt function. This code is patched into the interrupt vector via the GenSetRevector system service. The interrupt mask contains bits to generate the following interrupts: Receive character, Ready to send next character and Modem line change character. In SYS$AS5, all three interrupts are enabled in Start. It is important to realise that after masking in these bits, a Ready to send next character interrupt is almost immediately generated so the corresponding ComInt code must be able to handle this.

CheckHardwarePresent

This non-mandatory function is called from the Open vector and follows the lines of previously discussed CheckHardwarePresent code. After checking for an ASIC5 at the end of the serial link, the function returns with the carry flag clear if one is found. If a non-zero info byte is returned with an Asic5NormalId, then various other IDs are tried (Asic4Id, Asic8Id, Asic5PackId) before returning with the carry flag set.

CheckHardwarePresentFromStart

This is a non-mandatory function called only from the PDD Start vector. After checking that we have an ASIC5 at the end of the SIBO channel, the function sets the S_PERIPHERALMODE bit of the A5PortBMode register. This then puts ASIC5 into UART mode.

ComInt

The common interrupt routine code is entered with DI holding the address of the appropriate channel struct. ASIC5's control register (A5CtrlReg) is read first to determine which interrupt has occurred. The byte read from the register is compared against S_MDINT (Modem lines interrupt), S_RXINT (Receive character interrupt) and S_TXINT (Transmit interrupt). The code to handle each of these cases is then entered prior to returning control back to the LDD via the Channel struct fields that were filled by a previous LDD calls to the PDD strategy Open and SetHandlersCS vectors. The LDD has no knowledge of interrupts and the purpose of ComInt is therefore to hide the hardware details of handling interrupts from the LDD. Once the interrupt has been serviced, a write is made to A9BNonSpecificEoiW in the case of ASIC9 or A1NonSpecificEoi for an ASIC1/ASIC2 based system to indicate to the OS that the interrupt has been serviced.

13. DEBUGGING AND TESTING DEVICE DRIVERS

Introduction

This chapter will detail the techniques that can be used by the developer to first debug and then test device drivers. The emphasis will be on the software methods such as:

- Good use of variables. e.g. Starting all the fields in the CS Channel Status structure with CS.
- Debugging by eye. Even more important with regard to device drivers.
- Using SDBG. The pitfalls and benefits. Working your way around the strategy vector calls with SDBG.
- Construction of C test harness programs. Catching all the error flags that can be returned by device driver functions.
- The comprehensive memory check program mem.c.

Debugging Techniques

A Psion device driver is written in 8086 assembler as an asm file and built using the Borland Turbo Assembler compiler. The subsequent debugging process centres around the construction of an appropriate PLIB test harness. The purpose of a test harness is to check a number of the device driver vectors to ensure that they do not return errors or cause panics. The majority of PLIB calls that would be used in this context return a negative integer that is used to ascertain the cause of the problem in the corresponding device driver vector. The code below, for instance, would be used to test the install, open, strategy close and remove vectors of the A4EXIF LDD:

```
GLDEF_C VOID main(VOID)
{
VOID *serH;
INT ret;

if ((ret=p_loadldd("A4EXIF.LDD"))<0)
        {
        p_printf("Error %d on p_loadldd",ret);
        p_getch();
        p_exit(0);
        }
else
        p_printf("Successfully loaded A4EXIF.LDD");
if ((ret=p_open(&serH,"LED:",-1))<0)
        {
        p_printf("Error %d on p_open",ret);
        p_getch();
        }
else
        {
        p_printf("Successfully opened LED: channel");
        p_close(serH);
        p_printf("Successfully closed LED: channel");
        }
p_devdel("LED",E_LDD);
p_getch();
p_exit(0);
}
```

If as often happens at this stage, a bug in the device driver causes the test harness program to crash, it is necessary to debug the driver code either by eye or using SDBG. Debugging by eye should always be the first resort, however, whenever the bug can be readily pinned down to a particular vector. Things to look out for include:

- An unbalanced stack: Check that the number of 'pushes' equals the number of 'pops' in the appropriate vector code.
- Addressing the wrong location: Ensure that all the Channel status fields (stored in CS) are offset using the correct register and that that register holds the right value. The following code for instance requires DI to hold the address of the channel status struct prior to invocation:

  ```
  mov al, cs:[di].ChannelOpenFlag
  ```
- Consistency: With regard to the last point, it is important to be consistent if possible and try and use the same register (i.e. DI) to hold the channel status structure. A different one (usually BX) should also be used to hold the address of the open channel control block.
- Trashing BX and DI: If these registers are used in the vector code for anything other than addressing, check that their value is not being trashed by the operation.

Debugging using SDBG is an extension of debugging by eye. The SIBO debugger allows the programmer to trace through the driver's source code instruction by instruction and observe the contents of the CPU registers in the process. In this way it is possible to discover any discrepancies in terms of the values stored in the various registers. Furthermore, tracing with SDBG will enable the user to pinpoint the source of panics. The first objective of the device driver debugging process should be to get the code outlined on the previous page to work OK.

Further Testing Strategies

Once installing, opening, closing and unloading are dealt with, a test harness can be expanded to include `p_iow` or `p_ioc` calls which map onto the LDD's strategy vector. At this stage, SDBG is particularly useful for testing purposes as breakpoints can be set and jumped to. In this way for any `p_ioc(serH,<func>,&serStat,&A1,&A2)` call, the contents of the status word, `serStat`, and the `A1` and `A2` parameters can be tracked through the LDD's strategy vector. The value held by the status word at the end of a particular strategy call is the value returned by the corresponding PLIB `p_iow`/`p_ioc` call. Thus negative errors within the strategy code can be picked up by the test harness. A good test harness should be able to catch all the possible errors and at the least invoke `p_printf` to let the user know when one of them is returned. The file p_file.h contains a list of all the current return error values and the corresponding PLIB level error name. The entry for `PendingErr`, for instance is as follows:

```
#define PendingErr          (-46)
#define E_FILE_PENDING              PendingErr
```

In order to induce the return of these error values it is necessary to extend the basic test harness outline to allow the user to undertake various pathological actions. For instance, the harness may include code that tries to open a channel twice which should result in `InUseErr` being invoked. By such means it is possible to ensure that a driver is not only operating as it should in normal circumstances but returning the correct error value when relevant.

Memory Testing

The final process that should be undertaken to fully test a device driver consists of the construction of an appropriate memory test harness. Three functions are presented on the next two pages which provide the core of such a comprehensive memory test program. The first function, CheckMemory, uses the PLIB routines `p_allspc` and `p_sgfree` to print out the current free bytes on the heap and the number of free segments in the host RAM. This function can be invoked after installing, opening, closing or removing a device driver to ensure that memory is not going to 'alloc heaven'. The GobbleMemory function is first used to determine the amount of free memory available in segments through calling `p_sgfree`. The PLIB `p_sgcreate` function is then invoked to create a new segment, "Test", that consists of all of this free memory. Finally, the `p_sgcopyto` PLIB function is called to fill "Test" with 0x55's. By using up all free memory in this way, it is possible to use segment "Test" to determine whether a particular driver vector is writing to the wrong location. TestGobble, presented overleaf, is the third function outlined. It uses the PLIB function `p_sgcopyfr` to check the values of the bytes in the segment "Test". If any of the 0x55's have been overwritten then we know that we have a problem. These three functions should be incorporated within the standard test harness functions already presented in this chapter. In this way, it is possible to generate a powerful generic test program that can be used as the basis for all device driver testing.

```
LOCAL_C VOID CheckMemory(VOID)
{
VOID *Heap;
INT fbytes;

fbytes=p_allspc(&Heap);
p_printf("Free Heap Memory =>%x bytes",fbytes);
p_printf("Free Segments =>%d",p_sgfree());
p_getch();
}

/***********************************************/

LOCAL_C VOID GobbleMemory(VOID)
{
UINT nParas,segSize;
INT j;
LONG pos,i;
UBYTE buf[256];

p_printf("  System RAM size  = %d",p_getram());
p_printf("Internal RAM usage = %d",p_sgramdisk());
nParas=p_sgfree();
p_printf("Amount of free RAM = %d",nParas);
segH=p_sgcreate("Test",nParas,E_SEGMENT_HIGH);
if (segH)
       p_printf("Created segment \"test\"");
else
       {
       p_printf("Error in creating segment");
       p_getch();
       p_exit(0);
       }
segSize=p_sgsize(segH);
p_printf("Size of segment is %d",segSize);
p_printf("In 16-byte paragraphs");
p_sleep(5L);
for (j=0;j<16;j++)
       buf[j]=0x55;
i=0;
while (i<segSize)
       {
       pos=i*16;
       if (p_sgcopyto(segH,pos,&buf[0],16)<0)
              {
              p_printf("Failed on p_sgcopy");
              p_printf("%d",i);
              p_getch();
              }
       i++;
       }
p_printf("Test segment full of 0x55s");
p_getch();
}

/***********************************************/
```

```
LOCAL_C VOID TestGobble(VOID)
{
UINT segCount;
UBYTE buf[16];
INT i;

segCount=0;
p_printf("Checking segment integrity ...");
for (segCount=0; segCount<p_sgsize(segH); segCount++)
       {
       if (p_sgcopyfr(segH,segCount*16,&buf[0],16)<0)
              p_printf("Error in segment");
       else
              {
              for (i=0;i<16;i++)
              if (buf[i]!=0x55)
                     {
                     p_printf("OVERWRITE ERROR!");
                     p_printf("Segment count=%d",segCount);
                     p_printf("Byte count=%d",i);
                     }
              }
       }
p_printf("Test segment OK");
p_printf("Heap integrity checks OK");
p_allchk(44);
p_printf("If get here, heap OK");
p_getch();
}

/**********************************************/
```

