VS1000 Programmer's Guide

VSMPG "VLSI Solution Audio Decoder"

Project Code:	Support.VS1000
Project Name:	VSMPG

Revision History

Rev.	**Date**	**Author**	**Description**
0.1	2007-03-23	PKP	Preliminary version
0.11	2007-04-16	PKP	Minor adjustments
0.12	2007-06-28	PKP	Additions for version 1.33 of developer tools

Who needs to read this document

This document describes the programming interface, register map and and integrated peripherals of the VS1000. It's primarily meant for those that wish to add to the functionality of the ROM code in VS1000 or design completely new software for the chip.

If you use the USB...

The example "Changing the USB descriptors" should be read by all vendors that have USB functionality in their end-products. Although the ROM software is functional as is, all such vendors should change the USB descriptors to identify the vendor and product ID's correctly.

Additionally, all vendors that ship devices conforming to the USB Mass Storage Class specification should change the USB descriptors and create a unique serial number for each device. Instructions on how to do this are given in the example.

VS1000B/C

VS1000B is an updated version of VS1000A, which has many small internal fixes and some additions that remove some of the restricitions in VS1000A. It's mainly compatible with VS1000A, but code needs to be recompiled for VS1000B. This guide is written for VS1000A and does not yet take advantage of the extra features in VS1000B. VS1000C is another production test version of VS1000B. All new code should be written for VS1000B.

Table of Contents

1 Introducing the VS1000

VS1000 is a complete DSP system-on-chip that can be used to implement a multitude of applications such as a single-chip Ogg Vorbis player. VS1000 contains a high-performance low-power DSP core VS_DSP^4, NAND-FLASH interface, Full Speed USB port, general purpose I/O pins, SPI, UART, as well as a high-quality variable-sample-rate stereo DAC, and an earphone amplifier and a common voltage buffer.

