Specialix International Ltd

SX PROGRAMMERS GUIDE

Document number: 6210028
Issue: 2.4
Date: 10 December, 1999

© SPECIALIX INTERNATIONAL LIMITED 1999. ALL RIGHTS RESERVED

Copyright in the whole and every part of this document belongs to Specialix International Limited ("Specialix") and may not be used, sold, transferred, copied or reproduced in whole or in part in any manner or form in or on any media to any person without the prior written agreement of Specialix

Specialix International Ltd

Prepared By	Checked By

Document number: 6210028
Issue: 2.45
Date: 10 December, 1999
Written by: Neil Vassallo
Total pages: 57 (2 headers)

Hardware Department	Software Department	Product Management
	Development Director	

Amendment record

Amend No.	Date	Text Affected	Issue
0	26/06/96	Document Created	DRAFT
1	08/07/96	First Issue	1.0
2	11/09/97	Update and rework to include host card programming notes and download code internals	2.0
3	24/12/97	Corrections to details Refer to modem signals as DTE signals Add host card interrupt description	2.1
4	23/04/98	Board names changes to SX Integrate standard SXWINDOW and SXBOARDS structure and definition naming Add sample code section	2.2
5	24/04/98	Further integration of sample source notation	2.3
6	05/03/99	Add SX+ host card details	2.4

Contents

1. Introduction 4
1.1 This document describes the functionality and operation of the SX Intelligent Serial Multiport product including: - SX+ Coldfire MCF5206e based host card - SX Transputer T225 based host card 4
1.2 SX host cards support the following interfaces to the Host PC: - Shared Memory Window Software Interface - SX Host Hardware Interface - PCI or ISA Hardware Interface 4
1.3 SX+ host cards support the following interfaces to the Host PC: - Shared Memory Window Software Interface - SX Host Hardware Interface - PCI Hardware Interface 4
1.4 This document contains programming notes describing initialisation, control and operation of the SX+ and SX cards using these interfaces. 4
1.5 Internally, the host cards execute a download program which supports access to another interface: - Extended SI-Bus Hardware Interface to TA, MTA or SXDC adapter modules 4
1.6 The basic internal structure of the download code program is discussed, along with details of support for the TA/MTA adapter modules. 4
1.7 Extensions to the Shared Memory Window Software Interface to support MTA ASIC based adapter modules (SXDC) are also discussed. 4
1.8 A sample source code kit demonstrating the SX+ and SX card mechanisms described in this document is available and details of the kit's purpose and contents are given. 4
1.9 It is intended that this document be used as a programming reference for the SX+ and SX product 4

2. Scope 4
2.1 This document describes the features, structures, constants and operations required to program the hardware and software interfaces of the SX+ and SX range of host cards 4
2.2 It also contains some internal details of the download code program run by the host card. 4
2.3 It does **not** contain full details of the TA/MTA adapters modules connected to the extended SI-Bus. These can be found in [ref4] - MTA, [ref6] - Rev1 TA and [ref7] - Rev2 TA 4
2.4 It also does **not** contain details of the SXDC module [ref5]. 4
2.5 This document is intended for use by all personnel involved in the development, testing and maintenance of the SX product. 5

3. Overview 6
3.1 Document Structure 6
3.2 Background History 6
3.3 Host Card Functionality 6
3.4 Shared Memory Window Interface 6
3.5 Download Code Program 6
3.6 Sample Source Code 6

4. Background History 7
4.1 Introduction 7
4.2 SX+ 7
4.3 SX 7
4.4 SI/XIO 8
4.5 Comparison of SX+, SX and SI/XIO Boards 9

5. Host Card Functionality 10
5.1 SX+ & SX Architecture Overview 10
5.2 Common SX+ & SX Hardware Interface 11
5.3 SX+ PCI Hardware Interface 14
5.4 SX PCI Hardware Interface 15
5.5 SX ISA Hardware Interface 16
5.6 Programming Notes 17

6. Shared Memory Window Interface 22
6.1 Shared Memory Window Interface Overview 22
6.2 Card Structure (SXWINDOW.H, SXCARD) 23

6.3 Module Structure (SXWINDOW.H, SXMODULE) ... 24
6.4 Channel Structure (SXWINDOW.H, SXCHANNEL) ... 27
6.5 Programming Notes ... 33

7. Download Code Program ... 44
7.1 Introduction ... 44
7.2 SX Basic Program Structure ... 44
7.3 SX+ Basic Program Structure ... 45
7.4 Initialisation and Identification of Modules ... 45
7.5 TA Modules ... 45
7.6 MTA Modules ... 46
7.7 SXDC Modules ... 47
7.8 Host Card Interrupts ... 48
7.9 Source Modules and Include Files ... 49

8. Sample Source Code ... 50
8.1 Sample Sources Overview ... 50
8.2 Sample Utilities ... 50
8.3 Building the Sample Utilities ... 52
8.4 Sample Modules and Functions ... 52
8.5 Download Include Files ... 54

9. References ... 56

Figures

Figure 1 SX+, SX & SI/XIO Board Features....9
Figure 2 SX+ Hardware Architecture....10
Figure 3 SX Hardware Architecture....10
Figure 4 SX+ & SX Memory Map (not to scale)....11
Figure 5 SX VPD ROM Structure....13
Figure 6 SX Unique Identification....13
Figure 7 SX+ PCI Board Identifiers....15
Figure 8 Shared Memory Window Structures....22
Figure 9 Baud Rate Indexes....30
Figure 10 Extended Baud Rate Indexes....32
Figure 11 Channel Commands and States....34
Figure 12 Transmit and Receive Buffer Structure....42
Figure 13 Relocation of SI3_T225.BIN....44

1. Introduction

1.1 This document describes the functionality and operation of the SX Intelligent Serial Multiport product including:
- SX+ Coldfire MCF5206e based host card
- SX Transputer T225 based host card

1.2 SX host cards support the following interfaces to the Host PC:
- Shared Memory Window Software Interface
- SX Host Hardware Interface
- PCI or ISA Hardware Interface

1.3 SX+ host cards support the following interfaces to the Host PC:
- Shared Memory Window Software Interface
- SX Host Hardware Interface
- PCI Hardware Interface

1.4 This document contains programming notes describing initialisation, control and operation of the SX+ and SX cards using these interfaces.

1.5 Internally, the host cards execute a download program which supports access to another interface:

- Extended SI-Bus Hardware Interface to TA, MTA or SXDC adapter modules.

1.6 The basic internal structure of the download code program is discussed, along with details of support for the TA/MTA adapter modules.

1.7 Extensions to the Shared Memory Window Software Interface to support MTA ASIC based adapter modules (SXDC) are also discussed.

1.8 A sample source code kit demonstrating the SX+ and SX card mechanisms described in this document is available and details of the kit's purpose and contents are given.

1.9 It is intended that this document be used as a programming reference for the SX+ and SX product.

2. Scope

2.1 This document describes the features, structures, constants and operations required to program the hardware and software interfaces of the SX+ and SX range of host cards.

2.1.1 The SI/XIO, Z280 based product is described in a separate document. [ref1].

2.2 It also contains some internal details of the download code program run by the host card.

2.3 It does **not** contain full details of the TA/MTA adapters modules connected to the extended SI-Bus. These can be found in [ref4] - MTA, [ref6] - Rev1 TA and [ref7] - Rev2 TA.

2.4 It also does **not** contain details of the SXDC module [ref5].

2.5 This document is intended for use by all personnel involved in the development, testing and maintenance of the SX product.

3. Overview

3.1 Document Structure

3.1.1 This document is set out as follows:
- Overview
- Background History
- Host Card Functionality
- Shared Memory Window Interface
- Download Code Program
- Sample Source Code
- References

3.2 Background History

3.2.1 This describes the evolution of the SI XIO product and the differences between the current SX product and the new SX+ range.

3.3 Host Card Functionality

3.3.1 This describes the hardware interfaces presented to the host PC by the cards and includes the following topics:
- SX+ and SX Architecture Overview
- Common SX Hardware Interface
- PCI and ISA Hardware Interfaces
- Programming Notes - Finding, Initialising, Interrupts

3.4 Shared Memory Window Interface

3.4.1 This defines the Shared Memory Window Software Interface presented to the Host PC driver / application by the download code. The following is defined:
- Shared Memory Window Interface Overview
- Card, Module and Channel Structures
- Programming Notes - Initialisation, Port Open/Close, Configuration and Input/Output

3.5 Download Code Program

3.5.1 Brief notes discussing the structure and operation of the SX+ and SX download code program including:
- Basic Program Structure
- Initialisation and Identification of Modules
- TA, MTA and SXDC Modules
- Source Modules and Include Files

3.6 Sample Source Code

3.6.1 Describes the contents of the SX sample source code, including:
- Sample Sources Overview
- Sample Utilities
- Building the sample sources
- Details of sample source modules and functions

4. Background History

4.1 Introduction

4.1.1 This section provides a brief history of the SX+, SX & SI/XIO intelligent serial multiport product.

4.1.2 The information is arranged in reverse chronological order, describing the latest product first and concluding with an overall comparison of product features.

4.2 SX+

4.2.1 SX+ is the latest evolution (phase 4) of the SX product range, based on the Coldfire MCF5206e processor and supporting the PCI [ref2] bus architecture.

4.2.1.1 Product architecture and operation is designed to be backward compatible with the SX product as far as possible, supporting the same shared memory window interface and SXDC, MTA and TA adapter modules.

4.2.1.2 SX+ host cards require a **new** Coldfire based download code program with similar operation to the T225 based version.

4.2.1.3 Initialisation of SX+ boards varies from the SX and SI/XIO boards (requiring modification to driver/application code), but is mostly common across the PCI card range.

4.3 SX

4.3.1 The SX product (phase 3) is a range of T225 transputer based cards supporting ISA [ref3] and PCI [ref2] bus architectures.

4.3.1.1 Product architecture and operation is designed to be backward compatible with SI/XIO (phase 2) supporting the same shared memory window interface and TA/MTA adapter modules.

4.3.1.2 SX host cards require a T225 based download code program with identical operation to the z280 based version.

4.3.1.3 Initialisation of SX boards varies from the SI/XIO boards (requiring modifications to driver/application code), but is mostly common across the ISA /PCI card range.

4.3.1.4 The SX family of boards also extends to other Specialix devices, (i.e. RIO) to provide a common host card interface for all products.

4.3.2 SX and SX+ support the current TAs and MTAs, and thenew SX Device Concentrator (SXDC).

4.3.2.1 The SXDC is based upon the Specialix MTA ASIC UART device which supports the following enhanced features:

- higher baud rates (up to 921600)
- deeper receive and transmit FIFOs (up to 32 bytes)
- improved hardware flow control (automatic DTR and RTS handshaking)
- new automatic software flow control (XON/XOFF)

4.4 SI/XIO

4.4.1 The original Specialix SI/XIO Intelligent Serial Multiport product (Phase 2) consists of Z280 processor based host cards for the ISA, EISA, MCA and PCI bus architectures.

4.4.1.1 Interface to the host machine is via a shared memory window and interrupt.

4.4.1.2 Serial and parallel ports are provided by adapter modules connected to the host card via an extended SI-Bus. The following types of modules are currently supported:

TA (Rev 1)	SCC2698 based Terminal Adapter
TA (Rev 2)	TA8 ASIC based Terminal Adapter
MTA	CL-CD1400 based Modular Terminal Adapter

4.4.1.3 Up to four adapters may be connected together, MTA and TA types cannot be mixed.

4.4.1.4 The product name is determined by the type of modules connected:

SI	=	Host Card + TA(s)
XIO	=	Host Card + MTA(s)

4.4.1.5 Control of the shared memory window structure and TA/MTA hardware modules is performed by a program running on the Z280, downloaded when the system is booted. [ref1]

4.5 Comparison of SX+, SX and SI/XIO Boards

4.5.1 The main features of the SX+, SX and SI/XIO boards are shown in Figure 1.

Board	Processor	Memory	Host Card Interface	Memory Window	Bitness	Interrupts	Software Interface	Terminal Adapters
Z280 ISA	Z280 (12.5MHz)	32 Kbytes	Specific to Z280 ISA board	32 Kbytes in 16Mbyte range, set by rotary/dip switches	8 bit hardware	11, 12 and 15	Shared Memory Window Interface	TA, MTA
Z280 EISA	Z280 (12.5MHz)	64 Kbytes	Specific to Z280 EISA board	64 Kbytes, configured by EISA configuration utility		3, 4, 5, 6, 7, 9, 10, 11, 12, 14 and 15	Shared Memory Window Interface	TA, MTA
Z280 MCA	Z280 (12.5MHz)	32 Kbytes	Specific to Z280 MCA board	32 Kbytes, configured by MCA configuration utility		5 and 9	Shared Memory Window Interface	TA, MTA
Z280 PCI	Z280 (12.5MHz)	64 Kbytes	Specific to Z280 PCI board	1 Mbyte in 4 Gbyte range selected by PCI BIOS / System	16 bit hardware/ 16 bit sofrware	PCI Selected	Shared Memory Window Interface	TA, MTA
SX ISA	T225 (25MHz)	64 Kbytes	Common to SX Family boards	32/64 Kbytes in 16 Mbyte range set by rotary/dip switches	8/16 bit hardware	9, 10, 11, 12, and 15	Shared Memory Window Interface	TA, MTA, SXDC
SX PCI	T225 (25 MHz)	64 Kbytes	Common to SX Family boards	64 Kbytes selected by PCI BIOS / System	16 bit hardware	PCI Selected	Shared Memory Window Interface	TA, MTA, SXDC
SX+ PCI	MCF5206e (40 MHz)	128 Kbytes	Common to SX Family boards	64 Kbytes selected by PCI BIOS / System	16 bit hardware	PCI Selected	Shared Memory Window Interface	TA, MTA, SXDC

Figure 1 SX+, SX & SI/XIO Board Features

4.5.1.1 The **bitness** field defines the width of the interface across the bus. There are two definitions:

- Hardware bitness. Handled by the hardware interface with byte/word swapping
- Software bitness. Changes required in software to support hardware *feature*

4.5.2 The main advantages of the SX boards over theSI/XIO range can be summarised as follows:

- Faster Processor
- Common Host Card Control Interface for all boards (also RIO)
- Support for SXDC Enhanced Features

4.5.3 The main advantages of the new SX+ boards over the current SX range can be summarised as follows:

- Faster Processor
- Larger possible Shared Memory Interface Window

5. Host Card Functionality

5.1 SX+ & SX Architecture Overview

5.1.1 Figure 2 and Figure 3 show the basic architecture and components of the SX+ & SX host cards.

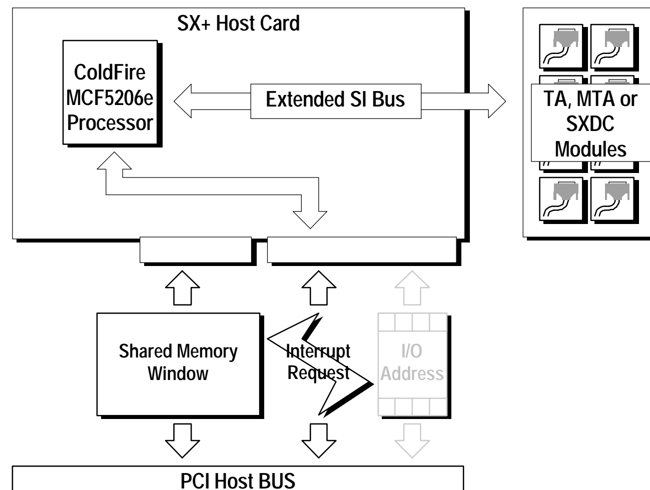


Figure 2 SX+ Hardware Architecture

5.1.2 The main interface to the Host PC is via a shared memory window, which is used to pass data to/from the card and an interrupt, which the card uses, to signal important events to the host. (e.g. data received by ports)

5.1.3 There is **no** I/O programming interface between the card processor and host, although the PCI cards do support an I/O interface for internal configuration purposes only.