APPENDIX: SOURCE CODE FILES

A4EXIF.ASM

```
title   A4EXIF -- Example ASIC4 Interface device driver
        subttl  Copyright (c) Psion PLC (1994)
        name    A4EXIF
;
; VER     DATE     BY      DESCRIPTION
; ----------------------------------------
; 1.00F  26/1/95   Mal     Working Version


BUILDSB=1                       ;S3a build environment (channels=1)
S3c=0                           ;Need to specify no s3c
;BUILDSC=1                       ;S3c build environment (channels=3)
;BUILDCH=1                       ;HC build environment (channels=3)
;BUILDHH=1                       ;S3 build environment  (channels=1)

        include ..\inc\epoc.inc
        include ..\inc\epoclib.inc
        include ..\inc\epocsibo.inc
        include ossibo.inc
        include ospack.inc

;       Example Logical Device Driver for prototype LED-ASIC4 Interface
;       circuit for Corporate/S3C/Consumer serial port.
;       Written by Mal Dec 1994/Jan 1995.

;       A4EXIF CONSTANTS AND TYPES
;       ==========================
;       The following constants and type definitions
;       are compiler directives used by the TCEP assembler
;       when it is creating the LDD.

if Consumer
        numberofchannels        equ     1
else
  if Corporate or S3c
        numberofchannels        equ     3
  else
        numberofchannels        equ     2
  endif
endif

;Channel StatusEnt block accessed through CS:DI
A4ExifStatusEnt struc
        A4ExifCSProcessId                dw      ?       ;Channel parent process id
        A4ExifCSChannelOpen              db      ?       ;Channel open flag
        A4ExifCSChannelRunning           db      ?       ;Channel hardware running
flag
        A4ExifCSChanReadCompleted        db      ?       ;Channel request completed
flag
        A4ExifCSChannelIntMask           db      ?       ;Channel interrupt mask
        A4ExifCSChannelIntNum            db      ?       ;Channel interrupt number
        A4ExifCSChannelSelect            db      ?       ;SIBO Channel select
        A4ExifCSChannelIntVec            dw      ?       ;Channel int vector number
        A4ExifCSTickHandle               dw      ?       ;TCK channel handle
        A4ExifCSA1Value                  db      ?     ;Channel strategy A1 parameter
        A4ExifCSA2Value                  db      ?     ;Channel strategy A2 parameter
A4ExifStatusEnt ends

;Open Channel Control block accessed through DS:BX
A4ExifEnt struc
        A4ExifDSIo              ChanEnt <>      ;Open channel control block Ent
        A4ExifDSHandlerPtr      dw      ?       ;Cant touch when under
        A4ExifDSStatusPtr       dw      ?       ;Hold, Resume, Reset or
        A4ExifDSA1Ptr           dw      ?       ;interrupt routine
        A4ExifDSA2Ptr           dw      ?
        A4ExifDSStatusEntPtr    dw      ?
A4ExifEnt ends
        A4PERIPH_MASK           equ     0f0h            ;11110000b
        EXTENDED_INFO_BYTE      equ     090h            ;10010000b
        U5_ENABLE_ON            equ     080h            ;Sets LBO for latch U5
        U5_ENABLE_OFF           equ     000h            ;Deselects LBO for U5

;Latch addresses for reading/writing
        U5OUTPUT_LATCH          equ     00000000b       ;Selects A0 for writing
        U3INPUT_BUFFER          equ     00000000b       ;Selects A0 for reading
        U4STATUS_BUFFER         equ     00000001b       ;Selects
cs:[di].A4ExifCsA1Valuefor                                                ;reading
        INTERRUPT_LATCH         equ     00000001b       ;Selects A1 for writing

;Status byte masks
        S2S3_ON                 equ     00000011b
        S2S3_OFF                equ     00000000b
        S2_ONLY                 equ     00000001b
        S3_ONLY                 equ     00000010b
        INTERRUPT_STATUS_MASK   equ     00000011b

;LED bytes
        SOME_LEDS_ON            equ     01010101b
        SOME_LEDS_OFF           equ     10101010b
        TOP_LEDS_ON             equ     11110000b
        BOTTOM_LEDS_ON          equ     00001111b
        ZERO_BYTE               equ     00h

        dgroup  group  stack
        assume ds:dgroup,es:dgroup,ss:dgroup

        CodeSeg

;       A4EXIF ENTRY TABLE
;       ==================

ProcBegin@ A4ExifLDD
;          =========

dw      LDDSignature
db      'LED',0,0,0,0,0
dw      (VectorEnd-Vector)/2
Vector:
        dw      A4ExifInstall
        dw      A4ExifRemove
        dw      A4ExifHold
        dw      A4ExifResume
        dw      A4ExifReset
        dw      A4ExifUnits
        dw      A4ExifOpen
        dw      A4ExifStrategy
VectorHandler:
        dw      A4ExifHandler
if Asic9
InterruptVectors:
        dw      A4ExifTickInt
endif
VectorEnd:


;       A4EXIF INTERNAL DATA SPACE
;       ==========================
;       Device Driver global variables follow.
;       These variables reside in the code segment
;       and as such can always be accessed with
;       the 'cs:' prefix.

if Asic9
  if Consumer
        Channel0        A4ExifStatusEnt<0,0,0,0,mask

        A9MSlave,HwIrq2Revector,SelectChannel5,IntVec0,0,0>
  else
    if Corporate or S3c
        Channel0        A4ExifStatusEnt<0,0,0,0,mask

        A9MExpIntA,HwIrq4Revector,SelectChannel3,IntVec0,0,0>
        Channel1        A4ExifStatusEnt<0,0,0,0,mask

        A9MExpIntB,HwIrq5Revector,SelectChannel4,IntVec1,0,0>
        Channel2        A4ExifStatusEnt<0,0,0,0,mask

        A9MSlave,HwIrq2Revector,SelectChannel5,IntVec2,0,0>
    else
        Channel0        A4ExifStatusEnt<0,0,0,0,mask

       A9MExpIntA,HwIrq4Revector,SelectChannel3,IntVec0,0,0>
        Channel1        A4ExifStatusEnt<0,0,0,0,mask

       A9MExpIntB,HwIrq5Revector,SelectChannel4,IntVec1,0,0>
    endif
  endif
else
  if Consumer
        Channel0        A4ExifStatusEnt<0,0,0,0,mask

       Asic2Int,HwIrq4Revector,SelectChannel7,IntVec0,0,0>
  else
    if Corporate
        Channel0        A4ExifStatusEnt<0,0,0,0,mask

       ExpIntLeftA,HwIrq3Revector,ExpChannelLeftA,IntVec0,0,0>
        Channel1        A4ExifStatusEnt<0,0,0,0,mask

       ExpIntRightB,HwIrq2Revector,ExpChannelRightB,IntVec1,0,0>
        Channel2        A4ExifStatusEnt<0,0,0,0,mask

       Asic2Int,HwIrq4Revector,SelectChannel7,IntVec2,0,0>
    else
        Channel0        A4ExifStatusEnt<0,0,0,0,mask

       ExpIntLeftA,HwIrq3Revector,ExpChannelLeftA,IntVec0,0,0>
        Channel1        A4ExifStatusEnt<0,0,0,0,mask

       ExpIntRightB,HwIrq2Revector,ExpChannelRightB,IntVec1,0,0>
    endif
  endif
endif

        HoldFlag        db      0       ;Hold/Resume flag
        SwitchStatus    db      ?       ;Holds the masked status byte from U4 after
interrupt

        ProcEnd noret


        ProcBegin@ A4ExifInstall,far
;                  =============
;       Installs the device driver.
;       Invoked after a PLIB 'p_loadldd("A4EXIF.LDD")' call to load the LDD
;       All of the fields inside the CS control blocks are preloaded
;       with the correct values at install time. Install therefore does
;       not do any work.
;       IN:
;               Nothing
;       OUT:
;               Carry Clear - driver successfully installed
;
        clc                     ;The ChannelOpen fields are set to zero
        ret                     ;in the relevant ChannelStatusEnt headers
        ProcEnd noret


        ProcBegin@ A4ExifRemove,far
;                  ============
;       Removes the device driver.
;       Invoked after a PLIB 'p_devdel("LED",E_LDD)'call to unload the
;       device driver.
;       A device driver cannot be removed if any of its channels
;       are still open.
;       IN:
;               Nothing
;       OUT:
;               Carry clear - successfully removed
;               Carry set - remove failed, error number in AL
;
        mov     cx, numberofchannels                    ;Check that each
        mov     di, offset Channel0                     ;channel is closed
        xor     ax, ax
CheckAllChannelsClosedLoop:
        cmp     cs:[di].A4ExifCSChannelOpen, al         ;Closed channels will
        jne     WeHaveAChannelOpenSoFail                ;have the value 0 in
        add     di, (size A4ExifStatusEnt)              ;their ChannelOpen
        loop    CheckAllChannelsClosedLoop              ;flags
        jmp     FinishedOkay
WeHaveAChannelOpenSoFail:
        mov     al, InUseErr                            ;Fail if not all
        stc                                             ;closed
        ret
FinishedOkay:
        clc
        ret
        ProcEnd noret


        ProcBegin@ A4ExifHold,far
;                  ==========
;       Called by the operating system whenever the device driver is
;       being moved or the machine is powering down.
;       This will also be called by our pack door polling function when
;       it sees that the doors have been opened and our peripheral has
;       lost power.
;       IN:
;               Reason for the hold in AH
;       OUT:
;               Nothing
;
        mov     cx, 1                                   ;Hold can be called
        xchg    cl, HoldFlag                            ;when the driver is
        cmp     cl, 0                                   ;under a hold so
        jne     AlreadyHeld                             ;re-entrancy blocking
A4HoldFromTick:                                         ;is required
        mov     cx, numberofchannels
        mov     di, offset Channel0                     ;Loop because Hold
HoldAllTheChannelsLoop:                                 ;must stop all the
        cmp     cs:[di].A4ExifCSChannelOpen, 0          ;channels
        je      DontHoldBecauseNotOpen                  ;Is the channel open?
        push    cx                                      ;If it is then stop
        call    StopTheChannelRunning                   ;all interrupts
        pop     cx
DontHoldBecauseNotOpen:
        add     di, (size A4ExifStatusEnt)
        loop    HoldAllTheChannelsLoop
AlreadyHeld:
        ret
        ProcEnd noret


        ProcBegin@ A4ExifResume,far
;                  ============
;       Called by the operating system when it has finished moving the
;       device driver in memory or when the machine is switching back on.
;       Also called by our door polling routine when it sees that the doors
;       have been closed and we can resume communication with our peripheral.
;       IN:
;               Nothing
;       OUT:
;               Nothing
;
        xor     cx, cx                          ;Block in case of re-entrancy
        xchg    cl, HoldFlag
        cmp     cl, 0
        je      AlreadyResumed
if Asic9
        GenDataSegment                          ;Check to see if the pack
        HwGetSsdData                            ;doors are still closed
        mov     bx, ax
        cmp     es:[bx].SsdDoorStatus, DoorOpen
        je      DoorsAreOpen
endif
A4ResumeFromTick:
        mov     cx, numberofchannels            ;Loop because resume must
        mov     di, offset Channel0             ;restart each open channel
ResumeAllTheChannelsLoop:
        cmp     cs:[di].A4ExifCSChannelOpen, 0  ;Is the channel open?
        je      DontResumeBecauseNotOpen
        push    cx
        call    StartTheChannelRunning
        pop     cx
DontResumeBecauseNotOpen:
        add     di, (size A4ExifStatusEnt)
        loop    ResumeAllTheChannelsLoop
AlreadyResumed:
        ret
DoorsAreOpen:
        mov     HoldFlag, 2
        ret
        ProcEnd noret


        ProcBegin@ A4ExifReset,far
;                  ===========
;       Called when an application which opened a channel terminates without
;       closing the device driver.
;       A device driver, for each open channel, has to request that the
;       operating system calls this function when the application
;       terminates abnormally (ie without closing an open channel).
;       Reset can supply one piece of identifying data which will be
;       passed in CX. This would usually be the channel's number or CS
;       control block pointer.
;       Reset just needs to clear interrupts, hardware reservations, and free
;       the channel. Any allocated space will be cleaned up for you by the OS.
;       IN:
;               The device driver's handle in BX
;               The address of the status struc identifying the channel in CX
;       OUT:
;               Nothing