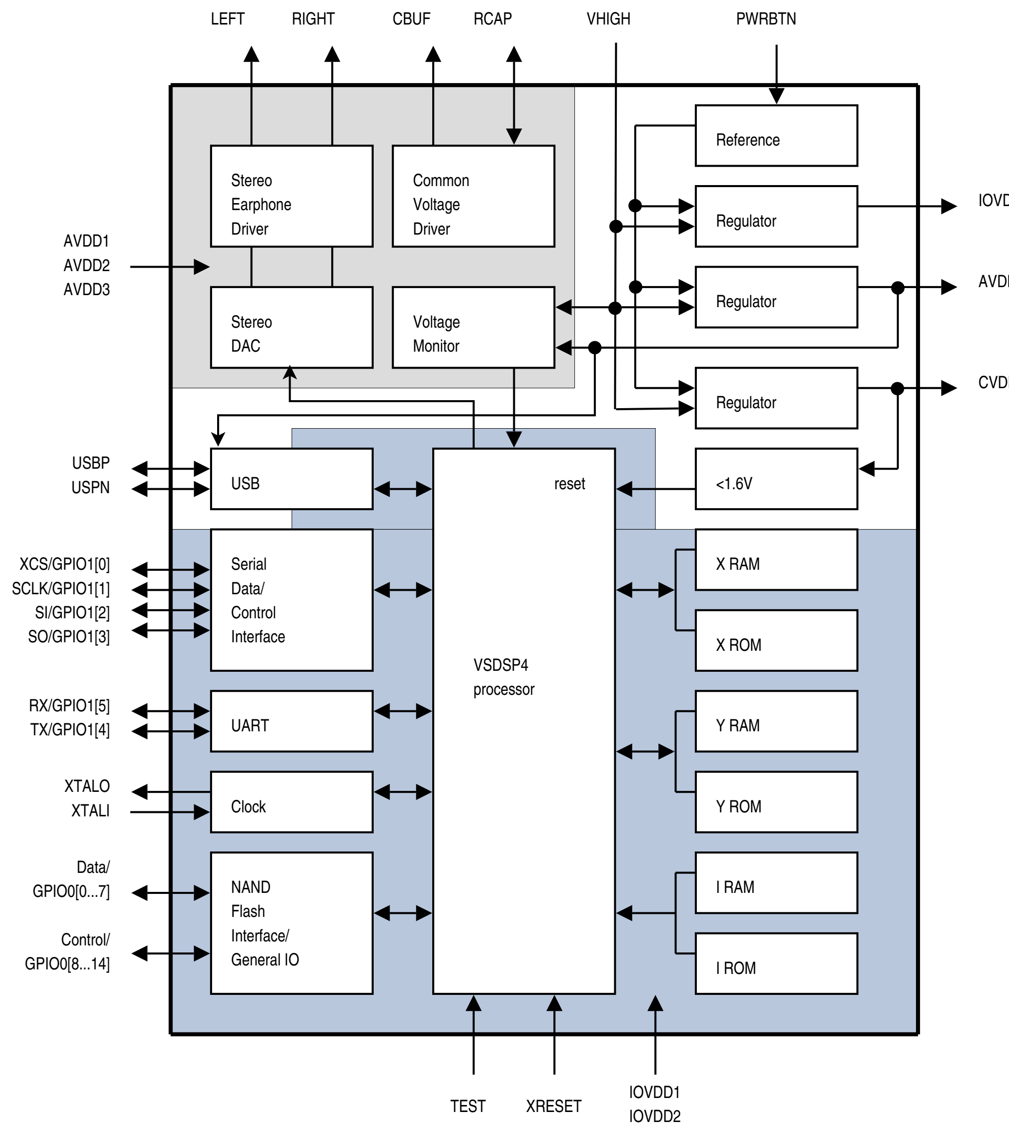

Figure 1.1: VS1000A Block Diagram

1.1 VS_DSP Basics

At VS1000's core is the VS_DSP4 signal processor. It has a 16-bit Harward architecture with three separate 16-bit address spaces: X and Y space for data and I space for instructions (running code). All of these spaces have both ROM and RAM. In addition, X and or Y spaces can occupy special function registers for peripheral devices.