5.1.4 Data is read and written to the shared memory window by the processor running a download code control program.

5.1.5 The download program transfers data to/from the TA, MTA and SXDC modules using a SI-Bus, which is extended outside the host PC using a Bus cable.

5.1.6 Thus, data is transferred between the host PC and the serial devices on the adapter modules by the download code program using the shared memory window, interrupt and SI-Bus.

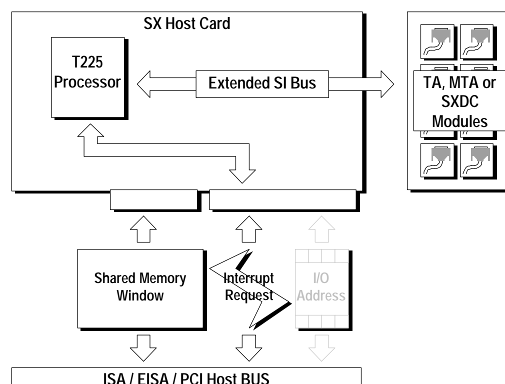


Figure 3 SX Hardware Architecture

5.2 Common SX+ & SX Hardware Interface

5.2.1 All variants of the SX+ & SX cards present the same shared memory window interface to the host PC. The main differences between different bus types relate to configuration and setup/control of interrupt.

5.2.2 Structures and constants defining the SX+ & SX memory map and registers are provided in the sample source header file SXBOARDS.H (See 8)

5.2.2.1 Structure and constant names from this file are included, where appropriate, in the following text as ***bold and italic***.

5.2.3 Figure 4 shows the shared memory map of the host card visible to both the host PC and the MCF5206e & T225 for all card types. (except the ISA board in 8-bit mode, when only the first 32K is visible)

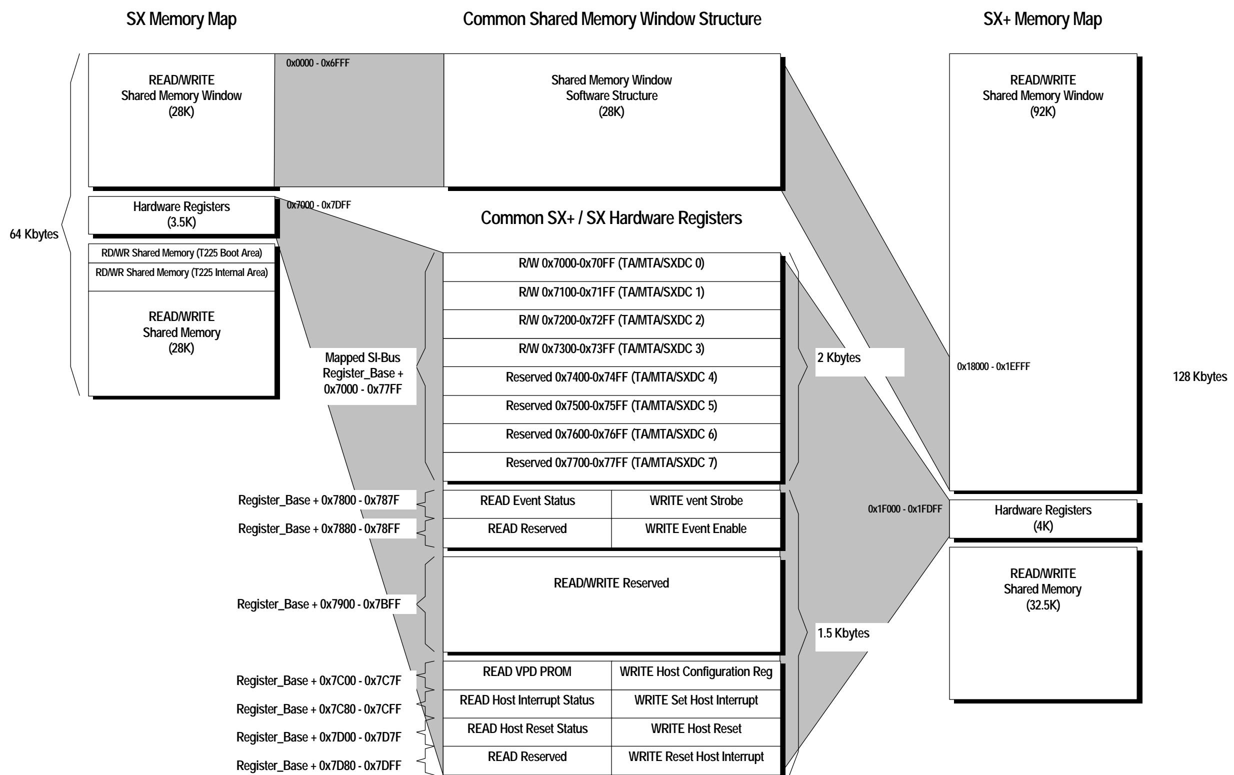


Figure 4 SX+ & SX Memory Map (not to scale)

5.2.4 The download code program is initially transferred to the following region:

- SX+ 0x00000 - 0x17FFF
- SX 0x0000 – 0x6FFF

5.2.4.1 Once the download code has initialised, the following region contains the Shared Memory Window Software Interface structures used to control the adapter devices and transfer data as described in 6:

- SX+ 0x18000 - 0x1EFFF
- SX 0x0000 – 0x6FFF

5.2.5 The SI-Bus is mapped allowing access to the TA, MTA or SXDC UART devices at addresses:
- SX+ 0x1F000 - 0x1F7FF
- SX 0x7000 - 0x77FF

5.2.5.1 This area is also visible to the host PC, allowing drivers / applications to access the adapter module hardware directly if desired. This feature was not available in phase 2 boards (except for z280 PCI).

5.2.6 SX hardware registers used to initialise, configure and control the host card appear at:
- SX+ 0x1F800 - 0x1FDFF
- SX 0x7800 - 0x7DFF

This area also contains identification strings and codes.

5.2.7 The following sections describe the hardware interface in detail and specify a register offset. This value is with respect to:
- SX+ 32K from end of memory window, i.e. 0x18000
- SX beginning of memory window, i.e. 0x0000

5.2.7.1 CF / T225 Event. The CF / T225 Event is a feature of the CF / T225 used to manage interrupts generated by modules attached to the extended SI-Bus. These interrupts are seen only by the download code program and are not presented to the host.

5.2.7.2 The CF / T225 Event Status/Strobe/Enable registers are provided mainly for diagnostic purposes and are not generally used by host applications or drivers.

5.2.7.3 **CF / T225 Event Status** (8-bit, read only, 0x7800-0x787F,***SX_EVENT_STATUS***).
A read of this register returns the current status of the Event:

7	6	5	4	3	2	1	0
Reserved							Event Status. If set, CF / T225 Event Pin asserted.

5.2.7.4 **CF / T225 Event Strobe** (8-bit, write only, 0x7800-0x787F,***SX_EVENT_STROBE***).
Write of any value to this register manually re-triggers a CF / T225 interrupt in the download code.

5.2.7.5 **CF / T225 Event Enable** (8-bit, write only, 0x7880-0x78FF, ***SX_EVENT_ENABLE***).
A write to this register allows the CF / T225 Event mechanism to be enabled/disabled.

7	6	5	4	3	2	1	0
Reserved							CF / T225 Event Enable. If set, Events are enabled.

5.2.7.6 **VPD PROM *(SX Only)***. (32 bytes, read only, 0x7C00-0x7C7F,***SX_VPD_ROM***).
This area contains Vital Product Data identification values and strings, allowing a host driver/application to uniquely identify a specific SX card. 32 bytes of information are presented to the host and are read every other byte as follows:

Offset	Label	Value	Description
0x00	***SX_VPD_SLX_ID1***	0x4D	Specialix Identifier
0x02	***SX_VPD_SLX_ID2***	0x98	Specialix Identifier
0x04	***SX_VPD_HW_REV***	0x??	Hardware Revision
0x06	***SX_VPD_HW_ASSEM***	0x??	Hardware Assembly Level
0x08	***SX_VPD_UNIQUEID4***	0x??	Unique Identifier Byte 4
0x0A	***SX_VPD_UNIQUEID3***	0x??	Unique Identifier Byte 3
0x0C	***SX_VPD_UNIQUEID2***	0x??	Unique Identifier Byte 2
0x0E	***SX_VPD_UNIQUEID1***	0x??	Unique Identifier Byte 1
0x10	***SX_VPD_MANU_YEAR***	0x??	Year Of Manufacture
0x12	***SX_VPD_MANU_WEEK***	0x??	Week Of Manufacture
0x14		0x00	Hardware Feature Byte 0
0x16		0x00	Hardware Feature Byte 1
0x18		0x00	Hardware Feature Byte 2
0x1A		0x00	Hardware Feature Byte 3
0x1C		0x00	Hardware Feature Byte 4
0x1E		0x00	OEM Identifier
0x20	***SX_VPD_IDENT***	0x4A 'J'	SX Identifier String
0x22		0x45 'E'	SX Identifier String
0x24		0x54 'T'	SX Identifier String
0x26		0x20 ' '	SX Identifier String
0x28		0x48 'H'	SX Identifier String
0x2A		0x4F 'O'	SX Identifier String
0x2C		0x53 'S'	SX Identifier String
0x2E		0x54 'T'	SX Identifier String
0x30		0x20 ' '	SX Identifier String
0x32		0x42 'B'	SX Identifier String
0x34		0x59 'Y'	SX Identifier String
0x36		0x20 ' '	SX Identifier String
0x38		0x4B 'K'	SX Identifier String
0x3A		0x45 'E'	SX Identifier String
0x3C		0x56 'V'	SX Identifier String
0x3E		0x23 '#'	SX Identifier String

Figure 5 SX VPD ROM Structure

5.2.7.6.1 Currently all SX boards are uniquely identified using bits 7-4 of the VPD ROM Unique Identifier Byte 1 (***SX_VPD_UNIQUEID1***):

SX Board	Unique Identifier 1
SX ISA	0x2?
SX PCI	0x5?
Jet RIO ISA	0xA?
Jet RIO PCI	0xD?

Figure 6 SX Unique Identification

5.2.7.6.2 Year of Manufacture (***SX_VPD_MANU_YEAR***) is relative to 1970, i.e. a value of 0x1C would indicate 1998.

5.2.7.6.3 Week of Manufacture (***SX_VPD_MANU_WEEK***) starts with 0 corresponding to the first week in January.

5.2.7.6.4 The VPD PROM method of identification is only guaranteed to be available on ISA cards. Although it is currently present on the SX PCI, it is not present on the SX+ PCI cards. Programmers should not rely on it being present and should use the Sub Vendor and Sub Device ID's to uniquely determine the type of board.

5.2.7.7 **Host Configuration Register** (8-bit, write only, 0x7C00-0x7C7F,***SX_CONFIG***).
This register allows features of the host card to be enabled/disabled:

7	6	5	4	3	2	1	0
Card Dependant / Reserved				Reserved	Host Interrupt Enable. If set, card can interrupt the host	T225 Bus Enable. If set, T225 can access memory and I/O on its bus.	Reserved

5.2.7.8 **Host Interrupt Status** (8-bit, read only, 0x7C80-0x7CFF,***SX_IRQ_STATUS***).
A read to this register returns the current host interrupt status. Generally used to confirm that an interrupt was caused by a specific card and is useful when implementing interrupt sharing.

7	6	5	4	3	2	1	0
Reserved							Host Interrupt Status. If clear, host interrupt is asserted.

5.2.7.9 **Set Host Interrupt** (8-bit, write only, 0x7C80-0x7CFF,***SX_SET_IRQ***).
A write of any value to this register manually triggers the host interrupt. Generally used for diagnostic purposes.

5.2.7.10 **Host Reset Status** (8-bit, read only, 0x7D00-0x7D7F.***SX_RESET_STATUS***).
A read to this register indicates whether the host card is in a reset state. Generally used in conjunction with a write to the Host Reset register when resetting a host card.

7	6	5	4	3	2	1	0
Reserved							Board Reset Status. If set, host card is being reset.

5.2.7.11 **Host Reset** (8-bit, write only, 0x7D00-0x7D7F,***SX_RESET***).
A write of any value to this register causes a minimum reset pulse of 3∓s to be applied to the host card. This will reset the processor and the TA/MTA/SXDCs and host interrupt, but will <u>not</u> reset the Host Configuration Register.

5.2.7.12 **Reset Host Interrupt**. (8-bit, read only, 0x7D80-0x7DFF,***SX_RESET_IRQ***).
A write of any value to this register resets an asserted host interrupt. Generally used within an interrupt service routine to reset a host card interrupt after interrupt processing.