;
        mov     di, cx
        cmp     cs:[di].A4ExifCSChannelOpen, 0
        je      NotOpenToReset
        call    StopTheChannelRunning                   ;Stop interrupts
        mov     al, cs:[di].A4ExifCSChannelIntMask      ;Free the reserved
        HwFreeChannel                                   ;hardware
        mov     cs:[di].A4ExifCSChannelOpen, 0          ;The channel is now
NotOpenToReset:                                         ;free
        ret
        ProcEnd noret


        ProcBegin@ A4ExifUnits,far
;                  ===========
;       Called to find how many open channels the driver will support
;       In:
;               Nothing
;       Out:
;               The total number of channels supported in AX
;
        mov     ax, numberofchannels
        ret
        ProcEnd noret


        ProcBegin@ A4ExifOpen,far
;                  ==========
;       Opens a device driver channel.
;       Invoked after the PLIB call 'p_open(&appHandle,"LED:*",-1)'to open a
;       channel to the device driver.
;       In:
;               OS device handle of the device driver in DX
;               Pointer to the OpenEnt struc in SI
;               Pointer to the IntEnt struc in BP
;               DS,ES,SS point to the applications data space
;       Out:
;               Carry clear - BX holds the address of the open channel
;               Carry set - AL holds the error number
;
        cld                                             ;Interrupts off to
        pushf                                           ;prevent multiple apps
        cli                                             ;calling open
        mov     si, [si].OpenNamePtr                    ;simultaneously
        mov     al, [si+1]
        CharToFoldedChar                                ;Read the unit no.
        cmp     al, 'A'                                 ;part of device name
        jb      OpenNameErr                             ;to find which channel
        sub     al, 'A'                                 ;to open. eg "LED:A"
        cmp     al, numberofchannels
        jae     OpenNameErr
        xor     ah, ah
        push    dx                                      ;Map the channel no.
        mov     dx, (size A4ExifStatusEnt)              ;to a channel control
        mul     dx                                      ;block in our CS
        pop     dx                                      ;space
        mov     di, ax
        add     di, offset Channel0                     ;Offset in DI
        cmp     cs:[di].A4ExifCSChannelOpen, 0
        je      GetATickChannel                         ;Check to see if
        popf                                            ;channel is already
        mov     ax, AlreadyOpenErr                      ;open
        jmp     ChannelAlreadyOpen
GetATickChannel:
        mov     cs:[di].A4ExifCSChannelOpen, 1          ;Obtain a TCK channel so
        mov     cs:[di].A4ExifCSChannelRunning,0        ;we can poll the door state
        popf
ife Asic9
        jmp     GetHardwareChannel                      ;If HC, we don't need
else                                                    ;to set up the TCK
        push    ax                                      ;routine
        mov     ax, ((':' shl 8)+'K')
        push    ax
        mov     ax, (('C' shl 8)+'T')                   ;Try to open "TCK:"
        push    ax                                      ;channel with "TCK:"
        mov     bx, sp                                  ;string on stack
        IoOpen
        jnc     GotATickChannel                         ;TCK will call our
        add     sp, 6                                   ;door-polling function
        mov     ax, LockedErr                           ;32 times a second
        jmp     OpenFailed
GotATickChannel:
        add     sp, 6
        mov     cs:[di].A4ExifCSTickHandle, ax          ;Start our
        push    dx                                      ;door-polling function
        mov     bx, ax                                  ;running by starting
        mov     ax, IoFuncStart                         ;the "TCK:" channel
        mov     cx, 1
        push    cx                                      ;Frequency 1 tick
        push    cx                                      ;Data irrelevant
        push    dx                                      ;Handle to driver
        mov     cx, (InterruptVectors-Vector)/2         ;TCK function to call
        push    cx
        mov     cx, sp
        IoWithWait
        add     sp, 8
        pop     dx
        jmp     GetHardwareChannel
endif
OpenNameErr:
        popf
        mov     ax, NameErr
        jmp     ChannelAlreadyOpen
GetHardwareChannel:                                     ;Check the hardware
        mov     al, cs:[di].A4ExifCSChannelIntMask      ;is available then
        HwGetChannel                                    ;reserve it
        jnc     CheckHardwareNowThatSIBOChannelIsOpen   ;Returns with carry
        mov     ax, InUseErr                            ;clear if OK
        jmp     FreeTckAndExit
FreeTckAndChannelHardware:
        mov     al, cs:[di].A4ExifCSChannelIntMask
        HwFreeChannel
        mov     ax, DeviceErr
FreeTckAndExit:
if Asic9
        push    ax
        mov     bx, cs:[di].A4ExifCSTickHandle
        IoClose
        pop     ax
endif
OpenFailed:
        mov     cs:[di].A4ExifCSChannelOpen, 0
ChannelAlreadyOpen:
        stc
OpenExit:
        ret
CheckHardwareNowThatSIBOChannelIsOpen:
        ProcId                                          ;Set up the calling
        mov     cs:[di].A4ExifCSProcessId, ax           ;process ID
        mov     cs:[di].A4ExifCSChanReadCompleted, 0
        call    CheckHardwarePresent                    ;Returns with carry
        jc      FreeTckAndChannelHardware               ;clear if successful
        mov     cx, (size A4ExifEnt)                    ;Allocate a control
        HeapAllocateCell                                ;block in our app's
        jc      FreeTckAndChannelHardware               ;data space
        mov     bx, ax
        push    bx                              ;IoAddHandler trashes BX
        mov     al, (VectorHandler-Vector)/2    ;Set up our wait handler
        IoAddHandler                            ;Leaves the address of
        pop     bx                              ;handler in AX
        jnc     GotHandler
        push    ax                              ;If no handler, open fails
        HeapFreeCell
        pop     ax
        jmp     FreeTckAndChannelHardware
GotHandler:
        mov     [bx].A4ExifDSStatusEntPtr, di
        mov     [bx].A4ExifDSHandlerPtr, ax
        mov     [bx].A4ExifDSStatusPtr, 0
        mov     [bx].A4ExifDSIo.ChanNext, bx
        mov     [bx].A4ExifDSIo.ChanSignature, IoChanSignature
        mov     [bx].A4ExifDSIo.ChanLibHandle, dx
        mov     cx, di                          ;IoRequestReset takes
        xchg    bx, dx                          ;the device handle in
        IoRequestReset                          ;BX and the channel
        xchg    bx, dx                          ;handle in CX
        xor     ax, ax
        call    StartTheChannelRunning
ReturnCLC:
        clc
        ret
        ProcEnd


if Asic9
        ProcBegin@ A4ExifTickInt,far
;                  ============
;       This function is called by the tick handler on every tick of
;       the system clock. This happens 32 times a second.
;       The operating system will call a device driver to hold when memory is
;       being moved and when the machine is being powered down. It will not
;       call the device driver when the pack doors are opened.
;       Opening the pack doors will cause power to the peripheral to be cut,
;       and therefore the driver needs to be held in the way it would be if
;       the machine powered down.  Only for Asic9 based machines.
;       This function checks the state of the doors on every tick and calls
;       Hold and resume when it sees the status of the doors change.
;       IN:
;               The state of the door in SI
;       OUT:
;               Nothing
;
        cmp     si, DoorOpen                    ;Is the door open
        je      TheDoorIsOpen                   ;or closed?
        cmp     HoldFlag, 2                     ;Closed Door, HoldFlag=2
        je      NeedToDoTheResume               ;means we do a resume
        ret                                     ;Closed Door, HoldFlag!=2
NeedToDoTheHold:
        mov     HoldFlag, 2
        jmp     A4HoldFromTick
NeedToDoTheResume:
        mov     HoldFlag, 0
        jmp     A4ResumeFromTick
TheDoorIsOpen:
        xor     ax, ax
        cmp     HoldFlag, al
        je      NeedToDoTheHold
        ret
        ProcEnd noret
endif


        ProcBegin@ A4ExifHandler,far
;                  ============
;       When an application is waiting within an iowait and a signal is
;       generated then before passing that signal to the application the
;       OS first runs any wait handlers belonging to the device driver
;       channels that the application has open.
;       The interrupt routine can generate a signal but cannot fill in any
;       status words or pass values back to the application because the
;       applications DS space is not available. The handler can consume a
;       signal generated by an interrupt and then fill any status words
;       before re-signalling the application. A handler always has access
;       to the application's DS space.  The wait handler can consume the
;       signal which is then no longer passed back to the application.
;       IN:
```

```
;               Pointer to our control block in applications DS space in BX
;               DS,ES,SS point at application's data space
;       OUT:
;               Carry clear - do not consume the signal, signal not for us
;               Carry set - consume the signal, re-enable handler if AL
;                           non-zero else don't re-enable handler if AL=0.
;
        cld
        pushf
        cli
        mov     di, [bx].A4ExifDSStatusEntPtr           ;Is the signal for us?
        cmp     cs:[di].A4ExifCSChanReadCompleted, 1
        jne     ExitHandlerSignalNotForUs               ;If it is, copy
        mov     cs:[di].A4ExifCSChanReadCompleted, 0    ;the values read in
        mov     al, cs:[di].A4ExifCSA1Value             ;the interrupt
        mov     ah, cs:[di].A4ExifCSA2Value             ;routine back to
        mov     di, [bx].A4ExifDSA1Ptr                  ;the application
        mov     [di], al                                ;Asynchronous read
        mov     di, [bx].A4ExifDSA2Ptr                  ;has been completed
        mov     [di], ah
        xor     di, di
        xchg    [bx].A4ExifDSStatusPtr, di              ;Clear the status
        mov     word ptr [di], 0                        ;word
        popf
        IoSignal
        xor     ax, ax
        stc                                             ;STC and AL!=0 =>
        ret                                             ;consume signal and
ExitHandlerSignalNotForUs:                              ;don't re-enable
        popf                                            ;handler
        clc
        ret
        ProcEnd


StrategyVectorTable     label   word
        dw      offset A4ExifDefault    ;StrategyPanic
        dw      offset A4ExifRead       ;StrategyRead           ie P_FREAD
        dw      offset A4ExifWrite      ;StrategyWrite          ie P_FWRITE
        dw      offset A4ExifClose      ;StrategyClose          ie P_FCLOSE
        dw      offset A4ExifCancel     ;StrategyCancel         ie P_FCANCEL
        dw      offset A4ExifDefault    ;StrategyAttach
        dw      offset A4ExifDefault    ;StrategyDetach
        dw      offset A4ExifSet        ;StrategySet            ie P_FSET
        dw      offset A4ExifSense      ;StrategySense          ie P_FSENSE


        ProcBegin@ A4ExifStrategy,far
;                  ==============
;       Called by the operating system when an I/O request is made on
;       the device driver.
;       Calls to this funcion from an owning application will usually take
;       the form p_ioc(pcb,func,&Stat,&A1,&A2);
;       The strategy function is called with a function number specifying
;       the action which the driver is to take, a status word to fill when