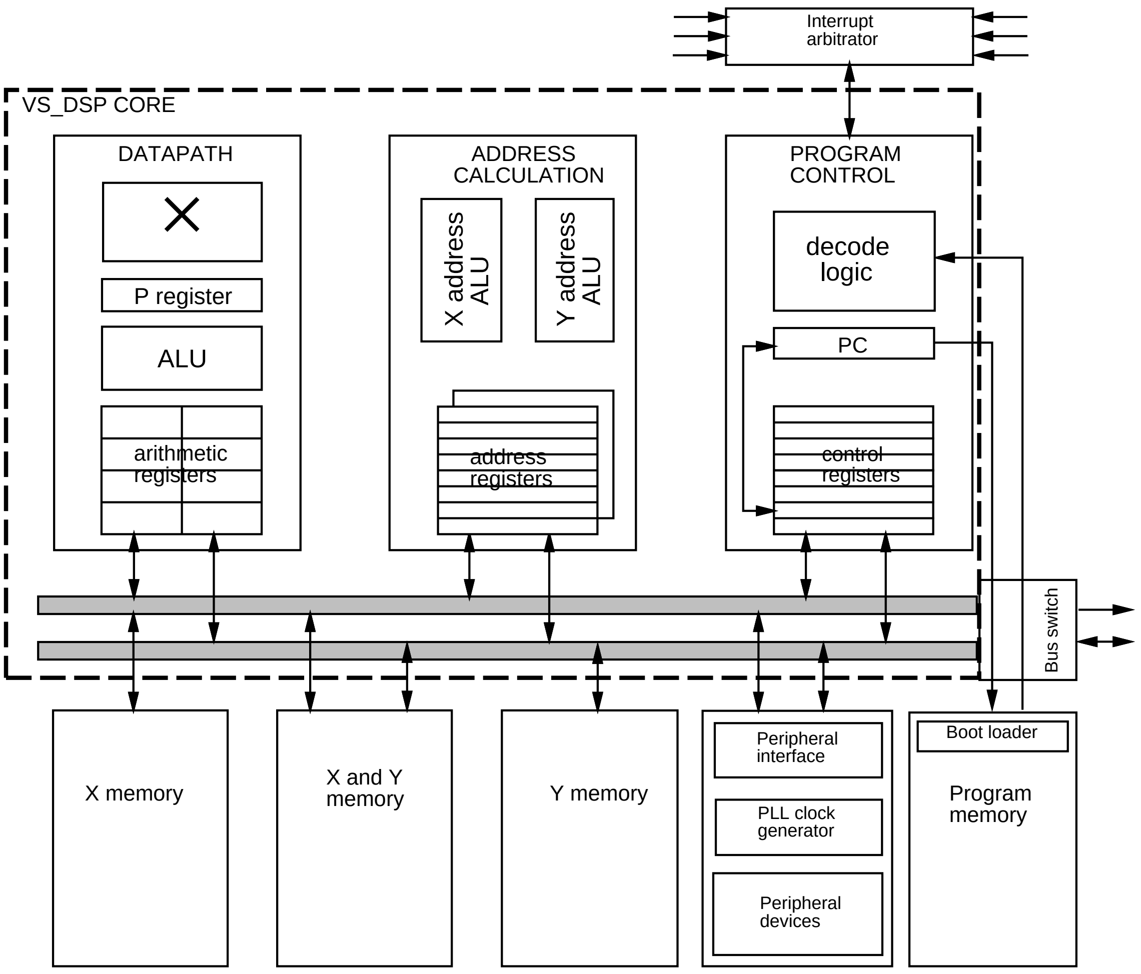

Figure 1.2: VS_DSP General Architecture

Most of the features of the VS_DSP processor can be accessed by using standard C language, without any specific VS_DSP knowledge. But if you need to develop really powerful DSP algorithms, use the 40-bit datapath, control the pipeline and take the maximum out of the parallel X, Y and I buses, you need to study the VS_DSP architecture and use assembly language.

Currently the VS_DSP4 architecture manual is not freely available from VLSI, but VS_DSP2 User's Manual is distributed in the VSKIT command line toolset for VS_DSP2, which is downloadable from the VLSI website (vskit116.zip).

Since VS_DSP4 is downward compatible with VS_DSP2 in all other respects except index register postmodification modes 010 and 011 (modulo +2 and modulo -2), which have different meanings in VS_DSP4, the VS_DSP2 User's Manual is the best free resource for learning about the DSP core operation of VS1000.

1.2 VS1000 Integrated Peripherals

VS1000 contains several integrated peripherals. They are controlled by memory-mapped special function registers. From the programmer's point of view this means reading and writing special memory locations. The peripheral registers in VS1000 are located in the X address space.

VS1000A chip has the following integrated peripherals:

- 21 GPIO pins multiplexed with peripherals, each capable of generating an interrupt
- SPI port with master/slave operation and programmable Frame Sync
- UART port with programmable bit rate and framing error detection
- USB port with 12 Mbit/s signaling rate and 4 KiB of buffer memory
- Digital-to-Analog converter and integrated earphone driver
- Byte-wide Bus / Nand Flash controller with fast 32-byte buffer and ECC calculation
- 2 32-bit timers with shared master clock divider
- Interrupt controller, 11 interrupt sources
- 3 programmable linear regulators for generating analog, I/O and core voltages
- Internal oscillator for external crystal, can also use external oscillator
- Integrated Clock Generator with PLL and clock multiplier and low-speed modes
- Watchdog timer

The VS1000A has 76 KiB of program ROM and 8 KiB of program RAM. While the latter might seem like a small amount, note that the ROM code contains many useful routines, interfaces and tables the RAM code can access. Many internal functions can be replaced or augmented by hooking a handler vector of a ROM routine.

The amount of data RAM available varies depending on the application. If Vorbis playing is not used, it can be over 50 KiB. For programs that do play Vorbis files, at least 2652 bytes can be used when Vorbis files are playing.

The complete peripheral documentation is in its own chapter.