5.2.8 On the SX card the region 0x7EFF - 0x7FFF is used during initialisation to contain a small bootstrap program which the T225 executes at location 0x7FFE. This is not used in the SX+

5.2.9 On the SX card the region 0x9000 - 0xFFFF is used to contain the download code program after initialisation. On the SX+ the code is located from 0x00

5.3 SX+ PCI Hardware Interface

5.3.1 The SX+ PCI host card is automatically configured by the PCI aware BIOS/System and requests the following resources:

- PLX-9050 Configuration Registers. 128 bytes of memory (PCI Config Register BADR0)
- PLX-9050 Configuration Registers. 128 bytes of I/O space (PCI Config Register BADR1)
- SX+ Shared Memory Window. 128Kbytes of memory (PCI Config Register BADR3)
- Host Card Interrupt.

5.3.2 The PLX-9050 configuration registers are used to configure/control the PLX PCI-9050 Bus Agent device. These registers are pre-set and do not require action by a host application/driver.

5.3.3 The SX+ shared memory window and host card interrupt are automatically allocated by the PCI aware BIOS/System and generally do not require further action by a host application/driver, other than to request for the card to be set up.

5.3.4 An SX+ host card is identified by examining the following PCI registers obtained from the BIOS/System:

PCI Vendor ID Word	0x11CB	(***SPX_VENDOR_ID***)
PCI Device ID Word	0x2000	(***SPX_PLXDEVICE_ID***)
PCI Subsystem Vendor ID Word	0x11CB	(***SPX_SUB_VENDOR_ID***)
PCI Subsystem Device ID Word	0x0300	(***CSX_SUB_SYS_ID***)

5.3.5 To help the design of PCI board search functions the following table shows the PCI Vendor/Device IDs for all current Specialix devices:

	Vendor Id	Device ID	Subsystem Vendor ID	Subsystem Device ID
SX+ PCI	0x11CB	0x2000	0x11CB	0x0300
z280 SI XIO PCI	0x11CB	0x4000	0x11CB	0x0400
SX PCI	0x11CB	0x2000	0x11CB	0x0200
RIO PCI	0x11CB	0x8000	0x11CB	0x0800
Jet RIO PCI	0x11CB	0x2000	0x11CB	0x0100
I/O8+ PCI	0x11CB	0x2000	0x11CB	0xB008

Figure 7 SX+ **PCI Board Identifiers**

5.4 SX PCI Hardware Interface

5.4.1 The SX PCI host card is automatically configured by the PCI aware BIOS/System and requests the following resources:

- PCI-9050 Configuration Registers. 128 bytes of memory (PCI Config Register BADR0)
- PCI-9050 Configuration Registers. 128 bytes of I/O space (PCI Config Register BADR1)
- SX Shared Memory Window. 64Kbytes of memory (PCI Config Register BADR2)
- Host Card Interrupt.

5.4.2 The PCI-9050 configuration registers are used to configure/control the PLX PCI-9050 Bus Agent device. These registers are pre-set and do not require action by a host application/driver.

5.4.3 The SX shared memory window and host card interrupt are automatically allocated by the PCI aware BIOS/System and generally do not require further action by a host application/driver, other than to request for the card to be set up.

5.4.4 An SX host card is identified by examining the following PCI registers obtained from the BIOS/System:

PCI Vendor ID Word	0x11CB	(***SPX_VENDOR_ID***)
PCI Device ID Word	0x2000	(***SPX_PLXDEVICE_ID***)
PCI Subsystem Vendor ID Word	0x11CB	(***SPX_SUB_VENDOR_ID***)
PCI Subsystem Device ID Word	0x0200	(***SX_SUB_SYS_ID***)

5.5 SX ISA Hardware Interface

5.5.1 The host card base memory address is manually configured using external dip / rotary switches and an external jumper. (See Section 4 of [ref3])

5.5.2 The ISA board may operate in 8-bit or 16-bit mode, set manually with an external jumper. In 8-bit mode only 32Kbytes of the shared memory window is available, in 16-bit mode all 64Kbytes are available. (See Section 4 of [ref3])

5.5.3 The board is located/identified by searching memory space for the following VPD ROM values:

5.5.3.1 **SX Identifier String** (See 5.2.7.6) set to "JET HOST BY KEV#'

5.5.3.2 **Unique Identifier Byte 1** (See 5.2.7.6.1) bits 7-4 set to 0x2?

5.5.4 The host card interrupt is configured using the **Host Configuration Register** (See 5.2.7.6.2) as follows:

7	6	5	4	3	2	1	0
Level of Host Card Interrupt. (valid settings: 9, 10, 11, 12, 15)				Reserved	Host Interrupt Enable. If set, card can interrupt the host	T225 Bus Enable. If set, T225 can access memory and I/O on its bus.	Reserved

5.6 Programming Notes

5.6.1 The following section contains descriptions of common operation generally performed by host applications/drivers to locate, initialise, configure and control the SX host card.

5.6.1.1 Reference are also made to the sample sources indicating (in brackets) the source modules and functions in which a given mechanism is demonstrated.

5.6.2 **Card Initialisation Overview** (CARD.C, CardInitialise)

5.6.2.1 The following actions are generally performed to initialise/configure a SX host card:

a. Locate/Identify Card
b If SX+, adjust pointer to shared memory
c. Stop Card
d. Install Download Program
e. Start Card and Enable/Configure Interrupt
f. Use shared memory window to interface with card (See 6)

5.6.3 **Locate/Identify Card** (CARD.C, CardFindPCI, CardFindISA)

5.6.3.1 a. Locate card, using method appropriate to ISA/PCI bus.
b. If ISA, Identify card by checking VPD PROM values in shared memory window.

5.6.3.2 ISA boards are generally pre-defined by a system setup utility and only require identification. However, on some systems memory space may be scanned over the ISA board address range for the VPD PROM SX Identification String (See 5.2.7.6)

5.6.3.3 PCI boards are automatically set up by the PCI aware BIOS/System.

5.6.3.3.1 The board parameters are determined by calling the PCI BIOS or Operating System resource manager to the check the PCI ID Values.

5.6.3.3.2 The shared memory window base address is defined by PCI register BADR2 for SX cards, BADR3 for SX+ cards.

5.6.3.3.3 The interrupt level is defined by PCI register INTLN.

5.6.3.4 For ISA cards, once the board has been located, then the appropriate identification values/strings in the shared memory window VPD PROM must be checked.

5.6.3.5 For SX+ cards, the base of the shared memory window and hardware registers is calculated as follows:

base address = actual base address + window length – 32K

5.6.3.6 For SX cards, the base address is the same as the actual base address.

5.6.4 **Stop Card** (and place in a known, steady, non-running state) (CARD.C, CardStop)

5.6.4.1
a. Set the Host Configuration Register (See 5.2.7.6.2) to 0.
b. Set the Host Reset Register (See 5.2.7.11) to any value.
c. Poll the Host Reset Status Register (See 5.2.7.10) for bit 0 to clear.

5.6.4.2 The host reset register resets the processor, TA/MTAs and the host interrupt. It **does not** reset the board configuration register. The value written to it also **does not** matter. The write only generates a 3 microsecond reset pulse, not a state.

5.6.4.3 So, a reset will cause the processor to begin execution again from its start point.

5.6.4.4 Therefore, the host configuration register is set to 0 **first**, disabling the processor bus and effectively preventing the processor from executing as it cannot access memory or I/O space.

5.6.4.5 Example:

```
int     CardStop(PCARD pCard)
{
        int     loop = 0;

        pCard->pBase[SX_CONFIG] = 0;
        pCard->pBase[SX_RESET] = 0;
        while((pCard->pBase[SX_RESET]&1) && loop++<10000);
        if(pCard->pBase[SX_RESET]&1)
                return(0);              /* Timeout */
        return(1);                      /* Stopped OK */

} /* CardStop*/
```

The timeout shown here is a tight loop, a more accurate method should be applied in driver software.

5.6.5 **Start Card** (CARD.C, CardStart)

5.6.5.1 *Assuming card has been stopped and download code copied into shared memory window*:

a. If SX, copy small bootstrap code to the processor start address (0x7FFA..0x7FFF)
b. Set the Host Reset Register (See 5.2.7.11) to any value.
c. Poll the Host Reset Status Register (See 5.2.7.10) for bit 0 to clear.
d. Set bit 1 of the Host Configuration Register (See 5.2.7.6.2) to enable the processor bus.

5.6.5.2 SX Cards. Once the processor bus is enabled, the processor begins execution of the bootstrap which transfers control to the download code at location 0 of the shared memory window.

5.6.5.3 SX+ Cards. Once it's bus is enabled, the processor begins execution of the code at 0 which contains the vector to the start of the actual code.

5.6.5.4 Example:

```
int     CardStart(PCARD pCard)
{
        pu8     code_p = "\x28\x20\x21\x02\x60\x0a"; /* Bootstrap */
        int     loop;

        if((pCard->Type == TYPE_SX_ISA)
        ||(pCard->Type == TYPE_SX_PCI))
        {
                for(loop = 0; loop < 6; loop++)
                        pCard->pBase[0x7FFA + loop] = *code_p++;
        }
        else if (pCard->Type = TYPE_SX+)
                pCard->pBase += CSX_SM_OFFSET;


        pCard->pBase[SX_RESET] = 1;                     /* Start card */

        loop = 0;
        while((pCard->pBase[SX_RESET]&1) && loop++<10000);
        if(pCard->pBase[SX_RESET]&1)
                return(0);                              /* Timeout */

        pCard->pBase[SX_CONFIG] = SX_CONF_BUSEN;        /* Enable bus */
        return(1);

} /* CardStart */
```

Again, the timeout mechanism should be more elegant.

5.6.6 **Download Card** (CARD.C, CardLoad)

5.6.6.1
a. Stop Card (See 5.6.4)
b. Copy download code to shared memory window from location 0.
c. Start Card (See 5.6.5)

5.6.6.2 It is obviously important that the card is placed in a known non-active state before copying across a new download image.

5.6.6.3 Example:

Here the download code image exists in an allocated buffer pointed to by *pImage* and of length *ImageLen*.

```
.
.
.
if(CardStop(pCard) == 0)                      /* Stop the card */
{
        fprintf(stderr,"ERROR: Unable to stop card.\n");
        free(pImage);                         /* Free allocated memory */
        return(0);                            /* Fail */
}
for(loop = 0; loop < ImageLen; loop++)
        pCard->pBase[loop] = pImage[loop];    /* Copy image to card */
free(pImage);                                 /* Free allocated memory */
if(CardStart(pCard) == 0)
{
        fprintf(stderr,"ERROR: Unable to start card.\n");
        return(0);                            /* Fail */
}
.
.
.
```

5.6.7 **Configuring Card Interrupts** (CARD.C, CardEnableIrq, CardDisableIrq)

5.6.7.1
a. Set bit 2 of the Host Configuration Register (See 5.2.7.6.2) to enable host interrupt.
b. If ISA board, set bits 7..4 of Host Configuration Register to be Host PIC IRQ Level.

5.6.7.2 Note that bit 1 of the host configuration register should also be set at the same time, otherwise the processor bus will be disabled(!). The register is defined as write only, and so **cannot** be read, ORed and written back.

5.6.7.3 It is likely that the interrupt will be set at the same time as the card is started (see 2 of this note) and can be performed as one write if desired.

5.6.7.4 The Host Configuration Register (See 5.2.7.6.2) allows the ISA IRQ level to be set, e.g. if an ISA card is to be enabled at IRQ 15, then a value of 0xF6 should be set in the host configuration register. Supported interrupts are: 15, 12, 11, 10 and 9.

5.6.7.5 Example:

```
void        CardEnableIrq(PCARD pCard)
{
            pCard->pBase[SX_CONFIG] =
                        SX_CONF_BUSEN|SX_CONF_HOSTIRQ|(pCard->IrqLevel<<4);

} /* CardEnableIrq */
```

5.6.8 **Handling Card Interrupts** (CARD.C, CardIsr)

5.6.8.1
a. If desired, test bit 0 of the Host Interrupt Status Register (See 5.2.7.8) to confirm that the interrupt was generated by this card.
b. When interrupt processing complete, write any value to the Host Interrupt Reset Register (See 5.2.7.12) to reset the host interrupt.

5.6.8.2 Note that "a." is optional and isn't implemented by the SI/XIO boards. This is intended for drivers and operating systems that support interrupt sharing and allows the driver to determine very quickly whether the interrupt was issued by the SX card, or not.

5.6.8.3 Example:

```
.
.
.
for(loop = 0; loop < NumberOfCards; loop++)
{
            pCard = CardTable[loop];                                /* Get Card structure */
            if(pCard->pBase[SX_IRQ_STATUS) & 0x1) continue;         /* No interrupt */

            /* Process interrupt for pCard, or request deferred procedure call (DPC) */

            pCard->pBase[SX_RESET_IRQ] = 0;                         /* Reset interrupts */
}
.
.
.
```

6. Shared Memory Window Interface

6.1 Shared Memory Window Interface Overview

6.1.1 The Shared Memory Window Interface is a software interface constructed by the Download Code Program during its initialisation process.

6.1.2 The shared memory window interface is defined by the SXWINDOW.H file released with the download code program and references are made to this file in the following sections.