;       the action is complete, and two arguments A1 and A2.
;       All strategy functions must complete with a signal to the application.
;       Functions can be asynchronous and need not complete immediately.
;       IN:
;               Pointer to our control block in the applications DS in BX
;               Device driver handle in DX
;               Pointer to the RqEnt struct in SI
;               Pointer to a IntEnt struct in BP
;               DS,ES,SS point at the applications data space
;       OUT:
;               Returned value in AX
;               Must call IoSignal somewhere to signal completion
;                       of the I/O request.
;
        mov     ax, [si].RqFunction             ;Get the function number
        mov     dx, [si].RqA1Ptr                ;DX holds the first argument
        mov     di, [si].RqStatusPtr            ;for convenience
        mov     word ptr [di], PendingErr       ;Status word holds
        shl     ax, 1                           ;E_FILE_PENDING
        mov     di, ax
        push    StrategyVectorTable[di]         ;Jump to required function
        mov     di, [bx].A4ExifDSStatusEntPtr   ;with our CS control block
        retn                                    ;pointer in DI
A4ExifDefault:
        IoRoot
        ret
ExitWithCompletionStatusZero:
        xor     ax, ax                          ;Common exit points
ExitWithOtherCompletionStatus:
        mov     di, [si].RqStatusPtr
        mov     word ptr [di], ax
        IoSignal
ExitStillPending:
        xor     ax, ax
        clc
        ret
        ProcEnd noret


;       STRATEGY VECTOR TABLE FUNCTIONS
;       ===============================
;IN:
;       DX holds the pointer to the first argument in 'p_iow(...)' call
;       BX holds the address of the open channel control block
;       CS:DI holds the address of status struc identifying the open channel
;

        ProcBegin@ A4ExifRead,far
;                  ==========
;       Strategy vector table function that handles asynchronous byte reads
;       from the LEDs and switches.
;       The corresponding PLIB call is p_ioc(pcb,P_FREAD,&Stat,&Arg1,&Arg2);
;       The request is completed when the interrupt routine signals the
;       handler which in turn signals the application passing back the values
;       read at the time of the interrupt through the A1Ptr and A2Ptr.
;       IN:
;               Pointer to control block in applications data space in BX
;               Pointer to control block in our CS space in DI
;               A1 (pointer to Arg1) in DX
;       OUT:
;               Jumps to common exit point
;               Panics if multiple requests
;
        cmp     [bx].A4ExifDSStatusPtr, 0       ;Panic if we already
        jne     PanicPending                    ;have an I/O read request
        pushf                                   ;pending on the channel
        cli                                     ;Disable interrupts
        mov     cs:[di].A4ExifCSChanReadCompleted, 2
        mov     ax, [si].RqA1Ptr
        mov     [bx].A4ExifDSA1Ptr, ax          ;Store the locations to
        mov     ax, [si].RqA2Ptr                ;put the data when we get it
        mov     [bx].A4ExifDSA2Ptr, ax
        mov     di, [si].RqStatusPtr
        mov     [bx].A4ExifDSStatusPtr, di      ;DI holds the address
        mov     word ptr [di], PendingErr       ;of status word and we
        popf                                    ;signal that we are waiting
        mov     bx, [bx].A4ExifDSHandlerPtr     ;for completion of read
        mov     cl, 1
        IoEnableHandler                         ;Enable Wait Handler
        jmp     short ExitStillPending          ;No IoSignal because
PanicPending:                                   ;we are still waiting
        mov     al, PanicIoPending
        ProcPanic
        ProcEnd noret


        ProcBegin@ A4ExifWrite,far
;                  ===========
;       This function sets the state of the LEDs.
;       It completes immediately after setting the state as it has nothing
;       to wait for.
;       Write is the same as set.
;       IN:
;               Pointer to control block in applications data space in BX
;               Pointer to control block in our CS space in DI
;               A1 in DX
;       OUT:
;               Jumps to Set
;
        jmp     WriteAndSetAreTheSame
        ProcEnd noret


        ProcBegin@ A4ExifClose,far
;                  ===========
;       Handles the request to close a channel.
;       called by the PLIB call p_iow(pcb,P_FCLOSE) or p_close(pcb)
;       IN:
;               Pointer to control block in applications data space in BX
;               Pointer to control block in our CS space in DI
;       OUT:
;               Jumps to common exit point
;
        call    StopTheChannelRunning                   ;Stop interrupts
if Asic9
        push    bx                                      ;Close down the
        mov     bx, cs:[di].A4ExifCSTickHandle          ;"TCK:" channel
        IoClose
        pop     bx
endif
        push    bx
        mov     ax, [bx].A4ExifDSIo.ChanLibHandle       ;Remove the wait
        push    ax                                      ;handler
        mov     bx, [bx].A4ExifDSHandlerPtr
        IoRemoveHandler
        cmp     cs:[di].A4ExifCSChanReadCompleted, 0    ;If the interrupt
        je      NoSignalToConsumeFromInterrupt          ;has signalled the
        IoWaitForSignal                                 ;handler we need to
NoSignalToConsumeFromInterrupt:                         ;consume its signal
        pop     bx                                      ;now the handler
        mov     cx, di                                  ;has been removed
        IoRequestResetCancel
        pop     bx                                      ;No longer need reset
        HeapFreeCell                                    ;Free our control
        mov     al, cs:[di].A4ExifCSChannelIntMask      ;block in app's DS
        HwFreeChannel
        mov     cs:[di].A4ExifCSChannelOpen, 0
        jmp     ExitWithCompletionStatusZero            ;To signal completion
        ProcEnd noret                                   ;of close request


        ProcBegin@ A4ExifCancel,far
;                  ============
;       Cancel any pending asynchronous read.
;       Called by the PLIB function p_iow(pcb,P_FCANCEL);
;       In:
;               Pointer to control block in applications data space in BX
;               Pointer to control block in our CS space in DI
;       Out:
;               Jumps to common exit point
;
        pushf
        cli
        cmp     [bx].A4ExifDSStatusPtr, 0               ;Check that there is
        je      NothingToCancel                         ;a request pending
        push    bx
        mov     bx, [bx].A4ExifDSHandlerPtr
        sub     cl, cl
        IoEnableHandler                                 ;Disable Wait Handler
        pop     bx
        cmp     cs:[di].A4ExifCSChanReadCompleted, 1
        jne     NoSignalFromInterrupt                   ;To consume any stray
        IoWaitForSignal                                 ;signal from the
NoSignalFromInterrupt:                                  ;interrupt routine
        mov     cs:[di].A4ExifCSChanReadCompleted, 0
        xor     di, di                                  ;To signal completion
        xchg    di, [bx].A4ExifDSStatusPtr              ;of the outstanding
        mov     word ptr [di], CancelErr                ;async read request
        IoSignal                                        ;Signal to p_waitstat
NothingToCancel:
        popf
        jmp     ExitWithCompletionStatusZero            ;To signal completion
        ProcEnd noret                                   ;of the cancel request


        ProcBegin@ A4ExifSet,far
;                  =========
;       Write a value to the LED latch.
;       Can be called by the PLIB call p_iow(pcb,P_FSET,&Arg1); where A1
;       is an unsigned byte.
;       Write is the same as set.
;       IN:
;               Pointer to control block in applications data space in BX
;               Pointer to control block in our CS space in DI
;               A1 (pointer to Arg1) in DX
;       OUT:
;               Jumps to common exit point
;
WriteAndSetAreTheSame:
        pushf
        cli
        mov     al, cs:[di].A4ExifCSChannelSelect       ;Select our SIBO
        HwSelectChannel                                 ;serial channel
        push    ax                                      ;Store old channel
        mov     bx, dx
        mov     dl, U5OUTPUT_LATCH                      ;DX (now BX) points
        mov     al, [bx]                                ;to the value to
        call    OutputByte                              ;output to our
        pop     ax                                      ;peripheral
        HwSelectChannel                                 ;Return old channel
        popf
        jmp     ExitWithCompletionStatusZero            ;To signal completion
        ProcEnd noret                                   ;of set request


        ProcBegin@ A4ExifSense,far
;                  ===========
;       Reads the state of the LEDS and Switches.
;       Can be called from PLIB using p_iow(pcb,P_FSENSE,&Arg1,&Arg2)
;       Where Arg1 and Arg2 are unsigned bytes.
;       In:
;               Pointer to control block in applications data space in BX
;               Pointer to control block in our CS space in DI
;               A1 (pointer to Arg1) in DX
;       Out:
;               Jumps to common exit point
;
        pushf
        cli                                             ;Select our SIBO
        mov     al, cs:[di].A4ExifCSChannelSelect       ;serial channel
        HwSelectChannel                                 ;Store old channel
        push    ax
        mov     bx, dx                                  ;DX (now BX) points
        mov     dl, U4STATUS_BUFFER                     ;to the variable in
        call    InputByte                               ;which to place the
        mov     [bx], al                                ;value read from U4.
        mov     bx, [si].RqA2Ptr                        ;BX now points to
        mov     dl, U3INPUT_BUFFER                      ;the variable in
        call    InputByte                               ;which to place the
        mov     [bx], al                                ;value read from U3.
        pop     ax
        HwSelectChannel                                 ;Return old channel
        popf
        jmp     ExitWithCompletionStatusZero            ;To signal completion
        ProcEnd noret                                   ;of a sense request


;       LOCAL DEVICE DRIVER FUNCTIONS
;       =============================

        ProcBegin@ InputByte
;                  =========
;       Interrupts must be off prior to the call to this function.
;       IN:
;               DL has address to which Asic4 is to write
;               AL holds the value to output
;
        mov     al, (SerialWriteSingle or A4Address)
        SBUSY
        SCONTOUT
        mov     al, dl
        SBUSY
        SDATAOUT
        mov     al, (SerialReadSingle or A4Data)
        SBUSY
        SCONTOUT
        SBUSY
        SDATAIN
        ret
        ProcEnd noret


        ProcBegin@ OutputByte
;                  ==========
;       Interrupts must be off prior to the call to this function.
;       IN:
;               DL has address to which Asic4 is to write
;               AL has the value to output
;
        push    ax
        mov     al, (SerialWriteSingle or A4Address)
        SBUSY
        SCONTOUT
        mov     al, dl
        SBUSY
        SDATAOUT
        mov     al, (SerialWriteSingle or A4Data)
        SBUSY
        SCONTOUT
        SBUSY
        XNOP
        pop     ax
        SDATAOUT
        ret
        ProcEnd noret


        ProcBegin@ IntVec0,far
;                  ===========
;       Interrupt on serial channel0
;       Calls Comint with channel control block pointer in DI
;
        mov     di, offset Channel0
        jmp     ComInt
        ProcEnd noret


if Corporate or S3c
        ProcBegin@ IntVec1,far
;                  ===========
;       Interrupt on serial channel1
;       Calls Comint with channel control block pointer in DI