1.3 VS1000 Register Map and Frequently Used Tables

VS1000A Peripheral Register Map

Address	**Register**	**Function**
0xC000	SCI_SYSTEM	System Controller control
0xC001	SCI_STATUS	System Controller control and status flags
0xC010	GPIO0_MODE	GPIO(0)/Peripheral(1) function for port 0 pins
0xC011	GPIO1_MODE	GPIO(0)/Peripheral(1) function for port 1 pins
0xC012	DAC_VOL	Digital-to-Analog Converter Volume
0xC013	FREQCTLL	Interpolator Frequency low part
0xC014	FREQCTLH	Interpolator Frequency high part
0xC015	DAC_LEFT	DAC Left Channel
0xC016	DAC_RIGHT	DAC Right Channel
0xC020	WDOG_CONFIG	Watchdog Config
0xC021	WDOG_RESET	Watchdog Reset
0xC022	WDOG_DUMMY	Watchdog dummy register
0xC028	UART_STATUS	Serial Port Status
0xC029	UART_DATA	Serial Port Data byte
0xC02A	UART_DATAH	Serial Port Data byte shifted 8 bits left
0xC02B	UART_DIV	Serial Port baudrate generator divider
0xC030	TIMER_CONFIG	Timer 0 and 1 Configuration
0xC031	TIMER_ENABLE	Timer 0 and 1 Enable/Disable
0xC034	TIMER_T0L	Low 16 bits of Timer 0 reload value
0xC035	TIMER_T0H	High 16 bits of Timer 0 reload value
0xC036	TIMER_T0CNTL	Low 16 bits of Timer 0 current value
0xC037	TIMER_T0CNTH	High 16 bits of Timer 0 current value
0xC038	TIMER_T1L	Low 16 bits of Timer 1 reload value
0xC039	TIMER_T1H	High 16 bits of Timer 1 reload value
0xC03A	TIMER_T1CNTL	Low 16 bits of Timer 1 current value
0xC03B	TIMER_T1CNTH	High 16 bits of Timer 1 current value
0xC040	GPIO0_DDR	Port 0 Data Direction ("1"=output)
0xC041	GPIO0_ODATA	Port 0 Output Data
0xC042	GPIO0_IDATA	Port 0 Input Data (pin state)
0xC043	GPIO0_INT_FALL	Falling Edge Interrupt Enable
0xC044	GPIO0_INT_RISE	Rising Edge Interrupt Enable
0xC045	GPIO0_INT_PEND	Interrupt Pending
0xC046	GPIO0_SET_MASK	Set output bits high
0xC047	GPIO0_CLEAR_MASK	Set output bits low
0xC048	GPIO0_BIT_CONF	Bit router engine 0 and 1 configuration
0xC049	GPIO0_BIT_ENG0	Bit router engine 0 data register
0xC04A	GPIO0_BIT_ENG1	Bit router engine 1 data register