6.1.3 Figure 8 shows the typical structure layout of a 32 port SI/XIO system.

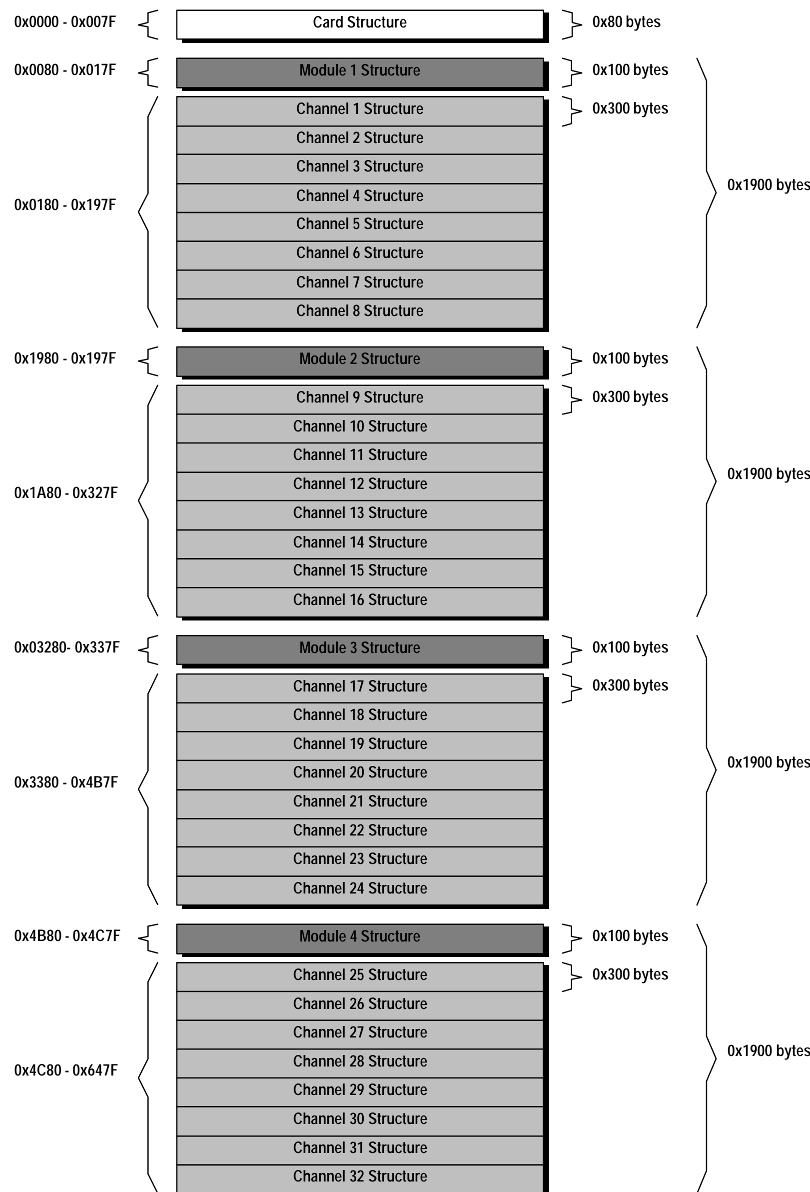


Figure 8 Shared Memory Window Structures

6.1.4 The interface consists of three structure types: Card, Module and Channel and is available to a host application/driver once the card structure cc_init_stat field becomes non-zero (see 6.2.2.1).

6.1.5 A single Card Structure exists, describing the global features of the card.

6.1.6 Up to 4 Module Structures exist corresponding to and describing the modules attached to the extended SI-Bus.

6.1.7 Up to 8 Channel Structures exist per Module corresponding to and describing the serial/parallel port channels provided by a given module. A full populated system contains 32 channel structures.

6.1.8 The layout and definition of these structures is common for the Phase 2 product and the new Phase 3 SX range. However, the SX version may contain extensions for new or enhanced features.

6.1.9 The following structure definitions are given in 'C' notation where:

```
#define    unsigned char     BYTE     /* Unsigned 8bits */
#define    unsigned short    WORD     /* Unsigned 16bits */
```

6.2 Card Structure (SXWINDOW.H, SXCARD)

6.2.1 The card structure is 0x80 bytes long and describes features global to the host card.

6.2.2

```
typedef    struct    _SXCARD
{
    BYTE    cc_init_status;          /* 0x00 Initialisation status */
    BYTE    cc_mem_size;             /* 0x01 Size of memory on card */
    WORD    cc_int_count;            /* 0x02 Interrupt count */
    WORD    cc_revision;             /* 0x04 Download code revision */
    BYTE    cc_isr_count;            /* 0x06 Count when ISR is run */
    BYTE    cc_main_count;           /* 0x07 Count when main loop is run */
    WORD    cc_int_pending;          /* 0x08 Interrupt pending */
    WORD    cc_poll_count;           /* 0x0A Count when poll is run */
    BYTE    cc_int_set_count;        /* 0x0C Count when host interrupt is set */
    BYTE    cc_curr_status;          /* 0x0C Count when host interrupt is set */
    BYTE    cc_rfu[0x80 - 0x0C];     /* 0x0C Pad structure to 128 bytes (0x80) */

} SXCARD;
```

6.2.2.1 **cc_init_status**. This field is polled for a non-zero value immediately after installing the download code program and starting the card. The following values are possible:

```
#define    ADAPTERS_FOUND       0x01    /* Initialisation done */
#define    NO_ADAPTERS_FOUND    0xFF    /* Initialisation found no modules */
```

6.2.2.2 **cc_mem_size**. Defines the amount of memory installed in the host card. Possible values are:

```
#define    CC_MEM_32K    0x20    /* 32k of dual-port RAM fitted */
#define    CC_MEM_64K    0x40    /* 64k of dual-port RAM fitted */
```

6.2.2.3 **cc_int_count**. For z280 based cards this field is reserved. For SX/SX+ based cards this field defines the frequency of interrupts generated by the card to the host. If not set, a default value of 100 (Hz) is assumed.

6.2.2.4 **cc_revision**. Defines the current revision of the download code program. The upper byte defines the major version and the lower byte the minor version, e.g. version "2.0.3" is represented as 0x0203.

6.2.2.5 **cc_isr_count**, is only supported for SX/SX+ based cards and used for internal diagnostics.

6.2.2.6 **cc_main_count**, is only supported for SX/SX+ based cards and used for internal diagnostics.

6.2.2.7 **cc_int_pending**. Set by the download code to indicate an interrupt pending to the host. This should be reset by the host application/driver when processing of an interrupt is complete.

6.2.2.8 **cc_poll_count**, is only supported for SX/SX+ based cards and used for internal diagnostics.

6.2.2.9 **cc_int_set_count**, is only supported for SX/SX+ based cards and used for internal diagnostics.

6.2.2.10 **cc_curr_status**,is only supported for SX/SX+ based cards and is used for internal diagnostics. On the SX+ Host Card, the on Board Leds reflect this field. The following is the meaning of the bits.

Bit/Led 0 - Adapters Found
Bit/Led 1 - No Adapters Found

Bit/Leds 2 and 3 will be an encoding of the type of system where a value of

0 = TA System
1 = MTA System
2 = SXDC System
3 = Adapter Clash

Bit/Led 4 - Some ports are open
Bit/Led 5 - Transmit Overload
Bit/Led 6 - Receive overload
Bit/Led 7 - Firmware Cycle (should flash)

6.3 Module Structure (SXWINDOW.H, SXMODULE)

6.3.1 This structure is 0x100 bytes long and defines the features of a given module.

6.3.2 A module structure is created for each TA, MTA or SXDC module detected on the extended SI-Bus.

6.3.3

6.3.4

```
typedef struct _SXMODULE
{
    WORD  mc_next;                /* 0x00 Next module "pointer" (ORed with 0x8000) */
    BYTE  mc_type;                /* 0x02 Type of TA in terms of number of channels */
    BYTE  mc_mod_no;              /* 0x03 Module number on SI bus cable */
    BYTE  mc_dtr;                 /* 0x04 Private DTR copy (TA only) */
    BYTE  mc_rfu1;                /* 0x05 Reserved */
    WORD  mc_uart;                /* 0x06 UART base address for this module */
    BYTE  mc_chip;                /* 0x08 Chip type / number of ports */
    BYTE  mc_current_uart;        /* 0x09 Current uart selected for this module */
    WORD  mc_chan_pointer[8];     /* 0x0A Pointers to channel structures */
    WORD  mc_rfu2;                /* 0x1A Reserved */
    BYTE  mc_opens1;              /* 0x1C No. open ports on 1st  4 MTA/SXDC ports */
    BYTE  mc_opens2;              /* 0x1D No. open ports on 2nd 4 MTA/SXDC ports */
    BYTE  mc_mods;                /* 0x1E MTA/SXDC connector module types */
    BYTE  mc_rev1;                /* 0x1F Revision of first CD1400 on MTA/SXDC */
    BYTE  mc_rev2;                /* 0x20 Revision of second CD1400 on MTA/SXDC */
    BYTE  mc_mtaasic_rev;         /* 0x21 Revision of MTA ASIC 1..4 -> A, B, C, D */
    BYTE  mc_rfu3[0x100 - 0x22];  /* 0x22 Pad structure to 256 bytes (0x100) */

} SXMODULE;
```

6.3.4.1 **mc_next**. Defines a pointer to the next Module Structure as an offset from the base of the shared memory window ORed with 0x8000. If ((mc_next & 0x7FFF) == 0) then there are no more modules in the system.

6.3.4.2 **mc_type**. Defines the number of ports supported by the module. Possible values are 4 or 8.

6.3.4.3 **mc_mod_no**. Specifies the module number attached to the Si-Bus. Possible values are 0,1,2 or 3, where 0 is the closest module to the host card.

6.3.4.4 **mc_uart**. Offset from the base of the shared memory window to the base of the UART device memory for this module.

6.3.4.5 **mc_chip**. Defines the type of module attached to the Si-Bus. Possible values are:

```
#define  FOUR_PORTS    (BYTE)4
#define  EIGHT_PORTS   (BYTE)8
#define  TA            (BYTE)0

#define  TA4           (TA|FOUR_PORTS)
#define  TA8           (TA|EIGHT_PORTS)
#define  TA4_ASIC      (BYTE)0x0A
#define  TA8_ASIC      (BYTE)0x0B
#define  MTA_CD1400    (BYTE)0x28
#define  SXDC          (BYTE)0x48
```

6.3.4.6 **mc_chan_pointer[8]**. An array of offsets from the base of the shared memory window corresponding to the channel control structures for this module.

6.3.4.7 **mc_opens1**. The number of opens in the first 4 ports of an MTA/SXDC module.

6.3.4.8 **mc_opens2**. The number of opens in the second 4 ports of an MTA/SXDC module.

6.3.4.9 **mc_mods**. Defines the revision of hardware module used by an MTA/SXDC module.

Some possible values are:

```
#define MOD_RS232DB25       0x00  /* RS232 DB25 (socket/plug) */
#define MOD_RS232RJ45       0x01  /* RS232 RJ45 (shielded/opto-isolated) */
#define MOD_RS422DB25       0x03  /* RS422 DB25 Socket */
#define MOD_PARALLEL        0x05  /* Parallel */
#define MOD_2_RS232DB25     0x08  /* Rev 2.0 RS232 DB25 (socket/plug) */
#define MOD_2_RS232RJ45     0x09  /* Rev 2.0 RS232 RJ45 */
#define MOD_RS232DB25MALE   0x0A  /* SXDC with Male connectors */
#define MOD_2_RS422DB25     0x0B  /* Rev 2.0 RS422 DB25 */
#define MOD_2_PARALLEL      0x0D  /* Rev 2.0 Parallel */
#define MOD_BLANK           0x0F  /* Blank Panel */
```

6.4 Channel Structure (SXWINDOW.H, SXCHANNEL)

6.4.1 This structure is 0x300 bytes long and defines the features of a given channel.

6.4.2

```
typedef     struct      _SXCHANNEL
{
      WORD  next_item;            /* 0x00 Window offset of next channels hi_txbuf */
      WORD  addr_uart;            /* 0x02 INTERNAL pointer to uart address. */
      WORD  module;               /* 0x04 Window offset of parent SXMODULE */
      BYTE  type;                 /* 0x06 Chip type / number of ports  */
      BYTE  chan_number;          /* 0x07 Channel number on the TA/MTA/SXDC */
      WORD  xc_status;            /* 0x08 Flow control and I/O status */
      BYTE  hi_rxipos;            /* 0x0A Receive buffer input index */
      BYTE  hi_rxopos;            /* 0x0B Receive buffer output index */
      BYTE  hi_txopos;            /* 0x0C Transmit buffer output index */
      BYTE  hi_txipos;            /* 0x0D Transmit buffer input index */
      BYTE  hi_hstat;             /* 0x0E Command register */
      BYTE  dtr_bit;              /* 0x0F INTERNAL DTR control byte (TA only) */
      BYTE  txon;                 /* 0x10 INTERNAL copy of hi_txon */
      BYTE  txoff;                /* 0x11 INTERNAL copy of hi_txoff */
      BYTE  rxon;                 /* 0x12 INTERNAL copy of hi_rxon */
      BYTE  rxoff;                /* 0x13 INTERNAL copy of hi_rxoff */
      BYTE  hi_mr1;               /* 0x14 Mode Register 1 (databits,parity,RTS rx flow)*/
      BYTE  hi_mr2;               /* 0x15 Mode Register 2 (stopbits,local,CTS tx flow)*/
      BYTE  hi_csr;               /* 0x16 Clock Select Register (baud rate) */
      BYTE  hi_op;                /* 0x17 Modem Output Signal */
      BYTE  hi_ip;                /* 0x18 Modem Input Signal */
      BYTE  hi_state;             /* 0x19 Channel status */
      BYTE  hi_prtcl;             /* 0x1A Channel protocol (flow control) */
      BYTE  hi_txon;              /* 0x1B Transmit XON character */
      BYTE  hi_txoff;             /* 0x1C Transmit XOFF character */
      BYTE  hi_rxon;              /* 0x1D Receive XON character */
      BYTE  hi_rxoff;             /* 0x1E Receive XOFF character */
      BYTE  close_prev;           /* 0x1F INTERNAL channel previously closed flag */
      BYTE  hi_break;             /* 0x20 Break and error control */
      BYTE  break_state;          /* 0x21 INTERNAL copy of hi_break */
      BYTE  hi_mask;              /* 0x22 Mask for received data */
      BYTE  mask;                 /* 0x23 INTERNAL copy of hi_mask */
      BYTE  mod_type;             /* 0x24 MTA/SXDC hardware module type */
      BYTE  ccr_state;            /* 0x25 INTERNAL MTA/SXDC state of CCR reg */
      BYTE  ip_mask;              /* 0x26 Input handshake mask */
      BYTE  hi_parallel;          /* 0x27 Parallel port flag */
      BYTE  par_error;            /* 0x28 Error code for parallel loopback test */
      BYTE  any_sent;             /* 0x29 INTERNAL data sent flag */
      BYTE  asic_txfifo_size;     /* 0x2A INTERNAL SXDC transmit FIFO size */
      BYTE  rfu1[2];              /* 0x2B Reserved */
      BYTE  csr;                  /* 0x2D INTERNAL copy of hi_csr */
      WORD  nextp;                /* 0x2E Window offset of next channel structure */
      BYTE  prtcl;                /* 0x30 INTERNAL copy of hi_prtcl */
      BYTE  mr1;                  /* 0x31 INTERNAL copy of hi_mr1 */
      BYTE  mr2;                  /* 0x32 INTERNAL copy of hi_mr2 */
      BYTE  hi_txbaud;            /* 0x33 Extended transmit baud rate (SXDC only) */
      BYTE  hi_rxbaud;            /* 0x34 Extended receive baud rate  (SXDC only) */
      BYTE  txbreak_state;        /* 0x35 INTERNAL MTA/SXDC transmit break state */
      BYTE  txbaud;               /* 0x36 INTERNAL copy of hi_txbaud */
      BYTE  rxbaud;               /* 0x37 INTERNAL copy of hi_rxbaud */
      WORD  err_framing;          /* 0x38 Count of receive framing errors */
      WORD  err_parity;           /* 0x3A Count of receive parity errors */
      WORD  err_overrun;          /* 0x3C Count of receive overrun errors */
      WORD  err_overflow;         /* 0x3E Count of receive buffer overflow errors */
      BYTE  rfu2[TX_BUFF_OFF - 0x40];   /* 0x40 Reserved until hi_txbuf */
      BYTE  hi_txbuf[BUFFER_SIZE];      /* 0x060 Transmit buffer */
      BYTE  hi_rxbuf[BUFFER_SIZE];      /* 0x160 Receive buffer */
      BYTE  rfu3[0x300 - 0x260];  /* 0x260 Reserved until 768 bytes (0x300) */

} SXCHANNEL;
```

6.4.3 **next_item**. Defines a pointer to the transmit buffer of the next Channel Structure as an offset from the base of the shared memory window ORed with 0x8000.