;
        mov     di, offset Channel1
        jmp     ComInt
        ProcEnd noret


        ProcBegin@ IntVec2,far
;                  ===========
;       Interrupt on serial channel2
;       Calls Comint with channel control block pointer in DI
;
        mov     di, offset Channel2
;       FALL THROUGH
        ProcEnd noret
endif


        ProcBegin@ ComInt,far
;                  ==========
;       The common interrupt service routine code.
;       When an interrupt occurs, the first task is to read the status
;       buffer of latch U4.  The subsequent action is dependent on the
;       postion of switches S1 and S2.  For the purposes of this example,
;       the four posibilities for the switch values correspond somewhat
;       arbitrarily to four different byte values that are written to U5.
;       in the 8086, interrupts cannot occur while in an interrupt routine.
;       If an asynchronous read is pending then the handler is signalled.
;       IN:
;               Channel's CS based control block pointer in DI
;
        cmp     cs:[di].A4ExifCSChanReadCompleted, 2    ;Only if 2 do we have
        jne     NotAsynchronousRead                     ;asynch read completed
        mov     cs:[di].A4ExifCSChanReadCompleted, 1
        mov     bx, cs:[di].A4ExifCSProcessId           ;Signals completion
        IoSignalByPidNoReSched                          ;of read to OS so as
NotAsynchronousRead:                                    ;to invoke handler
        mov     al, cs:[di].A4ExifCSChannelSelect
        HwSelectChannel
        push    ax
        mov     dl, U3INPUT_BUFFER
        call    InputByte
        mov     cs:[di].A4ExifCSA2Value, al             ;LED byte
        mov     dl, U4STATUS_BUFFER
        call    InputByte
        mov     cs:[di].A4ExifCSA1Value, al             ;Status byte
        and     al, INTERRUPT_STATUS_MASK
        xchg    al, SwitchStatus
        cmp     SwitchStatus, S2S3_ON                   ;S2 and S3 on => turn
        je      AllLEDsOn                               ;on alternate LEDs
        cmp     SwitchStatus, S2S3_OFF                  ;S2 and S3 off => not
        je      AllLEDsOff                              ;the alternate LEDs
        cmp     SwitchStatus, S2_ONLY                   ;S2 on, S3 off => turn
        je      TopLEDsOn                               ;on top four LEDs
        cmp     SwitchStatus, S3_ONLY                   ;S3 on, S2 off => turn
        je      BottomLEDsOn                            ;on bottom four LEDs
ErrorInSwitchStatusByte:
        jmp     ClearInterruptAndReschedule
AllLEDsOn:
        mov     al, SOME_LEDS_ON
        jmp     OutputLEDByte
AllLEDsOff:
        mov     al, SOME_LEDS_OFF
        jmp     OutputLEDByte
TopLEDsOn:
        mov     al, TOP_LEDS_ON
        jmp     OutputLEDByte
BottomLEDsOn:
        mov     al, BOTTOM_LEDS_ON
OutputLEDByte:
        mov     dl, U5OUTPUT_LATCH
        call    OutputByte
ClearInterruptAndReschedule:
        mov     dl, INTERRUPT_LATCH
        call    OutputByte
if      Asic9                                  ;A write to this location
        out     A9BNonSpecificEoiW, al          ;informs the interrupt
else                                            ;controller that the
        out     A1NonSpecificEoi, al            ;installed interrupt
endif                                           ;service routine has
        pop     ax                              ;finished
        HwSelectChannel
        clc                                     ;Reschedule if necessary
        ret
        ProcEnd noret


        ProcBegin@ StartInterrupts
;                  ===============
;       Start interrupts assuming interrupts are
;       off prior to call.
;       The EPOC GenSetRevector service loads in
;       a user-specified interrupt service routine
;       located at the address given in cx:bx
;       (segment cx, offset bx) for the interrupt
;       vector number given in AL.  Note that the
;       variable A4ExifCSChannelIntVec holds the
;       name of the appropriate required interrupt
;       vector routine for the channel.
;       IN:
;               Our CS control block pointer in DI
;       OUT:
;               Nothing
;
        mov     al, cs:[di].A4ExifCSChannelIntNum       ;Get the OS to call
        mov     cx, cs                                  ;the function in
        push    bx                                      ;CS:BX every time
        mov     bx, cs:[di].A4ExifCSChannelIntVec       ;that the interrupt
        GenSetRevector                                  ;whose number is in
        pop     bx                                      ;AL occurs
ife Asic9
        in      al, A1InterruptMask
        or      al, cs:[di].A4ExifCSChannelIntMask      ;Set the mask
        out     A1InterruptMask, al                     ;location so as
;interrupt
else                                                    ;to enable that
        in      al, A9BInterruptMaskRW                  ;interrupt
        or      al, cs:[di].A4ExifCSChannelIntMask
        out     A9BInterruptMaskRW, al

endif
        ret
        ProcEnd noret
        ProcBegin@ StopInterrupts
;                  ==============
;       Stop interrupts assuming that interrupts are off
;       The EPOC GenResetRevector OS service
;       replaces the previously loaded user-specified
;       interrupt service routine with the original
;       routine and interrupt mask for the vector
;       given in AL.
;       IN:
;               Our CS control block pointer in DI
;       OUT:
;               Nothing
;
ife Asic9
        in      al, A1InterruptMask                     ;Disable the
        mov     ah, cs:[di].A4ExifCSChannelIntMask      ;interrupt
        not     ah
        and     al, ah
        out     A1InterruptMask, al
else
        in      al, A9BInterruptMaskRW
        mov     ah, cs:[di].A4ExifCSChannelIntMask
        not     ah
        and     al, ah
        out     A9BInterruptMaskRW, al
endif
        mov     al, cs:[di].A4ExifCSChannelIntNum       ;Return the interrupt
        GenResetRevector                                ;vector to the OS
        ret                                             ;default
        ProcEnd noret


        ProcBegin@ StartTheChannelRunning
;                  ======================
;       Starts the hardware and interrupts going
;       If the hardware is aready running then there is nothing to do
;       Must check that the hardware has not vanished before restarting it
;       IN:
;               Our CS control block pointer in DI
;       OUT:
;               Nothing
;
        pushf
        cli
        cmp     byte ptr cs:[di].A4ExifCSChannelRunning, 0
        jne     ChannelAlreadyRunning
        call    CheckHardwarePresent
        jc      HardwareNotPresent
        mov     al, cs:[di].A4ExifCSChannelSelect
        HwSelectChannel
        push    ax
        mov     al, (SerialWriteSingle or A4Control)    ;Switch on the
        SBUSY                                           ;output latch U5
        SCONTOUT                                        ;by asserting
        mov     al, U5_ENABLE_ON                        ;the ASIC4 LBO line
        SBUSY                                           ;(LBO is inverted)
        SDATAOUT
        xor     ax,ax
        mov     dl, INTERRUPT_LATCH                     ;Clear any pending
        call    OutputByte                              ;interrupt on the
        mov     dl, U5OUTPUT_LATCH                      ;peripheral
        call    OutputByte                              ;Preset the LEDs
        pop     ax                                      ;to all off
        HwSelectChannel
        call    StartInterrupts
        mov     cs:[di].A4ExifCSChannelRunning, 1
ChannelAlreadyRunning:
        popf
        clc
        ret
HardwareNotPresent:
        popf
        stc
        ret
        ProcEnd noret


        ProcBegin@ StopTheChannelRunning
;                  =====================
;       Stops the hardware and interrupts
;       Checks to see if the hardware is really running to start with
;       IN:
;               Our CS control block pointer in DI
;       OUT:
;               Nothing
;
        pushf
        cli
        cmp     byte ptr cs:[di].A4ExifCSChannelRunning, 0
        je      ChannelNotRunning
        mov     al, cs:[di].A4ExifCSChannelSelect
        HwSelectChannel
        push    ax
        mov     al, (SerialWriteSingle or A4Control)
        SBUSY
        SCONTOUT
        mov     al, U5_ENABLE_OFF
        SBUSY
        SDATAOUT
        xor     ax, ax
        mov     dl, U5OUTPUT_LATCH
        call    OutputByte
        pop     ax
        HwSelectChannel
        call    StopInterrupts
        mov     cs:[di].A4ExifCSChannelRunning, 0
ChannelNotRunning:
        popf
        ret
        ProcEnd noret




        ProcBegin@ CheckHardwarePresent
;                  ===================
;       Used to determine whether the correct hardware is present
;       on the successfully procured serial channel.
;       IN:
;               CS control block pointer in DI
;       OUT:
;               Carry clear - correct hardware is there
;               Carry set - wrong or no hardware
;
       pushf
        cli                                             ;Select the correct
        mov     al, cs:[di].A4ExifCSChannelSelect       ;SIBO channel that
       HwSelectChannel                                      ;the peripheral is
        push    ax                                      ;attached to
       HwNullFrame
       mov     al,(SerialSelect or Asic4Id)
       SBUSY                                       ;First look for an
       SCONTOUT                                    ;ASIC4 at the other
       XNOP                                        ;end of the link
       SBUSY
       SDATAIN                                        ;If the returned
       test    al,al                                 ;value is non-zero
       je      ConnectionFailedA4NotPresent          ;we have an ASIC4
       mov     al,(SerialReadSingle or A4InfoR)
       SBUSY                                        ;Now see if we have
       SCONTOUT                                    ;the right peripheral
       XNOP                                           ;XNOP allows the busy
       SBUSY                                          ;signal to come
       SDATAIN                                        ;through for the wait
       and      al, A4PERIPH_MASK
        cmp     al, EXTENDED_INFO_BYTE                  ;Mask out the bottom
        jne     ConnectionFailed                        ;four bits as the
       pop      ax                                     ;upper four contain
        HwSelectChannel                                 ;the peripheral ID
        popf
       clc                                            ;If it is our hardware
       ret                                            ;exit with carry clear
ConnectionFailedA4NotPresent:
       mov     al,(SerialSelect or Asic5NormalId)    ;Its not an ASIC4
       SBUSY                                       ;peripheral
       SCONTOUT                                       ;By selecting a non
       XNOP                                           ;ASIC4 as an ASIC4,
       SDATAIN                                        ;we effectively
       test    al,al                                 ;disable whatever is
```

```
        jne     ConnectionFailed                        ;out there so we do
        mov     al,(SerialSelect or Asic5PackId)      ;a select for all
        SBUSY                                           ;possibilities so
        SCONTOUT                                        ;that we don't end
        XNOP                                            ;up disabling
        SDATAIN                                         ;anything that we
        test    al,al                                   ;can't control.
        jne     ConnectionFailed
        mov     al,(SerialSelect or Asic8Id)            ;Modem chip id
        SBUSY
        SCONTOUT
        XNOP
        SDATAIN
ConnectionFailed:
        pop     ax
        HwSelectChannel
        popf
        stc
        ret
        ProcEnd noret

        EndCodeSeg
        stack segment stack para 'data'
        stack ends
        end A4ExifLDD
```