VS1000A Peripheral Register Map (continued)		
Address	**Register**	**Function**
0xC050	GPIO1_DDR	Port 1 Data Direction ("1"=output)
0xC051	GPIO1_ODATA	Port 1 Output Data
0xC052	GPIO1_IDATA	Port 1 Input Data (pin state)
0xC053	GPIO1_INT_FALL	Falling Edge Interrupt Enable
0xC054	GPIO1_INT_RISE	Rising Edge Interrupt Enable
0xC055	GPIO1_INT_PEND	Interrupt Pending
0xC056	GPIO1_SET_MASK	Set output bits high
0xC057	GPIO1_CLEAR_MASK	Set output bits low
0xC058	GPIO1_BIT_CONF	Bit router engine 0 and 1 configuration
0xC059	GPIO1_BIT_ENG0	Bit router engine 0 data register
0xC05A	GPIO1_BIT_ENG1	Bit router engine 1 data register
0xC060	NFLSH_CTRL	Byte-wide Bus (Nand Flash) Controller Control
0xC061	NFLSH_LPL	Calculated Line Parity for 512-byte block
0xC062	NFLSH_CP_LPH	Calculated Column Parity for 512-byte block
0xC063	NFLSH_DATA	Buffer Data read/write register
0xC064	NFLSH_NFIF	Buffer-to-Physical Interface Control
0xC065	NFLSH_DSPIF	Buffer-to-DSP Interface Control
0xC066	NFLSH_ECC_CNT	Error Correction Code counter
0xC068	SPI0_CONFIG	Serial Peripheral Interface Configuration
0xC069	SPI0_CLKCONFIG	SPI Clock Configuration
0xC06A	SPI0_STATUS	SPI Status
0xC06B	SPI0_DATA	SPI Data read/write register
0xC06C	SPI0_FSYNC	Frame Sync output bit image
0xC070	INT_ENABLEL	Low Priority Interrupt Enable
0xC072	INT_ENABLEH	High Priority Interrupt Enable
0xC074	INT_ORIGIN	Interrupt Request Status
0xC076	INT_VECTOR	Last generated vector
0xC077	INT_ENCOUNT	Interrupt disable level counter
0xC078	INT_GLOB_DIS	Disable interrupts (increase ENCOUNT)
0xC079	INT_GLOB_EN	Enable interrupts (decrease ENCOUNT)
0xC080	USB_CONFIG	USB Device Config
0xC081	USB_CONTROL	USB Device Control
0xC081	USB_STATUS	USB Device Status
0xC082	USB_RDPTR	Receive buffer pointer (PC → Device)
0xC083	USB_WRPTR	Transmit buffer pointer (Device → PC)
0xC088	USB_EP_SEND0	EP0IN Transmittable Packet Info
0xC089	USB_EP_SEND1	EP1IN Transmittable Packet Info
0xC08A	USB_EP_SEND2	EP2IN Transmittable Packet Info
0xC08B	USB_EP_SEND3	EP3IN Transmittable Packet Info
0xC090	USB_EP_ST0	Flags for endpoints EP0IN and EP0OUT
0xC091	USB_EP_ST1	Flags for endpoints EP1IN and EP1OUT
0xC092	USB_EP_ST2	Flags for endpoints EP2IN and EP2OUT
0xC093	USB_EP_ST3	Flags for endpoints EP3IN and EP3OUT

VS1000A Interrupt Sources		
Name	**Vector**	**Source**
INTV_DAC	0	Digital to Analog Converter
INTV_SPI	1	Serial Peripheral Interface
INTV_USB	2	Universal Serial Bus
INTV_NFLSH	3	Byte-wide Bus (Nand Flash) Controller
INTV_TX	4	UART Transmit
INTV_RX	5	UART Receive
INTV_TIM0	6	Timer 0 underflow
INTV_TIM1	7	Timer 1 underflow
INTV_REGU	8	Input Voltage Monitor
INTV_GPIO0	9	I/O Pin Controller 0
INTV_GPIO1	10	I/O Pin Controller 1

VS1000A I/O Controller 0 pins and peripheral functions			
GPIO	**Ident**	LQFP Pin	**Function**
GPIO0[0]	NFDIO0	2	Nand-flash IO0 / General-purpose IO Port 0, bit 0
GPIO0[1]	NFDIO1	3	Nand-flash IO1 / General-purpose IO Port 0, bit 1
GPIO0[2]	NFDIO2	4	Nand-flash IO2 / General-purpose IO Port 0, bit 2
GPIO0[3]	NFDIO3	5	Nand-flash IO3 / General-purpose IO Port 0, bit 3
GPIO0[4]	NFDIO4	9	Nand-flash IO4 / General-purpose IO Port 0, bit 4
GPIO0[5]	NFDIO5	10	Nand-flash IO5 / General-purpose IO Port 0, bit 5
GPIO0[6]	NFDIO6	11	Nand-flash IO6 / General-purpose IO Port 0, bit 6
GPIO0[7]	NFDIO7	12	Nand-flash IO7 / General-purpose IO Port 0, bit 7
GPIO0[8]	NFRDY	13	Nand-flash READY / General-purpose IO Port 0, bit 8
GPIO0[9]	NFRD	14	Nand-flash RD / General-purpose IO Port 0, bit 9
GPIO0[10]	NFCE	15	Nand-flash CE / General-purpose IO Port 0, bit 10
GPIO0[11]	NFWR	20	Nand-flash WR / General-purpose IO Port 0, bit 11
GPIO0[12]	NFCLE	16	Nand-flash CLE / General-purpose IO Port 0, bit 12
GPIO0[13]	NFALE	17	Nand-flash ALE / General-purpose IO Port 0, bit 13
GPIO0[14]	CS2	21	General-purpose IO Port 0, bit 14