6.4.4 **addr_uart**. Offset from the base of the shared memory window to the base of the UART device for this channel.

6.4.5 **module**. Offset from the base of the shared memory window to the parent Module Structure.

6.4.6 **type**. See mc_type (6.3.4.2).

6.4.7 **hi_rxipos**. Input pointer for the channel receive buffer as an index into hi_rxbuf.

6.4.8 **hi_rxopos**. Output pointer for the channel receive buffer as an index into hi_rxbuf.

6.4.9 **hi_txopos**. Output pointer for the channel transmit buffer as an index into hi_txbuf.

6.4.10 **hi_txipos**. Input pointer for the channel transmit buffer as an index into hi_txbuf.

6.4.11 **hi_hstat**. Command register controlling channel states (See 0). The following commands / states are possible:

```
#define HS_IDLE_OPEN    0x00 /* Channel open state */
#define HS_LOPEN        0x02 /* Local open command */
#define HS_MOPEN        0x04 /* Modem open command */
#define HS_IDLE_MPEND   0x06 /* Waiting for DCD signal state */
#define HS_CONFIG       0x08 /* Configuration command */
#define HS_CLOSE        0x0A /* Close command */
#define HS_START        0x0C /* Start transmit break command */
#define HS_STOP         0x0E /* Stop transmit break command */
#define HS_IDLE_CLOSED  0x10 /* Closed channel state */
#define HS_IDLE_BREAK   0x12 /* Transmit break state */
#define HS_FORCE_CLOSED 0x14 /* Force close command */
#define HS_RESUME       0x16 /* Clear pending XOFF command */
#define HS_WFLUSH       0x18 /* Flush transmit buffer command */
#define HS_RFLUSH       0x1A /* Flush receive buffer command */
#define HS_SUSPEND      0x1C /* Suspend output command */
```

6.4.12 **hi_mr1**. Mode register 1 provides configuration for Parity, Databits and RTS receive handshaking. The register is a bit mask defined as follows:

7	6	5	4	3	2	1	0
RTS Receive Flow Control	Reserved		Parity			Databits	

```
#define MR1_BITS    0x03 /* Data bits mask */
#define MR1_5_BITS  0x00 /* 5 data bits */
#define MR1_6_BITS  0x01 /* 6 data bits */
#define MR1_7_BITS  0x02 /* 7 data bits */
#define MR1_8_BITS  0x03 /* 8 data bits */

#define MR1_PARITY  0x1C /* Parity mask */
#define MR1_ODD     0x04 /* Odd parity */
#define MR1_EVEN    0x00 /* Even parity */
#define MR1_WITH    0x00 /* Parity enabled */
```

```
#define     MR1_FORCE         0x08                /* Force parity */
#define     MR1_NONE          0x10                /* No parity */

#define     MR1_NOPARITY      MR1_NONE            /* No parity */
#define     MR1_ODDPARITY     (MR1_WITH|MR1_ODD)  /* Odd parity */
#define     MR1_EVENPARITY    (MR1_WITH|MR1_EVEN) /* Even parity */
#define     MR1_MARKPARITY    (MR1_FORCE|MR1_ODD) /* Mark parity */
#define     MR1_SPACEPARITY   (MR1_FORCE|MR1_EVEN) /* Space parity */

#define     MR1_RTS_RXFLOW    0x80                /* RTS receive flow control */
```

6.4.13 **hi_mr2**. Mode Register 2 provides configuration for Stopbits and CTS transmit handshaking. The register is a bit mask defines as follows:

```
#define     MR2_STOP          0x0F                /* Stop bits mask */
#define     MR2_1_STOP        0x07                /* 1 stop bit */
#define     MR2_2_STOP        0x0F                /* 2 stop bits */

#define     MR2_CTS_TXFLOW    0x10                /* CTS transmit flow control */
#define     MR2_RTS_TOGGLE    0x20                /* RTS toggle on transmit */

#define     MR2_NORMAL        0x00                /* Normal mode */
#define     MR2_AUTO          0x40                /* Auto-echo mode (TA only) */
#define     MR2_LOCAL         0x80                /* Local echo mode */
#define     MR2_REMOTE        0xC0                /* Remote echo mode (TA only) */
```

6.4.14 **hi_csr**. Clock Select Register defines the receive / transmit baud rate for the channel as follows:

Valid baud rate index values are shown in Figure 9.

Index	Label	TA Baud Rate	MTA Baud Rate	SXDC Baud Rate
0x0	CSR_75	75	75	75
0x1	CSR_110	110	115200	115200
0x2	CSR_38400	38400	38400	38400
0x3	CSR_150	150	150	150
0x4	CSR_300	300	300	300
0x5	CSR_600	600	600	600
0x6	CSR_1200	1200	1200	1200
0x7	CSR_2000	2000	2000	2000
0x8	CSR_2400	2400	2400	2400
0x9	CSR_4800	4800	4800	4800
0xA	CSR_1800	1800	1800	1800
0xB	CSR_9600	9600	9600	9600
0xC	CSR_19200	19200	19200	19200
0xD	CSR_57600	57600	57600	57600
0xE		Reserved	Reserved	Reserved
0xF	CSR_EXTBAUD	Reserved	Reserved	Use Extended Rate Index (See hi_txbaud & hi_rxbaud)

Figure 9 Baud Rate Indexes

6.4.15 **hi_op**. Output Control Register. This is a bitmask allowing the following output modem signals to be controlled:

7	6	5	4	3	2	1	0
Reserved						DTR	RTS

```
#define    OP_RTS    0x1    /* Set RTS */
#define    OP_DTR    0x2    /* Set DTR */
```

6.4.16 **hi_ip**. Input Status Register. Bitmask returning the current status of the input modem signals as follows:

7	6	5	4	3	2	1	0
Reserved	RI	DSR	Reserved	Reserved	DCD	CTS	Reserved

```
#define    IP_CTS    0x02    /* CTS is set */
#define    IP_DCD    0x04    /* DCD is set */
#define    IP_DSR    0x20    /* DSR is set */
#define    IP_RI     0x40    /* RI is set */
```

6.4.17 **hi_state**. Status register reporting events and error conditions. Bitmask defined as follows:

7	6	5	4	3	2	1	0
Reserved			Parity Error	Framing Error	Overrun Error	DCD	Receive Break

```
#define    ST_BREAK    0x01
#define    ST_DCD      0x02
#define    ST_OE       0x04    /* TA only */
#define    ST_FE       0x08    /* TA only */
#define    ST_PE       0x10    /* TA only */
```

6.4.18 **hi_prtcl**. Protocol Register. Bitmask defining the flow control mode of the channel as follows:

7	6	5	4	3	2	1	0
Check Parity of Receive Chars	DTR Receive Flow Control	Monitor Input Modem Signals	Reserved	Receive Xon/Xoff Flow Control	Reserved	Transmit Xon/Xoff Flow	Transmit Xon/Xany Flow Control (only if Transmit Xon/Xoff set)

```
#define    SP_TANY          0x01    /* Transmit XON/XANY */
#define    SP_TXEN          0x02    /* Transmit XON/XOFF flow control */
#define    SP_RXEN          0x08    /* Rx XON/XOFF enabled */
#define    SP_DCEN          0x20    /* Monitor modem signals */
#define    SP_DTR_RXFLOW    0x40    /* DTR receive flow control */
#define    SP_PAEN          0x80    /* Parity checking enabled */
```

6.4.19 **hi_txon**. Defines the XON character used with transmit XON/XOFF flow control.

6.4.20 **hi_txoff**. Defines the XOFF character used with transmit XON/XOFF flow control.

6.4.21 **hi_rxon**. Defines the XON character used with receive XON/XOFF flow control.

6.4.22 **hi_rxoff**. Defines the XOFF character used with receive XON/XOFF flow control.

6.4.23 **close_prev**. Indicates if the channel was previously closed.

6.4.24 **hi_break**. Received Break and Parity processing mode. Bitmask defined as follows:

7	6	5	4	3	2	1	0
Treat parity/framing/overrun errors as exceptions	Reserved			If set, received parity errors are ignored	If set, parity errors replaced with 0xFF, 0x00, *data*, where *data* is character received in error	If set, receive break causes interrupt and ST_BREAK set in hi_state, else Null character and parity / framing error	If set, all received breaks are ignored.

```
#define    BR_IGN       0x01    /* Ignore any received breaks */
#define    BR_INT       0x02    /* Interrupt on received break */
#define    BR_PARMRK    0x04    /* Enable parmrk parity error processing */
#define    BR_PARIGN    0x08    /* Ignore chars with parity errors */
#define    BR_ERRINT    0x80    /* Treat parity/framing/overrun errors as exceptions */
```

6.4.25 **hi_mask**. All received characters are ANDed with this mask, i.e. a value of 0x7F will strip the top bit of each received byte.

6.4.26 **hi_parallel**. If set, then this channel is a parallel port, else a serial port.

6.4.27 **hi_txbaud**. Extended transmit baud rate indexes, only used if ((hi_csr & 0xF) == 0x0F). Possible indexes are shown in Figure 10.

Index	Label	SXDC Baud Rate
0x00	BAUD_75	75
0x01	BAUD_115200	115200
0x02	BAUD_38400	38400
0x03	BAUD_150	150
0x04	BAUD_300	300
0x05	BAUD_600	600
0x06	BAUD_1200	1200
0x07	BAUD_2000	2000
0x08	BAUD_2400	2400
0x09	BAUD_4800	4800
0x0A	BAUD_1800	1800
0x0B	BAUD_9600	9600
0x0C	BAUD_19200	19200
0x0D	BAUD_57600	57600
0x0E	BAUD_230400	230400
0x0F	BAUD_460800	460800
0x10	BAUD_921600	921600
0x11	BAUD_50	50
0x12	BAUD_110	110
0x13	BAUD_134_5	134.5
0x14	BAUD_200	200
0x15	BAUD_7200	7200
0x16	BAUD_56000	56000
0x17	BAUD_64000	64000
0x18	BAUD_76800	76800
0x19	BAUD_128000	128000
0x1A	BAUD_150000	150000
0x1B	BAUD_14400	14400
0x1C	BAUD_256000	256000
0x1D	BAUD_28800	28800

Figure 10 Extended Baud Rate Indexes

6.4.28 **hi_rxbaud**. Extended receive baud rate indexes, only used if ((hi_csr & 0xF0) == 0x0F0). See hi_txbaud (6.4.26) for valid index values.

6.5 Programming Notes

6.5.1 Shared Memory Window Initialisation (CARD.C, CardInitWindow)

6.5.1.1 Following installation of the download code program and starting the card, the download code initialisation will detect and identify all modules attached to the extended Si-Bus and build the shared memory window interface. (See 7.4)

6.5.1.2 A host application/driver should poll the cc_init_stat field of the card structure for a non-zero value. Possible values are:

0x01 Initialisation complete, modules found. Driver may continue with initialisation.
0xFF Initialisation complete, no modules detected. The SI-Bus is empty.

6.5.2 Module and Channel Detection (CARD.C, CardInitWindow)

6.5.2.1 If modules are detected, then the host application/driver can proceed to check the module structures to determine the number and type of modules and channels available.

6.5.2.1.1 The first module structure immediately follows the card structure. (See Figure 8)

6.5.2.1.2 The next module structure can be determined by examining the mc_next field (See 6.3.4.1). If ((mc_next & 0x7FFF) == 0) then there are no more modules. Otherwise (mc_next & 0x7FFF) defines an offset from the base of the shared memory window of the next module structure.

6.5.2.1.3 The number of channels present on a module can be found using the mc_type field.

6.5.2.1.4 The type of attached module can be determined using the mc_chip field.

6.5.2.1.5 The first channel structure for a given module immediately follows the module structure.

6.5.2.1.6 The next channel structure immediately follows the first channel structure, and so on.

6.5.3 Channel Programming (CHAN.C, ChanOpen, ChanConfig, ChanClose)

6.5.3.1 A host application/driver gains access to a channel by performing the following actions:

6.5.3.1.1 Check initial port state in the hi_hstat field. This must be HS_IDLE_CLOSED, otherwise the port is being used by another process.

6.5.3.1.2 Set up initial channel configuration. The following channel structure fields should be configured:

- hi_mr1 (See 6.4.12)
- hi_mr2 (See 6.4.13)
- hi_csr (See 6.4.14)
- hi_op (See 6.4.15)
- hi_prtcl (See 6.4.18)
- hi_txon (See 6.4.19)
- hi_txoff (See 6.4.20)

- hi_rxon (See 6.4.21)
- hi_rxoff (See 6.4.22)
- hi_break (See 6.4.24)
- hi_mask (See 6.4.25)
- hi_txbaud (Optional, See 6.4.26)
- hi_rxbaud (Optional, See 0)

6.5.3.1.3 Set the hi_hstat command register to HS_LOPEN for a local open, or HS_MOPEN for a modem open.

6.5.3.1.4 Poll the hi_hstat register for the HS_IDLE_OPEN state.

6.5.3.1.5 Once this state has been reached, data may be sent / received from the port using the transmit/receive buffers and the port and further commands may be issued using the hi_hstate register.