SYS$AS5.ASM

```
        title   AS5PDD  Epoc Serial physical device driver for the 16550
        subttl  Copyright Psion PLC 1993
        name    SYS$AS5

;       VERSION   DATE   DESCRIPTION
;       ------- -------- ---------------
;         1.0   08/12/94 Initial version

;       Written by Jason December 1994

;       Serial Driver for Epoc based around ASIC5

Sr5S3   =       0
Sr5S3a  =       1

ifdef   BUILDS3
Sr5S3   =       1
Sr5S3a  =       0
BUILDHH equ     1
endif

if      Sr5S3a
BUILDSB equ     1
endif

        include ..\inc\epoc.inc
        include ..\inc\epocser.inc
        include ..\inc\epoclib.inc
        include ..\inc\epocsibo.inc
        include ..\srcs\ossibo.inc
        include ..\srcs\ospack.inc

Sr5ChannelStruct struc
        Sr5Open                 db      ?               ; Is the channel open
        Sr5Ctrl                 db      ?               ; State of control lines
        Sr5IntVector        db      ?               ; The Vector number
        Sr5Channel          db      ?               ; Which channel are we
        Sr5Mask                 db      ?               ; InterruptMask
        Sr5Running          db      ?               ; Are we running
        Sr5IntRoutineVec    dw      ?               ; Vector to Interrupt
        Sr5Baud                 dw      ?               ; The baud rate
        Sr5LddData          dw      ?               ; Info from Ldd above
        Sr5StatusInt        dd      ?               ; Vectors in serial
        Sr5RecvInt          dd      ?               ; Above to be called
        Sr5XmitInt          dd      ?               ; On input/output
        Sr5ClockEnable      db      ?               ; Reason to stop
        Sr5TheLines         db      ?               ; State of the modem lines
Sr5ChannelStruct ends

A5Ent struc                                         ; ASIC5 Read/Write
        A5PortA                 db      ?           ; Port A R/W
        A5PortB                 db      ?           ; Port B R/W
        A5PortBMode             db      ?           ; Inc/Mode
        A5PortD                 db      ?           ; Port CD Write only
        A5Swipe1                db      ?
        A5Swipe2                db      ?
        A5IntMask               db      ?           ; Interrupt mask R/W
        A5CtrlReg               db      ?           ; IntType/Ctrl register
        A5USR                 db      ?             ; UART Status/Ctrl
        A5RHR                 db      ?             ; Receive/Transmit
        A5BDLSB                 db      ?           ; Baud Rate write only
        A5BDMSB                 db      ?           ; Baud Rate write only
        A5MCR1Eoi               db      ?           ; MCR shift register
        A5MCRPresentEoi         db      ?           ; Barcode data&ints
        A5MCR2Eoi               db      ?
        A5DUMMYF                db      ?
A5Ent ends

if      Consumer
        NumberOfChannels        equ     1           ; S3 Single Channel
else
        NumberOfChannels        equ     3           ; HC,S3C Three Channel
endif
       OsActivityMeter         equ     158ch       ; Activity Channel
        StopTimeOut             equ     1000        ; ~1s (1000ms) wait

        S_RS232ON       equ     00000001b           ; RS232 on
        S_RSTTLON       equ     00000100b               ; Line drivers on
        S_CENTON        equ     00010000b               ; Line drivers enable
        S_RXENB                equ     00000001b               ; Receive interrupt on
        S_TXENB                equ     00000010b               ; Transmit interrupt on
        S_TXEMPTY       equ     00010000b               ; transmit buffer empty
        S_RXINT                equ     00000001b               ; Receive interrupt?
        S_TXINT         equ     00000010b               ; transmit interrupt?
        S_MDINT         equ     00000100b               ; Modem status interrupt
        S_CTS           equ     00000001b               ; CTS
        S_RTS           equ     00000010b               ; RTS
        S_DCD          equ     00000100b               ; DCD
        S_DSR           equ     00000010b               ; DCR
        S_DTR           equ     00000100b               ; DTR
        OVERRUN_ERROR   equ     01000000b               ; Character overrun
        PARITY_ERROR    equ     10000000b               ; Parity error
        S_PERIPHERALMODE equ    00000011b               ; ASIC5 RS232 mode
        S_UART_OFF      equ     00000010b               ; ASIC5 peripheral mode

dgroup  group   stack
        assume  ds:dgroup,es:dgroup,ss:dgroup

        CodeSeg

        ProcBegin@ OsAS5PDD
;       ====================

        dw      PDDSignature
        db      'TTY.SR5',0
        dw      (VectorEnd-Vector)/2
Vector:
        dw     OsAS5Install
        dw     OsAS5Remove
        dw      OsAS5Open
        dw      OsAS5Strategy
VectorEnd:

        BaudRateTable   dw      -077fh,-04ffh,-0368h,-02cch,-027fh,-013fh
                        dw      -009fh,-004fh,-0035h,-0030h,-0027h,-0019h
                        dw      -0013h,-000ch,-0009h,-0004h
        DataBitsTable   db      0,2,4,6                 ; 5,6,7,8 bits per char frame
        ParityTable     db      08h,18h,0h,0h           ; Even,Odd,Mark,Space parity

        Chan0           Sr5ChannelStruct        <>
        Chan1           Sr5ChannelStruct        <>
        Chan2           Sr5ChannelStruct        <>

if Consumer
  if Asic9
        SetupTable      dw      offset AS5Int1
                        db      HwIrq2Revector,mask A9MSlave
                        db      SelectChannel5,(mask A9MClockEnable5 shr 8)
  else
        SetupTable      dw      offset AS5Int1
                        db      HwIrq4Revector,mask Asic2Int
                        db      SelectChannel7,(mask ClockEnable7 shr 8)
  endif
else
  if Asic9
        SetupTable      dw      offset AS5Int1
                        db      HwIrq4Revector,mask A9MExpIntA
                        db      SelectChannel3,(mask A9MClockEnable3 shr 8)
                        dw      offset AS5Int2
                        db      HwIrq5Revector,mask A9MExpIntB
                        db      SelectChannel4,(mask A9MClockEnable4 shr 8)
                        dw      offset AS5Int3
                        db      HwIrq2Revector,mask A9MSlave
                        db      SelectChannel5,(mask A9MClockEnable5 shr 8)
  else
        SetupTable      dw      offset AS5Int1
                        db      HwIrq3Revector,mask ExpIntLeftA
                        db      ExpChannelLeftA,(mask ClockEnable6 shr 8)
                        dw      offset AS5Int2
                        db      HwIrq2Revector,mask ExpIntRightB
                        db      ExpChannelRightB,(mask ClockEnable5 shr 8)
                        dw      offset AS5Int3
                        db      HwIrq4Revector,mask Asic2Int
                        db      SelectChannel7,(mask ClockEnable7 shr 8)
  endif
endif
        ProcEnd noret
        ProcBegin@ OsAS5Install,far
;       ============================

;       Install the device driver
;       Out: Carry clear -happy to install

        cld                                         ; Clear the channels
        mov     cx, NumberOfChannels                ; and set up fixed
        mov     di, offset Chan0                    ; parameters such
        mov     si, offset SetupTable               ; as the interrupt
        pushf                                       ; vectors and masks
        cli                                         ; for each channel
        push    ds
        mov     ax, cs
        mov     ds, ax
ResetAllChannelsLoop:
        mov     [di].Sr5Open, 0                     ; Channel not open
        lodsw
        mov     [di].Sr5IntRoutineVec, ax           ; Set the interrupt
        lodsb                                       ; handler to call
        mov     [di].Sr5IntVector, al               ; Which Interrupt
        lodsb
        mov     [di].Sr5Mask, al                    ; Mask for that
        lodsb                                       ; interrupt
        mov     [di].Sr5Channel, al                 ; Which Psion serial
        lodsb                                       ; channel
        mov     [di].Sr5ClockEnable, al             ; Baud rate clocking
        add     di, size Sr5ChannelStruct           ; enable
        loop    ResetAllChannelsLoop
        pop     ds
        popf
AllChannelsOkay:
        clc                                     ; Returns with
        ret                                     ; Carry clear
        ProcEnd noret

        ProcBegin@ OsAS5Remove,far
;       ==========================

;       Remove the device driver
;       Out: Carry clear -happy to remove
;            Carry set -we have an open channel and cant be removed.

        xor     ax, ax                              ; If we have a
        or      al, Chan0.Sr5Open                   ; channel Still open
        or      al, Chan1.Sr5Open                   ; then return a can't
        or      al, Chan2.Sr5Open                   ; do error else
        jz      AllChannelsOkay                     ; complete okay
        mov     ax, InUseErr
        stc
        ret
        ProcEnd noret

        ProcBegin@ OsAS5Open,far
;       ========================

;       Open a serial channel
;       In:  SS:SI is a pointer to the open Ent Structure
;       Out: Carry clear, control block in BX
;            Carry set, error in AX

        cld
        mov     si, [si].OpenNamePtr                ; Open the channel
        mov     al, [si+1]                          ; Get the channel
        CharToFoldedChar                            ; Make Upper Case
        cmp     al, 'A'                             ; Indicator which is
        jb      ErrorInOpen                         ; Part of the name
        sub     al, 'A'                             ; Should be A,B,C
        cmp     al, NumberOfChannels
        jae     ErrorInOpen
        xor     ah, ah
        mov     bx, offset Chan0                    ; What Channel are
        cmp     al, 1                               ; We Openning
        jb      GotChan                             ; Pointer to Control
        mov     bx, offset Chan2                    ; Block in bx
        ja      GotChan
        mov     bx, offset Chan1
GotChan:
        mov     al, 1
        xchg    al, cs:[bx].Sr5Open
        cmp     al, 0                                ; Now try to Open
        je      OkayToOpen                           ; That channel
CantOpen:
        mov     ax, InUseErr
CantOpenDiffErr:
        stc
        ret
ErrorInOpen:
        mov     ax, NameErr
        stc
        ret
OkayToOpen:
        xor     al, al
        mov     cs:[bx].Sr5Running, al
        mov     cs:[bx].Sr5TheLines, al
        mov     cs:[bx].Sr5Ctrl, al
        mov     al, cs:[bx].Sr5Mask
       HwGetChannel
        jc      CantOpenSoClose                          ; Check that we have
       call    CheckHardwarePresent                 ; The right Hardware
       jnc     OpenedOkay
       mov     al, cs:[bx].Sr5Mask
       HwFreeChannel
CantOpenSoClose:
        mov     cs:[bx].Sr5Open, 0
        mov     ax, DeviceErr
        jmp     short CantOpenDiffErr               ; Return with the
OpenedOkay:                                         ; Offset of our
        xor     ax, ax                              ; Control Block
        ret                                         ; In bx
        ProcEnd noret

AS5StrategyJumpTable label word
        dw      offset AS5Open                          ; Load Handler offsets
        dw      offset AS5Close                          ; Close the channel
        dw      offset AS5Start                                 ; Start the
channel
        dw      offset AS5Stop                          ; Stop the channel
        dw      offset AS5Set                       ; Does nothing
        dw      offset AS5Sense                          ; Returns chan status
        dw      offset AS5Control                       ; Drive the lines
        dw      offset AS5Enquire                       ; Returns baud rate
        dw      offset AS5Enable                        ; Begin Output
        dw      offset AS5SetHandlerCS                  ; Get Handler segments

       ProcBegin@ OsAS5Strategy,far
;      ============================

;       Strategy functions entry point
;      Warning! This can be called from within Interrupt
;      In:  vector number in AX + various data in other registers
;           DS is OsDataGroup (and MUST be preserved)
;      Out: DI is pointer to control block
;           Old channel and flags on stack

       cld
       mov     bx, sp
       mov     bx, ss:[bx+4]
       mov     bl, cs:[bx].Sr5Channel          ; Select the correct
        xchg    bx, ax                          ; Output channel
       pushf                                   ; Then Call the right
       cli                                     ; Function to deal
        HwSelectChannel                         ; With the strategy
        mov     ah, bh                          ; request
        xor     bh, bh
        push    ax                              ; Interrupts off
        mov     ax, di                          ; and flags are on
       mov     di, sp                          ; the stack
       mov     di, ss:[di+8]
       jmp     AS5StrategyJumpTable[bx]
       ProcEnd noret

       ProcBegin@ AS5Open,far
;      =====================

;       Old channel and flags on the stack, interrupts off

        mov     cs:[di].Sr5LddData, cx                 ; Load the offsets of
        mov     word ptr cs:[di].Sr5StatusInt, ax      ; the data send and
        mov     word ptr cs:[di].Sr5RecvInt, si            ; receive routines in
        mov     word ptr cs:[di].Sr5XmitInt, dx         ; the Ldd above us
        xor     dx, dx                                  ; DX=0 -We don't support
        pop     ax                                      ; power management
        HwSelectChannel                                 ; Return the old
        popf                                            ; channel
        ret
        ProcEnd noret

        ProcBegin@ AS5Close,far
;       ======================

;       Old channel and flags on the stack, interrupts off

        and     cs:[di].Sr5Open, 0                      ; Close the Channel
        mov     al, cs:[di].Sr5Mask                     ; Free up our channel
        HwFreeChannel                                   ; And the Hardware
        pop     ax                                      ; Channel
        HwSelectChannel                                 ; Return the old
        popf                                            ; channel
        ret
        ProcEnd noret

        ProcBegin@ AS5Start,far
;       ======================

;       Old channel and flags on the stack, interrupts off

        call    CheckHardwareFromStart
        mov     dx, 0
        jc      CantStartSomethingWhichIsntThere

if      Asic9
        in      ax, A9WControlExtraRW                   ; Start the S3s clock
        or      ah, cs:[di].Sr5ClockEnable              ; Generator
        out     A9WControlExtraRW, ax
else
        mov     al, cs:[di].Sr5ClockEnable
        HwSetA2Control2Bits
endif

        mov     al, SerialWriteSingle or A5USR          ; Set the baud rate
        SBUSY                                           ; and other
        SCONTOUT                                        ; characteristics
        mov     al, cs:[di].Sr5Ctrl
        SBUSY
        SDATAOUT
        mov     al, SerialWriteSingle or A5BDLSB
        SBUSY
        SCONTOUT
        mov     al, byte ptr cs:[di].Sr5Baud
        SBUSY
        SDATAOUT
        mov     al, SerialWriteSingle or A5BDMSB
        SBUSY
        SCONTOUT
        mov     al, byte ptr cs:[di+1].Sr5Baud
        SBUSY
        SDATAOUT

        call    GetTheInterrupt                         ; Get the interrupt
        xor     cx, cx                                  ; and clear down
        call    DriveRts                                ; the RTS line (DTR
                                                        ; stays at the state
        mov     al, SerialWriteSingle or A5CtrlReg      ; it was set)
        SBUSY
        SCONTOUT
        mov     al, (S_CENTON or S_RS232ON or  S_RSTTLON)
        SBUSY
        SDATAOUT                                        ; Switch on the line
        mov     cx, 8 ; 8 ms                            ; drivers and wait
        pop     dx                                      ; for them to power up
        call    WaitTimer
        push    dx
        mov     ah, (S_RXENB or S_TXENB or S_MDINT)     ; Start all interrupts
        call    EnableTheInterrupt
        mov     cs:[di].Sr5Running, 1
        mov     bx, Asic5SerialCurrent
        HwSetPCurrent
        call    Status
```

```
CantStartSomethingWhichIsntThere:
        pop     ax                                  ; Return the old
        HwSelectChannel                         ; channel
        popf
        ret
        ProcEnd noret

        ProcBegin@ AS5Stop,far
;       ======================

;       Old channel and flags on the stack, interrupts off


        xor     cx, cx                              ; Clear the state of
        call   DriveRts                           ; the modem lines
        mov     al, SerialReadSingle or A5IntMask   ; Stop all of the
        SBUSY                                       ; interrupts
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        and     al, not (S_TXENB or S_RXENB or S_MDINT)
        mov     ah, al
        mov     al, SerialWriteSingle or A5IntMask
        SBUSY
        SCONTOUT
        mov     al, ah
        SBUSY
        SDATAOUT
        pop     dx
        cmp     dh, DevHoldPowerFail
        je      DontStopHardware
        cmp    cs:[di].Sr5Running, 1
        jne     DontWait
        mov     cx, StopTimeOut
        jmp     short CompareNow
WaitForEmpty:
        call    TickTimer
CompareNow:
        mov     al, SerialReadSingle or A5USR
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        test    al, S_TXEMPTY
        loopne WaitForEmpty
DontWait:
        cmp    bh,DevHoldNormal
        je     DontStopHardware

if      Asic9
        mov     cl, cs:[di].Sr5ClockEnable          ; Turn off baud rate
        not     cl                                  ; clocking from the
        in      ax, A9WControlExtraRW               ; S3/3a
        and     ah, cl
        out     A9WControlExtraRW, ax
else
        mov     al, cs:[di].Sr5ClockEnable
        HwClearA2Control2Bits
endif
        mov     al, SerialWriteSingle or A5CtrlReg  ; Stop the drivers
        SBUSY                                       ; and stuff
        SCONTOUT
        sub     al, al                              ; clear S_CENTON,
        SBUSY                                       ; S_RS232ON, S_RSTTLON
        SDATAOUT
DontStopHardware:
        mov     al, dl                              ; Return the old
        HwSelectChannel                             ; channel
        popf
        xor     bx, bx
        HwSetPCurrent
        ret
        ProcEnd noret

        ProcBegin@ AS5Set,far
;       ====================

;       Old channel and flags on the stack, interrupts off
;       In:  Information required is on the stack

        pop     ax                                      ; Return the old
        HwSelectChannel                                 ; channel
        popf
        mov    ah, ss:[si].SerialCharTbaud             ; Set the Recieve,
        cmp    ah, ss:[si].SerialCharRbaud             ; Transmit characteristics
        jnz     ErrorInSet                              ; First set the Baud
        cmp     ah, P_BAUD_50                           ; Rate
        jb      ErrorInSet
        cmp     ah, P_BAUD_19200
        ja      ErrorInSet
GotTheSpecialBaud:
        dec     ah
       mov     cl, ss:[si].SerialCharFrame              ; Then the number of
        mov     al, ss:[si].SerialCharParity            ; Stop bits
        dec     al
        xor     ch, ch
        push    bx
        test    cl, P_TWOSTOP
        jz      OnlyOneStopBit
        or      ch, 020h
OnlyOneStopBit:
        test    cl, P_PARITY                            ; The Parity
       jz      NoParity
        mov     bx, offset ParityTable
        xlat    cs:[ParityTable]
        or      ch, al
NoParity:
        mov     bx, offset DataBitsTable                ; And finally the
        and     cl, P_DATA_FRM                          ; Number of data bits
        mov     al, cl                                  ; Per frame
        xlat    cs:[DataBitsTable]
        or      ch, al
        mov     bx, offset BaudRateTable
        mov     al, ah
        xor     ah, ah
        shl     ax, 1
        add     bx, ax
        mov     ax, cs:[bx]
        pop     bx
        mov     cs:[di].Sr5Ctrl, ch
        mov     cs:[di].Sr5Baud, ax
        xor     al, al
        ret
ErrorInSet:
        mov     al, NotSupportedErr
        stc
        ret
        ProcEnd noret



        ProcBegin@ AS5Sense,far
;       =======================

;       Old channel and flags on the stack, interrupts off
;       Out: The state of the DCD,CTS,DSR lines returned in DX

        call    Status
       pop     ax                                       ; Return the old
        HwSelectChannel                                 ; channel
        popf
        ret
        ProcEnd noret

        ProcBegin@ Status
;       =================

;       Out: State of modem lines in DX

        mov     al, SerialReadSingle or A5USR           ; Get the current
        SBUSY                                           ; Modem status
        SCONTOUT                                        ; Lines and return
        XNOP                                            ; With the result
        SBUSY                                           ; In dx
        SDATAIN
        and    al, (S_CTS or S_DSR or S_DCD)           ; Get the lines we want
        xor    al, (S_CTS or S_DSR or S_DCD)           ; Invert signals
        xor     ah, ah
        mov     dx, ax
        ret
        ProcEnd noret

        ProcBegin@ AS5Control,far
;       =========================

;       Old channel and flags on the stack, interrupts off
;       In:  Lines to drive and state to drive them in DX

        mov     cl, dl                                  ; Set the state of one
        test    dh, P_SRCTRL_DTR                        ; of the modem Lines
        jz      DriveRtsNow                             ; DH is the line to
        call    DriveDtr                                ; Drive and DL is the
        pop     ax                                  ; State to drive it to
        HwSelectChannel                                 ; Return the old
        popf                                            ; channel
        ret
DriveRtsNow:
        call    DriveRts
        pop     ax                                      ; Return the old
        HwSelectChannel                                 ; channel
        popf
        ret
        ProcEnd noret

        ProcBegin@ DriveDtr
;       ===================

        mov     ah, S_DTR                               ; Set/Reset the DTR
        jmp     short DriveTheLine                      ; Line
        ProcEnd noret

        ProcBegin@ DriveRts
;       ===================

        mov     ah, S_RTS                               ; Set/Reset RTS
DriveTheLine:
        mov     al, SerialWriteSingle or A5PortD        ; Common code to set
        SBUSY                                           ; and reset either
        SCONTOUT                                        ; line while
        mov     al, cs:[di].Sr5TheLines                 ; preserving the
        test    cl, cl                                  ; states of the other
        jz      ClearLine                               ; lines
       or      al, ah
        jmp     DoTheOutput
ClearLine:
        not     ah
        and     al, ah
DoTheOutput:
        mov     cs:[di].Sr5TheLines, al
        SBUSY
        SDATAOUT
        ret
        ProcEnd noret

        ProcBegin@ AS5Enquire,far
;       =========================

;       Old channel and flags on the stack, interrupts off
;       Out: DX,AX are the supported baud rates AX for 50 to 19200 DX for above
;            CX says what data bits, parity, etc we support

       pop     ax                                      ; Return the old
       HwSelectChannel                                 ; channel
       popf
if      Asic9
       mov     ax, -1
else
        mov     ax, 07fffh
endif
        xor     dx, dx
        mov     cx, (0ffffh AND (NOT (P_SRINQ_SPLIT OR P_SRINQ_PARSPACE OR
P_SRINQ_PARMARK)))
        ret
        ProcEnd noret

        ProcBegin@ AS5Enable,far
;       ========================
;       Old channel and flags on the stack, interrupts off

        mov     ah, S_TXENB                             ; Begin Output by
        call    EnableTheInterrupt                      ; enabling transmit
        pop     ax                                      ; interrupts
        HwSelectChannel                                 ; Return the old
        popf                                            ; channel
        ret
        ProcEnd noret

        ProcBegin@ AS5SetHandlerCS,far
;       ==============================

;       Old channel and flags on the stack, interrupts off
;       CX is CS of above LDD

        mov     word ptr cs:[di].(Sr5StatusInt+2), cx   ; Load the Segment
        mov     word ptr cs:[di].(Sr5RecvInt+2), cx     ; Within which the
        mov     word ptr cs:[di].(Sr5XmitInt+2), cx     ; The Ldd above us
        pop     ax                                      ; Return the old
        HwSelectChannel                                 ; channel
        popf                                            ; Resides
        ret
        ProcEnd noret

        ProcBegin@ AS5Int3,far
;       ======================

        mov     di, offset Chan2                        ; Channel 3 interrupt
        mov     ax, PortCActive                         ; vector
        jmp     ComInt                                  ; Jumps to Comint
        ProcEnd noret

        ProcBegin@ AS5Int2,far
;       ======================

        mov     di, offset Chan1                        ; Channel 2 interrupt
        mov     ax, PortBActive                         ; vector
        jmp     ComInt                                  ; Jumps to Comint
        ProcEnd noret

        ProcBegin@ AS5Int1,far
;       ======================

        mov    di,offset Chan0                         ; Channel 1 interrupt
        mov     ax,PortAActive                          ; vector
;       FALL THROUGH                                    ; Falls through
        ProcEnd noret                                   ; to Comint


        ProcBegin@ ComInt,far
;       =====================

;       The common interrupt handler
;       DS points to OS data space
;       DI is our control block
;       AX is the Active channel

if      S3b or S3c
        or      ds:[OsActivityMeter], ax                ; Set active state
endif
        mov     al, cs:[di].Sr5Channel                  ; Select our
        HwSelectChannel                                 ; channel
        push    ax
        mov     bx, cs:[di].Sr5LddData                  ; Data for LDD above
TheInterruptLoop:                                       ; in bx
        mov     al, SerialReadSingle or A5CtrlReg       ; Find out what
        SBUSY                                           ; caused the
        SCONTOUT                                        ; interrupt
        XNOP
        SBUSY
        SDATAIN
        test   al, al
        je     NothingToDo
HaveWeGotAModemStatusLineInterrupt:
        test    al, S_MDINT
        jz      HaveWeGotARecieveInterrupt
        mov     al, SerialReadSingle or A5USR
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        test    al, (OVERRUN_ERROR or PARITY_ERROR)
        jz      ModemStatusInterruptOnly
        push    ax                                      ; Something has been
        mov     ah, SERPARITY_ERR                       ; recieved in error
        test    al, PARITY_ERROR
        jnz     IsAParityError                          ; Establish the error
        mov     ah, SEROVERRUN_ERR
IsAParityError:
        mov     al, SerialReadSingle or A5RHR           ; Get the character
        SBUSY                                           ; in question
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        push    di
        call    dword ptr cs:[di].Sr5RecvInt
        pop     di
        pop     ax
ModemStatusInterruptOnly:
        and     ax, (S_CTS or S_DSR or S_DCD)
        xor     al, (S_CTS or S_DSR or S_DCD)   ; invert signals
        mov     dx, ax
        push    di
        call    dword ptr cs:[di].Sr5StatusInt
        pop     di
        jmp     short TheInterruptLoop
HaveWeGotARecieveInterrupt:
        test    al, S_RXINT
        jz      HaveWeGotATransmitInterrupt
        mov     al, SerialReadSingle or A5RHR
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        xor     ah, ah                          ; AX has character received
        push    di
        call    dword ptr cs:[di].Sr5RecvInt
        pop     di
        jmp     short TheInterruptLoop
HaveWeGotATransmitInterrupt:
        test    al, S_TXINT
        jz      NothingToDo
        push    di
        call    dword ptr cs:[di].Sr5XmitInt
        pop     di
        test    ax, ax          ; -1 if disable
        js      DisableTransmitInts
        mov     ah, al
        mov     al, SerialWriteSingle or A5RHR
        SBUSY
        SCONTOUT
        mov     al, ah
        SBUSY
        SDATAOUT
        jmp     TheInterruptLoop
NothingToDo:
if      Asic9
        out     A9BNonSpecificEoiW,al
else
        out     A1NonSpecificEoi, al
endif
        pop     ax
        HwSelectChannel
        clc                                ; Resced if neccessary
        ret
DisableTransmitInts:                       ; disable TX interrupts
        mov     al, SerialReadSingle or A5IntMask
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        and     al, not S_TXENB
        mov     ah, al
        mov     al, SerialWriteSingle or A5IntMask
        SBUSY
        SCONTOUT
        mov     al, ah
        SBUSY
        SDATAOUT
        jmp     TheInterruptLoop
        ProcEnd noret

        ProcBegin@ GetTheInterrupt
;       =========================

        mov     al, cs:[di].Sr5IntVector                ; Load the address of
        mov     cx, cs                                  ; the appropriate
        mov     bx, cs:[di].Sr5IntRoutineVec            ; interrupt routine
        GenSetRevector                                  ; into the correct
        ret                                             ; vector
        ProcEnd noret

        ProcBegin@ EnableTheInterrupt
;       ============================

        mov      al, SerialReadSingle or A5IntMask
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        or      ah, al
        mov     al, SerialWriteSingle or A5IntMask
        SBUSY
        SCONTOUT
        mov     al, ah
        SBUSY
        SDATAOUT
if      Asic9
        in      al,A9BInterruptMaskRW               ; Set the mask
        or      al, cs:[di].Sr5Mask                 ; to enable Interrupts
        out     A9BInterruptMaskRW,al

else
        in      al, A1InterruptMask
        or     al, cs:[di].Sr5Mask
        out     A1InterruptMask, al
endif
        ret
        ProcEnd noret


        ProcBegin@ StopInterrupts
;       =========================

        mov     al, SerialReadSingle or A5IntMask       ; Stop interrupts
        SBUSY                                           ; from Asic5
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
        and     al, not (S_RXENB or S_TXENB or S_MDINT)
        mov     ah, al
        mov     al, SerialWriteSingle or A5IntMask
        SBUSY
        SCONTOUT
        mov     al, ah
        SBUSY
        SDATAOUT

        mov     ah, cs:[di].Sr5Mask
        not     ah
if      Asic9
        in      al,A9BInterruptMaskRW               ; Stop Interrupts
        and     al,ah                               ; By clearing the
        out     A9BInterruptMaskRW,al                   ; Mask and resetting
else                                                    ; The Vector
        in      al, A1InterruptMask
        and     al,ah
        out     A1InterruptMask, al
endif

        mov     al, cs:[di].Sr5IntVector
        GenResetRevector
        ret
```

```
         ProcEnd noret

        ProcBegin@ CheckHardwarePresent
;       ===============================

        pushf
        cli
        mov     al, cs:[bx].Sr5Channel                  ; Check that the
        HwSelectChannel                                      ; Harware is there
        HwNullFrame                                  ; And that it is what
        mov     al,(SerialSelect or Asic5NormalId)   ; It should be
        SBUSY                                        ; First look for An
        SCONTOUT                                     ; ASIC4 at the other
        XNOP                                         ; End of the link
        SBUSY
        SDATAIN
        test    al,al
        je      ConnectionFailed
GotConnection:
        popf
        clc
        ret
ConnectionFailed:
        mov     al, SerialSelect or Asic4Id             ; Asic4 Id
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
         test    al, al
        jne     ConnectionFailedExit
        mov     al, SerialSelect or Asic8Id             ; Modem chip Id
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
         test    al, al
        jne     ConnectionFailedExit
        mov     al, SerialSelect or Asic5PackId         ; Asic5pack Id
        SBUSY
        SCONTOUT
        XNOP
        SBUSY
        SDATAIN
ConnectionFailedExit:
        popf
        stc
        ret
        ProcEnd noret

        ProcBegin@ CheckHardwareFromStart
;       =================================

        pushf
        HwNullFrame                                  ; Check that the
        mov     al,(SerialSelect or Asic5NormalId)   ; Harware is there
        SBUSY                                        ; First look for An
        SCONTOUT                                     ; ASIC5 at the other
        XNOP                                         ; End of the link
        SBUSY
        SDATAIN
        test    al,al
        je      ConnectionFailed                        ; If not a 3link or
        test    al, mask A5MultiDrop                    ; if in multidrop
        jne     ConnectionFailedExit                    ; mode then we are
        popf                                             ; in trouble
        mov      al, SerialWriteSingle or A5PortBMode
         SBUSY
         SCONTOUT                                        ; Put the 3Link in
         mov     al, S_PERIPHERALMODE                    ; peripheral mode
         SBUSY
         SDATAOUT
        clc
        ret
        ProcEnd noret

        ProcBegin@ WaitTimer
;       ===================
;       In:  CX number of ms to wait for
;            Channel store in DL
;        Out: Channel store in DL

        inc     cx                                  ; Wait for  a given
WaitTickLoop:                                       ; Number of ms
        call    TickTimer                           ; Plus one to guarantee
        loop    WaitTickLoop                        ; That at least cx ms
        ret                                         ; Go by
        ProcEnd
;
        ProcBegin@ TickTimer
;        ====================

;       Uses writes down our channel to simulate tick timer waits
;       Must allow interrupts so other things can run -we will be here
;       for 1/1000 of a second and may be called many times
;        ChannelStore in/out in DL

        mov     al, dl                                  ; Return the old
        HwSelectChannel                                 ; channel
        mov     ah, al
        pushf
        sti
        push    cx
        mov     cx, 12                              ; 1ms
WaiterLoop:                                         ; 128 frames =1ms
        pushf                                       ; about 12 times
        cli                                             ; round the loop
        out    ResetWatchDog,al
        mov     al, ah
        HwSelectChannel
        mov     dl, al                                  ; Get Correct channel
        mov    al, (SerialWriteSingle or 0)         ; Do a write to
        SBUSY                                       ; nowhere and
        SCONTOUT                                    ; waste some time
        SBUSY
        SCONTOUT                                    ; Do it eight times
        SBUSY
        SCONTOUT
        SBUSY
        SCONTOUT
        SBUSY
        SCONTOUT
        SBUSY
        SCONTOUT
        SBUSY
        SCONTOUT
        SBUSY
        SCONTOUT
        SBUSY
        mov     al, dl                              ; Return the old
        HwSelectChannel                             ; channel
        mov     ah, al
        popf
        loop   WaiterLoop
        pop    cx
        popf
        mov     al, ah
        HwSelectChannel
        mov     dl, al
        ret
        ProcEnd

        EndCodeSeg

stack   segment stack para 'data'
stack   ends

        end OsAS5PDD
```