6.5.3.2 When the host application/driver no longer requires access to a channel, control is relinquished with the following sequence of actions:

6.5.3.2.1 Check that the current hi_hstat status is HS_IDLE_OPEN.

6.5.3.2.2 Set the hi_hstat field to HS_CLOSE.

6.5.3.2.3 Poll the hi_hstat field for the HS_IDLE_CLOSED state.

6.5.3.2.4 It is possible that the port will not close for various reasons (e.g. data still transmitting). In this case, a HS_FORCE_CLOSED command may be used instead of HS_CLOSE, which will clear all conditions preventing a normal close. (i.e. flush transmit and receive buffers)

6.5.3.3 Channel Commands and States

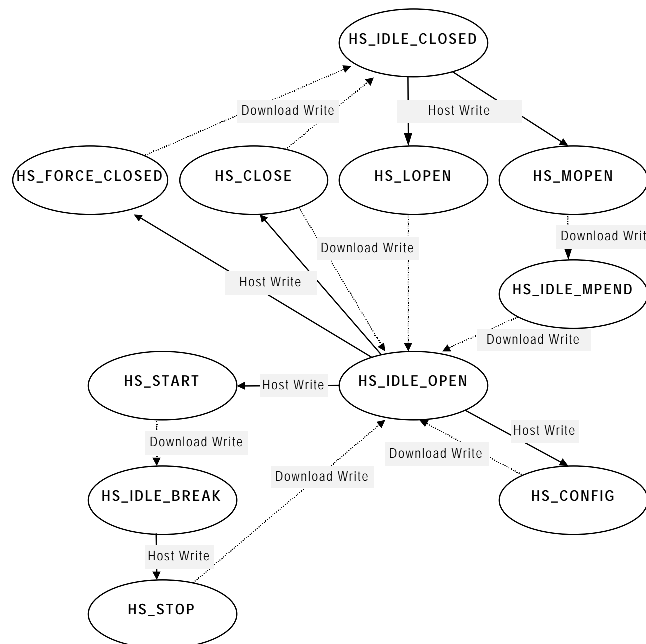


Figure 11 Channel Commands and States

6.5.3.3.1 Figure 11 shows possible channel commands and states set using the hi_hstat field.

6.5.3.3.2 HS_IDLE_CLOSED is the (initial) closed state of the port, set by the Download Code.

6.5.3.3.3 HS_LOPEN is a local open command, set by the Host Application/Driver. The download code will attempt to open the port using the preconfigured channel structure and then set the status to HS_IDLE_OPEN.

6.5.3.3.4 Example:

```
#define         TIMEOUT_OPEN   1000

PCHAN           ChanOpen(PCARD pCard,int nChan)
{
        PCHAN   pChan;
        _u32    Timeout;

        if(pCard == NULL)
                return(NULL);                   /* Invalid Card structure */

        if((nChan >= 0)&&(nChan < pCard->nChannels))
                pChan = pCard->ChannelTable[nChan];
        else    return(NULL);                   /* Invalid channel range */

/* Try to open the port on the card... */

        if(pChan->hi_hstat != HS_IDLE_CLOSED)
                return(NULL);                   /* Port not closed */

        pChan->hi_hstat = HS_LOPEN;             /* Issue open command */

/* Wait for the channel to open... */

        Timeout = 0;                            /* Reset timeout */
        while((pChan->hi_hstat != HS_IDLE_OPEN)&&(Timeout < TIMEOUT_OPEN))
                Timeout += Sys_DelaymS(100);    /* Wait for a bit */

        if(pChan->hi_hstat == HS_IDLE_OPEN)
                return(pChan);                  /* Success */
        else    return(NULL);                   /* Failure */

} /* ChanOpen */
```

6.5.3.3.5 HS_MOPEN is a modem open command, set by the Host Application/Driver. The download code will attempt to open the port using the preconfigured channel structure and will set the status to HS_IDLE_MPEND.

6.5.3.3.5.1 The download code will remain in this state until the DCD modem input line is raised, whereupon the state will be changed to HS_IDLE_OPEN.

6.5.3.3.6 HS_CLOSE is a close command, set by the Host Application/Driver. The download code will attempt to close the port and if successful set the state to HS_IDLE_CLOSED. If unsuccessful, (possible if data is still waiting to be transmitted, but flow control is preventing this), then the state remain as HS_CLOSE. If, after a timeout period, the state

remains HS_CLOSE, then a force close may be issued to clear any conditions preventing the normal close.

6.5.3.3.7 Example:

```
#define         TIMEOUT_CLOSE 1000

int     ChanClose(PCHAN pChan)
{
        _u32    Timeout;

        if(pChan == NULL)
                return(0);              /* Invalid Channel structure */

/* Try to close the channel on the card... */

        if(pChan->hi_hstat != HS_IDLE_OPEN)
                return(0);              /* Port not closed */

        pChan->hi_hstat = HS_CLOSE;     /* Issue close command */

/* Wait for the channel to close... */

        Timeout = 0;                    /* Reset timeout */
        while((pChan->hi_hstat != HS_IDLE_CLOSED)&&(Timeout < TIMEOUT_CLOSE))
                Timeout += Sys_DelaymS(100);

        if(pChan->hi_hstat == HS_IDLE_CLOSED)
                return(1);              /* Success */

/* Normal close didn't work, try a force close... */

        pChan->hi_hstat = HS_FORCE_CLOSED;/* Issue force close command */
        Timeout = 0;                    /* Reset timeout */
        while((pChan->hi_hstat != HS_IDLE_CLOSED)&&(Timeout < TIMEOUT_CLOSE))
                Timeout += Sys_DelaymS(100);

        if(pChan->hi_hstat == HS_IDLE_CLOSED)
                return(1);              /* Success */
        else    return(0);              /* Fail */

} /* ChanClose */
```

6.5.3.3.8 HS_FORCE_CLOSED is a force close command, set by the Host Application/Driver. The download code will close the port, resetting any conditions which may have prevented a normal close, before setting the status to HS_IDLE_CLOSED.

6.5.3.3.9 HS_START is a start transmit break command, set by the Host Application/Driver. The download code prepares the port to transmit a break signal and then sets the status to HS_IDLE_BREAK. If a HS_CLOSE or HS_FORCE_CLOSED is issued while the state is HS_IDLE_BREAK, then the break signal is lowered, before proceeding with the normal close/force close.

6.5.3.3.10 HS_STOP is a stop transmit break command, set by the Host Application/Driver. The download code stops the break signal and returns the status to HS_IDLE_OPEN. This command may only be issued if the current state is HS_IDLE_BREAK.

6.5.3.3.11 HS_CONFIG is a configure command issued by the Host Application/Driver. While the port is opened, any of the channel structure configuration fields set during the port open (See 6.5.3.1.2) may be altered. These changes will not be applied to the port until a HS_CONFIG command is issued to notify the download code. The download code reconfigures the port using the current channel structure fields and returns the state to HS_IDLE_OPEN.

6.5.3.3.12 Example:

The following example uses as input a configuration structure defined below and shows the typical translation process from an internal driver application structure to the channel structure format:

```
typedef     struct     _CFG
{
        _u32    Changes;        /* Bitmask defining fields to set up */
        _u32    TxBaud;         /* Transmit baud rate */
        _u32    RxBaud;         /* Receive baud rate */
        _u8     Parity;         /* Parity setting */
        _u8     Databits;       /* Data bits */
        _u8     Stopbits;       /* Stop bits */
        _u8     TxFlow;         /* Transmit flow control */
        _u8     RxFlow;         /* Receive flow control */
        _u8     XonChar;        /* XON character */
        _u8     XoffChar;       /* XOFF character */
        _u16    Mode;           /* Various I/O modes */

} CFG, *PCFG;

/* CONFIG.Changes settings... */
#define         CFG_ALL         0xFFFFFFFF
#define         CFG_TXBAUD      0x00000001
#define         CFG_RXBAUD      0x00000002
#define         CFG_PARITY      0x00000004
#define         CFG_DATABITS    0x00000008
#define         CFG_STOPBITS    0x00000010
#define         CFG_TXFLOW      0x00000020
#define         CFG_RXFLOW      0x00000040
#define         CFG_XONCHAR     0x00000080
#define         CFG_XOFFCHAR    0x00000100
#define         CFG_MODE        0x00000400

/* CONFIG.Parity settings... */
#define         PAR_NONE        0x00
#define         PAR_ODD         0x01
#define         PAR_EVEN        0x02
#define         PAR_MARK        0x03
#define         PAR_SPACE       0x04

/* CONFIG.Stopbits settings... */
#define         STOPB_1         0x00
```

```
#define         STOPB_1_5       0x01
#define         STOPB_2         0x02

/* CONFIG.TxFlow settings... */
#define         TXF_NONE        0x00
#define         TXF_XANY        0x01
#define         TXF_XONXOFF     0x02
#define         TXF_DSR         0x10
#define         TXF_CTS         0x20

/* CONFIG.RxFlow settings... */
#define         RXF_NONE        0x00
#define         RXF_XONXOFF     0x02
#define         RXF_DTR         0x10
#define         RXF_RTS         0x20

#define         XON_CHAR        0x11
#define         XOFF_CHAR       0x13

/* CONFIG.Mode settings... */
#define         MODE_LOCAL      0x0001  /* Local loopback mode */

typedef     struct  _BAUDXLAT
{
    _u32    baudrate;
    _u8     csr_index;
    _u8     baud_index;

} BAUDXLAT;

BAUDXLAT    BaudTable[] =
{
    {50,        0xF,    0x11},
    {75,        0x0,    0x00},
    {110,       0xF,    0x12},
    {134,       0xF,    0x13}
    {150,       0x3,    0x03},
    {200,       0xF,    0x14},
    {300,       0x4,    0x04},
    {600,       0x5,    0x05},
    {1200,      0x6,    0x06},
    {1800,      0xA,    0x0A},
    {2000,      0x7,    0x07},
    {2400,      0x8,    0x08},
    {4800,      0x9,    0x09},
    {7200,      0xF,    0x15},
    {9600,      0xB,    0x0B},
    {14400,     0xF,    0x1B},
    {19200,     0xC,    0x0C},
    {28800,     0xF,    0x1D},
    {38400,     0x2,    0x02},
    {56000,     0xF,    0x16},
    {57600,     0xD,    0x0D},
    {64000,     0xF,    0x17},
    {76800,     0xF,    0x18},
    {115200,    0x1,    0x01},
```

```
		{128000,		0xF,	0x19},
		{150000,		0xF,	0x1A},
		{230400,		0xF,	0x0E},
		{256000,		0xF,	0x1C},
		{460800,		0xF,	0x0F},
		{921600,		0xF,	0x10}

};

#define		MAXBAUDINDEX	(sizeof(BaudTable)/sizeof(BAUDXLAT))
#define		TIMEOUT_CONFIG		1000

int	ChanConfig(PCHAN pChan,PCFG pCfg)
{
	_u32	Timeout;
	int	loop;

	if((pChan == NULL)||(pCfg == NULL))
		return(0);			/* Invalid Channel structure */

	if(pCfg->Changes == 0) return(1);	/* Nothing to do */

/* Try to configure the port... */

	if(pChan->hi_hstat != HS_IDLE_OPEN)
		return(0);			/* Port not open */

/* Transmit baud rate... */

	if(pCfg->Changes & CFG_TXBAUD)
	{
		for(loop = 0; loop < MAXBAUDINDEX; loop++)
		{
			if(pCfg->TxBaud == BaudTable[loop].baudrate)
			{
				if((BaudTable[loop].csr_index) == 0xF)
				&&(pChan-> chip != MTA_ASIC))
					return(0);		/* Not SXDC */
				pChan->hi_csr = (pChan->hi_csr & 0xF0)
					| BaudTable[loop].csr_index;
				pChan->hi_txbaud =
					BaudTable[loop].baud_index;
				break;
			}
		}
	}

/* Receive baud rate... */

	if(pCfg->Changes & CFG_RXBAUD)
	{
		for(loop = 0; loop < MAXBAUDINDEX; loop++)
		{
			if(pCfg->RxBaud == BaudTable[loop].baudrate)
			{
				if((BaudTable[loop].csr_index) == 0xF)
```

```
                                        &&(pChan-> chip != MTA_ASIC))
                                                return(0);              /* Not SXDC */
                                        pChan->hi_csr = (pChan->hi_csr & 0x0F)
                                                | (BaudTable[loop].csr_index << 4);
                                        pChan->hi_rxbaud =
                                                BaudTable[loop].baud_index;
                                        break;
                                }
                        }
                }

        /* Parity... */

                if(pCfg->Changes & CFG_PARITY)
                {
                        switch(pCfg->Parity)
                        {
                        case    PAR_NONE:
                        pChan->hi_mr1 = (pChan->hi_mr1 & 0xE3) | MR1_NONE;
                        break;
                        case    PAR_ODD:
                        pChan->hi_mr1 = (pChan->hi_mr1 & 0xE3) | MR1_WITH|MR1_ODD;
                        break;
                        case    PAR_EVEN:
                        pChan->hi_mr1 = (pChan->hi_mr1 & 0xE3) | MR1_WITH|MR1_EVEN;
                        break;
                        case    PAR_MARK:
                        pChan->hi_mr1 = (pChan->hi_mr1 & 0xE3) | MR1_FORCE|MR1_ODD;
                        break;
                        case    PAR_SPACE:
                        pChan->hi_mr1 = (pChan->hi_mr1 & 0xE3) | MR1_FORCE|MR1_EVEN;
                        break;
                        default:
                        break;
                        }
                }

        /* Databits... */

                if(pCfg->Changes & CFG_DATABITS)
                {
                        switch(pCfg->Databits)
                        {
                        case    5:
                                pChan->hi_mr1 = (pChan->hi_mr1&~MR1_BITS)|MR1_5_BITS;
                                break;
                        case    6:
                                pChan->hi_mr1 = (pChan->hi_mr1&~MR1_BITS)|MR1_6_BITS;
                                break;
                        case    7:
                                pChan->hi_mr1 = (pChan->hi_mr1&~MR1_BITS)|MR1_7_BITS;
                                break;
                        case    8:
                                pChan->hi_mr1 = (pChan->hi_mr1&~MR1_BITS)|MR1_8_BITS;
                                break;
                        default:        break;
```

```
                }
        }

    /* Stopbits... */

        if(pCfg->Changes & CFG_STOPBITS)
        {
                switch(pCfg->Stopbits)
                {
                case    STOPB_1:
                pChan->hi_mr2 = (pChan->hi_mr2 & ~MR2_STOP) | MR2_1_STOP;
                break;
                case    STOPB_2:
                pChan->hi_mr2 = (pChan->hi_mr2 & ~MR2_STOP) | MR2_2_STOP;
                break;
                default:
                break;
                }
        }

    /* Transmit flow control... */

        if(pCfg->Changes & CFG_TXFLOW)
        {
                pChan->hi_mr2 = (pChan->hi_mr2 & ~(MR2_RTSCONT));
                pChan->hi_prtcl = (pChan->hi_prtcl & ~(SP_TANY|SP_TXEN));
                if(pCfg->TxFlow&TXF_XANY)           pChan->hi_prtcl |= SP_TANY;
                if(pCfg->TxFlow&TXF_XONXOFF)        pChan->hi_prtcl |= SP_TXEN;
                if(pCfg->TxFlow&TXF_CTS)  pChan->hi_mr2 |= MR2_CTS_TXFLOW;
        }

    /* Receive flow control... */

        if(pCfg->Changes & CFG_RXFLOW)
        {
                pChan->hi_mr1 = (pChan->hi_mr1 & ~(MR1_CTSCONT));
                pChan->hi_prtcl = (pChan->hi_prtcl & ~(SP_RXEN));
                if(pCfg->TxFlow&RXF_XONXOFF)        pChan->hi_prtcl |= SP_RXEN;
                if(pCfg->TxFlow&RXF_RTS)  pChan->hi_mr1 |= MR1_RTS_RXFLOW;
        }

    /* XON/XOFF characters... */

        if(pCfg->Changes & CFG_XONCHAR)
        {
                pChan->hi_txon = pCfg->XonChar;
                pChan->hi_rxon = pCfg->XonChar;
        }
        if(pCfg->Changes & CFG_XOFFCHAR)
        {
                pChan->hi_txoff = pCfg->XoffChar;
                pChan->hi_rxoff = pCfg->XoffChar;
        }

    /* Try to configure the port on the card... */
```

```
        pChan->hi_hstat = HS_CONFIG;                    /* Issue configuration command */

    /* Wait for the port to complete configuration... */

        Timeout = 0;                                    /* Reset timeout */
        while((pChan->hi_hstat != HS_IDLE_OPEN)&&(Timeout < TIMEOUT_OPEN))
                Timeout += Sys_DelaymS(100);            /* Wait for a bit */

        if(pChan->hi_hstat == HS_IDLE_OPEN) return(1);          /* Success */
        else        return(0);

    } /* ChanConfig */
```