Assembler Macros

Excerpts from the include file ossibo.inc.

```
if ASIC1
SCONTOUT        macro
        out A2SerialControl, al
        endm
SDATAOUT        macro
        out     A2SerialData, al
        endm
SBUSY   macro
        wait
        endm
SREAD   macro _REG
        mov     al, SerialReadSingle or _REG
        out     A2SerialControl, al
        nop
        SBUSY
        in      al, A2SerialData
        endm
SREADM  macro _REG
        mov     al, SerialReadMulti or _REG
        out     A2SerialControl, al
        nop
        SBUSY
        in      al, A2SerialData
        endm
SWRITE  macro _REG,_VAL
        mov     al, SerialWriteSingle or _REG
        out     A2SerialControl, al
        SBUSY
        mov     al, _VAL
        out     A2SerialData, al
        endm
SWRITEM macro _REG,_VAL
        mov     al, SerialWriteMulti, _REG
        out     A2SerialControl, al
        SBUSY
        mov     al, _VAL
        out     A2SerialData, al
        endm
endif

if      ASIC9
SCONTOUT        macro
        out     A9BSerialControlW, al
        endm
SDATAOUT        macro
        out     A9BSerialDataRW, al
        endm
SDATAIN macro
        in      al, A9BSerialDataRW
        endm
SBUSY   macro
        endm
SREAD   macro _REG
        mov     al, SerialReadSingle or _REG
        out     A9BSerialControlW, al
        in      al, A9BSerialDataRW
        endm
SREADM  macro _REG
        mov     al, SerialReadMulti or _REG
        out     A9BSerialControlW, al
        in      al, A9BSerialDataRW
        endm
SWRITE  macro _REG,_VAL
        mov     al, SerialWriteSingle or _REG
        out     A9BSerialControlW, al
        mov     al, _VAL
        out     A9BSerialDataRW, al
        endm
SWRITEM macro _REG,_VAL
        mov     al, SerialWriteMulti or _REG
        out     A9BSerialControlW, al
        mov     al, _VAL
        out     A9BSerialDataRW, al
        endm
endif
```