6.5.4 Transmitting and Receiving Data (CHAN.C, ChanRead, ChanWrite)

6.5.4.1 Data is transmitted and received from a port using the circular transmit and receive buffers. See Figure 12.

6.5.4.2 Each buffer is 256 character long and is controlled using input and output pointers.

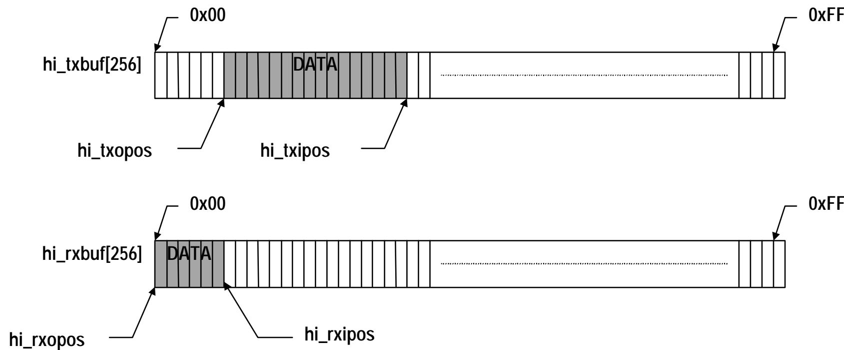


Figure 12 Transmit and Receive Buffer Structure

6.5.4.3 Transmitting data is performed with the transmit buffer hi_txbuf.

6.5.4.3.1 hi_txipos specifies the index into the transmit buffer of the first free space. This field is only altered by the Host Application/Driver as data is placed in the buffer. When the index reaches 255, the pointer wraps back to 0.

6.5.4.3.2 hi_txopos specified the index into the transmit buffer of data to transmit. This field is only altered by the Download Code as data is removed from the buffer for transmission. When the index reaches 255, the pointer wraps back to 0.

6.5.4.3.3 Once data is placed in the transmit buffer it is removed by the download code and placed into the UART FIFOs. No further action is necessary from the driver/application.

6.5.4.3.4 data_in_buffer = hi_txipos - hi_txopos

6.5.4.3.5 space_in_buffer = (hi_txopos - hi_txipos -1) & 0xFF

6.5.4.3.6 To transmit a single character:

```
if(((hi_txipos + 1) & 0xFF) != hi_txopos                /* Is there space in buffer ? */
{
        hi_txbuf[hi_txipos] = data;                     /* Set data in buffer */
        hi_txipos = ((hi_txipos + 1) & 0xFF);           /* Update input pointer */
}
```

6.5.4.4 Receiving data is performed with the receive buffer hi_rxbuf.

6.5.4.4.1 hi_rxipos specifies the index into the receive buffer of the first free space. This field is only altered by the Download Code when data is received from the port. When the index reaches 255, the pointer wraps back to 0.

6.5.4.4.2 hi_rxopos specified the index into the receive buffer of data to read. This field is only altered by the Host Application/Driver as data is read from the buffer. When the index reaches 255, the pointer wraps back to 0.

6.5.4.4.3 data_in_buffer = hi_rxipos - hi_rxopos

6.5.4.4.4 space_in_buffer = (hi_rxopos - hi_rxipos -1) & 0xFF

6.5.4.4.5 To receive a single character:

```
if(hi_rxopos != hi_rxipos)                              /* Is there data in buffer ? */
{
        data = hi_rxbuf[hi_rxopos];                     /* Get data from buffer */
        hi_rxopos = ((hi_rxopos + 1) & 0xFF);           /* Update output pointer */
}
```

7. Download Code Program

7.1 Introduction

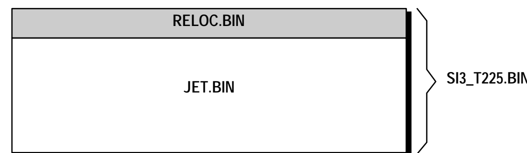
7.1.1 This section gives a brief description of the host card download code program operation.

7.1.2 The download code sources are not available for general modification, however, internal details are given as an aid to understanding and to help with driver/application design and development.

7.2 SX Basic Program Structure

7.2.1 The download code's main function is to present terminal adapter hardware on the extended SI-Bus to the host PC using the shared memory window structure and host interrupt.

7.2.2 The download code image (SI3_T225.BIN) consists of two concatenated components; the Relocation Program (RELOC.BIN) and the Main Control Program (JET.BIN)



7.2.2.1 Relocation Program (RELOC.BIN). This program is executed first when the SI3_T225.BIN image is download and started. Its purpose is to relocate the JET.BIN program from the bottom of the shared memory window to the end of the host card memory (See Figure 13).

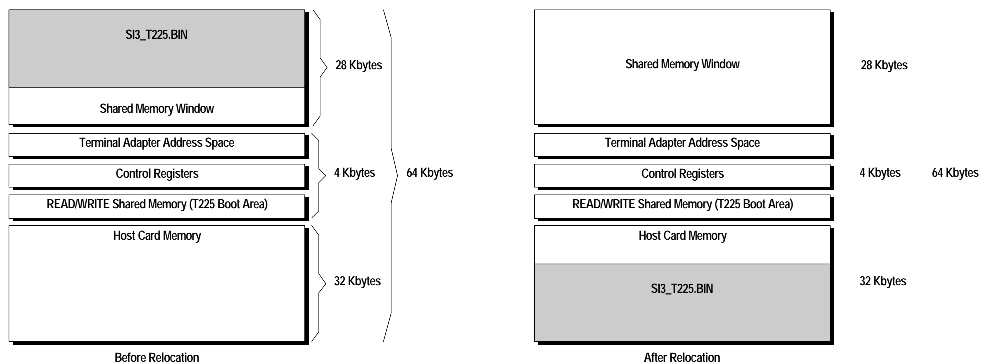


Figure 13 Relocation of SI3_T225.BIN

7.2.2.2 Main Control Program (JET.BIN). The main control program responsible for identifying and initialising terminal adapters on the SI-Bus and constructing and maintaining the shared memory window interface.

7.3 SX+ Basic Program Structure

7.3.1 The download code's main function is to present terminal adapter hardware on the extended SI-Bus to the host PC using the shared memory window structure and host interrupt.

7.3.2 The download code image (SI4_CF.BIN) consists of the binary program

7.3.2.1 The download code (SI4_CF.BIN) resides at location 0 with respect to the start of the Card Memory window. It creates the shared Memory window interface structures which start at 0x1800 with reference to the start of the Card Memory window. The Control registers in turn start at 0x1F000. SI4_CF.BIN is the main control program responsible for identifying and initialising terminal adapters on the SI-Bus and constructing and maintaining the shared memory window interface.

7.4 Initialisation and Identification of Modules

7.4.1 The first action of the download code is to clear the shared memory window area, initialise the Card structure (See 6.2) and test the SI-Bus to determine the modules attached to it. Initialisation is performed by the INIT.C module.

7.4.2 The search assumes that TA modules will be attached by default and attempts to prove that they exist by examining the four Input Port Registers (IP) (one per pair of channels) of each module for a default value.

7.4.2.1 If the default is valid for all four IP registers, then a TA modules is recognised.

7.4.2.2 If the default does not match, then a MTA/SXDC modules is assumed.

7.4.3 As each module and its channels are identified, the devices are initialised to a known state and the structures in the shared memory window constructed. (See 6)

7.4.4 Once initialisation is complete, the cc_init_status field of the Card Structure (See 6.2.2.1) is set indicating whether modules have been detected and initialised, before calling either the 2698.C or CIRRUS.C modules as appropriate.

7.5 TA Modules

7.5.1 The Terminal Adapter (TA) presents the following devices on the T225 bus:

- TA Revision 1 SCC2698 Octal UART [ref6]
- TA Revision 2 TA8 ASIC [ref7]

7.5.2 Both devices are functionally equivalent as far as software programming is concerned.

7.5.3 Referring to Figure 4, the TA UART devices appear to the processor at the following memory locations:

TA1	0x7000
TA2	0x7100
TA3	0x7200
TA4	0x7300

7.5.3.1 Each TA UART consists of eight individual serial channels which appear at the following offsets from the TA base address (See 7.5.3):

Channel 1 0x0000
Channel 2 0x0010
Channel 3 0x0020
Channel 4 0x0030
Channel 5 0x0040
Channel 6 0x0050
Channel 7 0x0060
Channel 8 0x0070

7.5.4 All UART register indexes specified in the SCC2698 reference [ref6] must be multiplied by 2 (with respect to the TA base address) as each register appears as a 16bit word in the SX/SX+ memory map.

7.5.5 The TA modules are managed using the following processes:

7.5.5.1 Main Loop. Contained in 2698.C, this loop polls each of the channels in turn to transmit and receive data and to process any changes to a channels state (hi_hstat field of the Channel Structure, see 6.4.11).

7.5.5.1.1 Any important event (i.e. received data, change in modem status) is registered in an internal interrupt request program variable.

7.5.5.2 Poll Process. Contained in POLL.C, this process executes periodically (throttled by the cc_int_count field of the Card structure) and issues an interrupt to the host PC, if the interrupt request variable has been set.

7.6 MTA Modules

7.6.1 The Modular Terminal Adapter (MTA) presents a CD1400 UXART to the processor.

7.6.2 Referring to Figure 4, the MTA UXART devices appear to the processor at the following memory locations:

MTA1 0x7000
MTA2 0x7100
MTA3 0x7200
MTA4 0x7300

7.6.3 Each MTA consists of 2 CD1400 UXARTs, only one of which can be accessed at any time via a SELECTOR register.

7.6.3.1 The SELECTOR register appears at offset (0x7F * 2) from the MTA base address and is defined as follows:

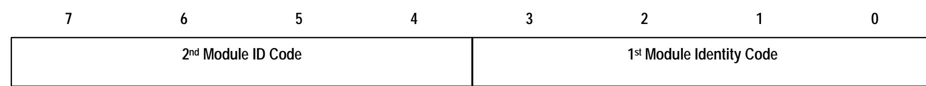
SELECTOR Write:

7	6	5	4	3	2	1	0
Undefined	0 - do nothing 1 - 2nd LED OFF	0 - do nothing 1 - 2nd LED ON	0 - do nothing 1 - 1st LED OFF	0 - do nothing 1 - 1st LED ON	0 - 1st CD1400 1 - 2nd CD1400	Undefined	Undefined

7.6.3.2 Each MTA is composed physically of two hardware modules (corresponding to the two CD1400s) which may be of several different types (RS232 DB25, RS232 RJ45, Parallel etc.).

7.6.3.3 Each hardware module exports an ID which may be read from the SELECTOR register:

SELECTOR Read:



7.6.3.4 The ID of each hardware module is read during initialisation and placed in the mc_mods field of the Module structure. (See 6.3.4.9)

7.6.4 Each CD1400 provides 4 serial channels, only one of which may be accessed at any one time, selected by the CD1400 Channel Access Register (CAR).

7.6.5 All UART register indexes specified in the CD1400 reference [ref4] must be multiplied by 2 (with respect to the MTA base address) as each register appears as a 16bit word in the SX/SX+ memory map.

7.6.6 The MTA modules are managed using the following processes:

7.6.6.1 Main Loop. Contained in CIRRUS.C, this loop polls each of the channels to process any changes to a channels state (hi_hstat field of the Channel Structure, see 6.4.11).

7.6.6.2 Interrupt Process. Contained in ISR.C, this process is executed in response to an interrupt from the CD1400s on the SI-Bus.

7.6.6.2.1 All CD1400 interrupt lines are Ored together into the processor interrupt pin. (See 5.2.7.1)

7.6.6.2.2 The Interrupt Process identifies the CD1400 causing the interrupt, processes receive, transmit and modem requests as necessary, before resetting the Event mechanism.

7.6.6.2.3 Any important event (i.e. received data, change in modem status) is registered in an internal interrupt request program variable.

7.6.6.3 Poll Process. Contained in POLL.C, this process executes periodically (throttled by the cc_int_count field of the Card structure) and issues an interrupt to the host PC, if the interrupt request variable has been set.

7.7 SXDC Modules

7.7.1 The SX Device Concentrator (SXDC) presents an MTA ASIC [ref5] on the processor Bus.

7.7.2 This device is functionally similar to the CD1400 UXART from a software programming point of view and is treated by the download code as an MTA Module (See 7.6).

7.7.3 The SXDC also provides a number of enhancements most of which are supported by the download code:

- higher baud rates (up to 921600)
- deeper receive and transmit FIFOs (up to 32 bytes)
- improved hardware flow control (automatic DTR and RTS handshaking)
- new automatic software flow control (XON/XOFF)

7.8 Host Card Interrupts

7.8.1 Interrupts are generated by the host card to indicate to the host driver/application that an event which requires servicing has occurred.

7.8.2 The rate at which the host card generates interrupts is throttled using the **cc_int_count** field of the card structure (see 6.2.2.3).

7.8.2.1 This field contains the maximum number of interrupts which the board is able to interrupt the host with every second. (i.e. Hz)

7.8.2.2 If not set by the host driver/application, this fields defaults to 100 Hz.

7.8.2.3 It is not recommended that driver/applications modify this field. Future versions of the download code may automatically alter the interrupt frequency to match the highest baud rate selected. This is more of an issue when rates such as 921600 may be selected.

7.8.3 The following internal events will cause the download code to request an interrupt to the host:

7.8.3.1 Any command issued to the **hi_hstat** field of the channels structure. (i.e. HS_LOPEN, HS_MOPEN, HS_CONFIG etc.)

7.8.3.2 CD1400/MTA ASIC generates a receive interrupt in response to received data.

7.8.3.3 CD1400/MTA ASIC generates a transmit interrupt and the transmit buffer level falls below the lower threshold (25% full).

7.8.3.4 CD1400/MTA ASIC generates an exception interrupt in response to a programmed event or a line error (i.e. parity, framing, overrun).

7.8.3.5 CD1400/MTA ASIC generates a modem interrupt in response to a change in the modem signals.

7.9 Source Modules and Include Files

7.9.1 See Ref 9

8. Sample Source Code

8.1 Sample Sources Overview

8.1.1 The sample sources for the SX family of cards are intended to demonstrate the mechanisms and features described in this document. The sources are provided on an "as is" basis and are not directly supported.

8.1.2 The sources have been designed to be modular to aid understanding future development and related functions are contained in the same source module and have similar function names.

i.e. All card related functions are found in CARD.C module and are named CardXxxx.

8.1.3 This section describes the utilities built using the sample sources, how to build them, and a description of each module and the functions it contains.

8.1.4 Note that CSX and SX-C were the names by which the SX+ was known during development.

8.2 Sample Utilities

8.2.1 Two utilities are generated using the sample sources:

- SAMPTTY.EXE
- SAMPIO.EXE

8.2.2 To run the sample utilities copy the executables along with the download code binaries (SI2_Z280.BIN, SI3_t225.BIN and JETCF.BIN) into the same directory on the target PC as the utilities will try to locate these files for downloading to all found SX cards.

8.2.3 SAMPTTY.EXE

8.2.3.1 Simple TTY utility which can be attached to one channel of any cards installed in the system.

8.2.3.2 The help screen (generated by typing "SAMPTTY /h") is as follows:

```
SAMPTTY v1.0.0 SX/SI/XIO Sample TTY Utility
Copyright (c) 1998 Specialix International Ltd.

SAMPTTY{/h} {/?} {chan=x} {<baud>} {<parity>} {<data>} {<stop>}
        {txon} {rxon} {rts} {cts} {local} {verbose} {debug}
        {si2=<filename>} {sx=<filename>} {csx=<filename>}{pci} {isa}
Where:
/h or /?      generates this help screen,
chan=x        number of channel to attach to       default=0
<baud>        baud rate                            default=19200
<parity>      parity (none,odd,even,mark,space)    default=none
<data>        data bits (8,7,6,5)                  default=8
<stop>        stop bits (1,2)                      default=1
txon          transmit XON/XOFF software flow control
rxon          receive XON/XOFF software flow control
rts           receive RTS hardware flow control
```

cts	transmit CTS hardware flow control
local	internal local loopback mode
verbose	displays detailed initialisation report
debug	displays debugging information
si2=<filename>	name of SI2 download file image
sx=<filename>	name of SX download file image
csx=<filename>	name of CSX download file image
pci	search for PCI boards, default=all
isa	search for ISA boards, default=all
{}	indicates optional
<>	indicates value

8.2.3.3 When running, all keypresses are sent to the channel and all incoming data is displayed on the screen.

8.2.3.4 In addition, function keys may be used to perform additional control functions: A help screen showing this functions is displayed by typing “F1”:

Sample TTY Function Keys...

F1 - Show this screen
F2 - Modem signal monitor (typing any other key ends)
F3 - Raise DTR modem output signal
F4 - Lower DTR modem output signal
F5 - Raise RTS modem output signal
F6 - Lower RTS modem output signal
F7 - Set transmit Break signal
F8 - Clear transmit Break signal
F10 - Exit Sample TTY

8.2.4 SAMPIO.EXE

8.2.4.1 This is a broadcast utility which outputs test strings to all specified channels and optionally displays one of the channels input.

8.2.4.2 The help screen (generated by typing “SAMPIO /h”) is as follows:

SAMPIO v1.0.0 Sample I/O Test Utility
Copyright (c) 1998 Specialix International Ltd.

SAMPIO {/h} {/?} <chan1> <chan2-3> {display=<chan>}
{<baud>} {<parity>} {<data>} {<stop>}
{txon} {rxon} {rts} {cts} {local} {verbose} {debug}
{si2=<filename>} {sx=<filename>} {csx=<filename>}{pci} {isa}

Where:

/h or /?	generates this help screen,	
chan1	single channel to access (e.g. 'chan1')	
chan2-3	range of channels to access (e.g.'chan2-3'->'chan2' & 'chan3')	
display=<chan>	displays data received from the specified channel	
<baud>	baud rate	default=19200
<parity>	parity (none,odd,even,mark,space)	default=none
<data>	data bits (8,7,6,5)	default=8
<stop>	stop bits (1,2)	default=1
txon	transmit XON/XOFF software flow control	
rxon	receive XON/XOFF software flow control	

rts	receive RTS hardware flow control
cts	transmit CTS hardware flow control
local	internal local loopback mode
verbose	displays detailed initialisation report
debug	displays debugging information
si2=<filename>	name of SI2 download file image
sx=<filename>	name of SX download file image
csx=<filename>	name of CSX download file image
pci	search for PCI boards, default=all
isa	search for ISA boards, default=all
{}	indicates optional
<>	indicates value

8.3 Building the Sample Utilities

8.3.1 The sample sources are built using the DJGPP compiler and tools.

8.3.1.1 To obtain this compiler see: http://www.delorie.com/djgpp/

8.3.2 To build the sample utilities: (from the main sample code directory)

- run SETENV.BAT to set up the DJGPP environment variables
- run MAKE

8.4 Sample Modules and Functions

8.4.1

- CARD.C
- CHAN.C
- UTILS.C
- ASCII.C
- SYSTEM.C

8.4.2 CARD.C

8.4.2.1 Card related functions prefixed **Card**Xxxx, contains the following functions:

CardFindPCI.	Locates all SX PCI cards in the system
CardFindISA	Locates all SX ISA cards in the system
CardStart	Starts card processor
CardStop	Stops card processor
CardLoad	Loads download code onto card
CardInitWindow	Initialises shared memory window interface
CardReport	Generates a report of the installed cards and modules
CardInitialise	Finds, loads and initialises all cards found in the system
CardRegister	Used to register cards found by CardFindPCI and CardFindISA
CardInstallIsr	Installs an interrupt service routine for a card
CardIsr	General interrupt service routine
CardDeInstallIsr	Deinstalls interrupt service routine for a card
CardEnableIrq	Enables card interrupts
CardDisableIrq	Disables card interrupts
CardNumber	Returns card number from the system channel number
CardPointer	Returns CARD structure pointer from card number

8.4.3 CHAN.C

8.4.3.1 Channel related functions prefixed **Chan**Xxxx, contains the following functions:

ChanOpen	Opens a channel for read/write/config
ChanClose	Closes a channel
ChanWrite	Writes a block of data to a channel
ChanRead	Read a block of data from a channel
ChanConfig	Configures a channel
ChanSetModemSignals	Sets channel modem output signals
ChanSetTxBreak	Sets a channel break signal
ChanGetModemSignals	Returns the channels current modem signals
ChanGetEvents	Returns the channels current event status
ChanGetErrors	Returns the channels error counts
ChanNumber	Returns the channel number from an SXCHANNEL structure pointer

8.4.4 UTILS.C

8.4.4.1 General pupose functions used for debugging:

DbgPrintf	Same as printf, if debugging is enabled
DbgOn	Enables debugging
DbgOff	Disables debugging
Dump	Displays a dump of memory (as bytes)
DumpDword	Displays a dump of memory as dwords

8.4.5 ASCII.C

8.4.5.1 Functions for generating ASCII string representations of SX structure fields:

ascii_hi_hstat	ASCII string for SXCHANNEL.hi_hstat
ascii_card_phase	ASCII string for PHASE portion of CARD.type
ascii_card_bus	ASCII string for BUS portion of CARD.type
ascii_mc_chip	ASCII string for SXMODULE.mc_chip
ascii_cd1400_rev	ASCII string for SXMODULE.mc_revx
ascii_mc_mods	ASCII string for MTA SXMODULE.mc_mods
ascii_sxdc_mc_mods	ASCII string for SXDC SXMODULE.mc_mods
ascii_sxdc_rev	ASCII string for SXDC hardware revision

8.4.6 SYSTEM.C

8.4.6.1 System dependant functions for managing memory, interrupts, timing and PCI access:

Sys_MemAlloc	Allocate memory
Sys_MemFree	Free memory
Sys_GetPointer	Convert physical address to virtual pointer
Sys_FreePointer	Free previously allocated virtual pointer
Sys_RegisterService	Register interrupt service routine
Sys_DeRegisterService	Remove interrupt service routine
Sys_DelaymS	Delay for a given number of milliseconds
Sys_TimeStampmS	Return a timestamp in milliseconds
Sys_ElapsedmS	Return period elapsed since last timestamp in mS
Pci_ReadConfig	Read a PCI configuration header for given bus/dev/fn

Pci_WriteConfig	Write a PCI configuration header for given bus/dev/fn
Sys_Initialise	Perform any system related initialisation

8.4.7 Sample Include Files

- CARD.H
- UTILS.H
- SYSTEM.H

8.4.8 CARD.H

8.4.8.1 Prototypes for all CardXxxx, ChanXxxx, ascii_xxx functions.

8.4.8.2 CARD and CONFIG structures used by the sample code.

8.4.9 UTILS.H

8.4.9.1 Prototypes for the general purpose utility functions.

8.4.10 SYSTEM.H

8.4.10.1 Prototypes for the general system dependant functions.

8.5 Download Include Files

8.5.1 The sample code also includes the latest version of the download code binaries for the T225 and Z280 based SX/SI/XIO cards as well as the MCF5206e Coldfire based SX+ card.

8.5.2 When later versions of the download code are released the following files should be obtained and copied to the "dcode" subdirectory of the sample code directory:

SI2_Z280.BIN	Binary version of Z280 download code
SI2_Z280.C	'C' source version of Z280 download code (for embedding)
SI3_T225.BIN	Binary version of T225 download code
SI3_T225.C	'C' source version of T225 download code (for embedding)
SI4_CF.BIN	Binary version of Coldfire download code
SI4_CF.C	'C' source version of Coldfire download code (for embedding)
SXWINDOW.H	Shared Memory Window Interface structures/definitions
SXBOARDS.H	SX/SI/XIO boards hardware definitions and ID codes

8.5.3 It is recommended that software is developed using the SXWINDOW.H and SXBOARDS.H files so that possible futures features to the download code can be quickly added merely by updating the header files.

8.5.4 SXWINDOW.H

8.5.4.1 Contains structures and definitions used to describe the shared memory window software interface:

SXCARD	Card structure
SXMODULE	Module structure
SXCHANNEL	Channel structure

8.5.5 SXBOARDS.H

8.5.5.1 Contains definitions for Ids, hardware features and registers relating to the SX/SI/XIO board hardware interfaces.

9. References

[ref1] Title: SLXOS Technical Reference Manual
A Guide To Writing Device Drivers for the Specialix SI & XIO
Document: SP-US402
Issue: 3rd Edition
Date: March 1994

[ref2] Title: SI/XIO (JET) PCI Host Hardware Functional Specification
Document: 6200060
Issue: 2.1
Date: 19th August 1997

[ref3] Title: SI XIO (JET) ISA Host Hardware Functional Specification
Document: 6200026
Issue: 1.8
Date: 21st August 1997

[ref4] Title: Cirrus Logic CL-CD1400 Data Sheet
Document:
Issue:
Date: March 1992

[ref5] Title: MTA ASIC Hardware Functional Specification
Document: 6200012
Issue: 1.6
Date: 23rd October 1995

[ref6] Title: Philips Semiconductors
Enhanced Octal Asynchronous Receiver/Transmitter (Octal UART)
SCC2698B Product Specification
Document: Stored as 6200080
Issue: 1.0
Date: 1st May 1995

[ref7] Title: TA8 ASIC Hardware Functional Specification
Document: 6200049
Issue: 1.4
Date: 28th April 1994

[ref8] Title: 8 Port SX Device Concentrator (SXDC) Hardware Functional Specification
Document: 6200077
Issue: 1.1
Date: 4th February 1998

[ref9] Title: Firmware for SX Coldfire and T225 Host Card Software Functional Specification
Document: 6210077
Issue: 0.1
Date: 27th January 1999