VS1000A I/O Controller 1 pins and peripheral functions			
GPIO	**Ident**	LQFP Pin	**Function**
GPIO1[0]	XCS	22	SPI XCS / General-Purpose I/O Port 1, bit 0
GPIO1[1]	SCLK	23	SPI CLK / General-Purpose I/O Port 1, bit 1
GPIO1[2]	SI	24	SPI MISO / General-Purpose I/O Port 1, bit 2
GPIO1[3]	SO	25	SPI MOSI / General-Purpose I/O Port 1, bit 3
GPIO1[4]	TX	26	UART TX / General-Purpose I/O Port 1, bit 4
GPIO1[5]	RX	27	UART RX / General-Purpose I/O Port 1, bit 5

VS1000A Handler Vectors (Services)			
Address	**Vector Name**	**Default Handler**	**Remark**
0x0000	IdleHook	UserInterfaceIdleHook	CPU idle
0x0002	InitFileSystem	FatInitFileSystem	Init storage
0x0004	OpenFile	FatOpenFile	Open file
0x0006	ReadFile	FatReadFile	Read file
0x0008	Seek	FatSeek	Set file position
0x000a	Tell	FatTell	Get file position
0x000c	ReadDiskSector	MapperReadDiskSector	Read 512 bytes
0x000e	StereoCopy	OldStereoCopy	Output samples
0x0015	Sine Test	SinTest	Sine test
0x0016	Memory Test	MemTest	Memory test 1
0x0017	Memory Test	MemTests	Memory tests
0x0018	SetRate	RealSetRate	Set sample rate
0x001a	PowerOff	RealPowerOff	Close and shutdown
0x001c	PlayCurrentFile	RealPlayCurrentFile	Start playing file
0x001e	USBHandler	RealUSBHandler	USB Task

VS1000A Handler Vectors (Interrupt Controller)			
Address	**Vector Name**	**Default Handler**	**Remark**
0x0020	DAC Interrupt	dac_int	Update sample
0x0021	SPI Interrupt	_int (Default Null Handler)	
0x0022	USB Interrupt	_int (Default Null Handler)	
0x0023	Nand Flash Interrupt	_int (Default Null Handler)	
0x0024	TX Interrupt	_int (Default Null Handler)	
0x0025	RX Interrupt	rx_int	ROM Monitor
0x0026	Timer 0 Interrupt	tim0_int	System timer
0x0027	Timer 1 Interrupt	_int (Default Null Handler)	
0x0028	Power Interrupt	_int (Default Null Handler)	
0x0029	GPIO0 Interrupt	_int (Default Null Handler)	
0x002a	GPIO1 Interrupt	_int (Default Null Handler)	

VS1000A Handler Vectors (Services)			
Address	**Vector Name**	**Default**	**Remark**
0x002c	MSCPacketFromPC	RealMSCPacketFromPC	MSC cmd or data
0x002e	DecodeSetupPacket	RealDecodeSetupPacket	Control endpoint
0x0030	ScsiTaskHandler	RealScsiTaskHandler	Disk task
0x0032	LoadCheck	RealLoadCheck	Clock adjust
0x0034	UnsupportedFile	DefUnsupportedFile	Unknown format

2 Software Tools

Here is a list of the software tools that are necessary to compile and run the examples of this programming guide. A more complete documentation of the software tools can be found in the "Tools Manual", available from VLSI. These command line tools are available for UNIX and Windows. In addition to these files we recommend using GNU Make to automatize the compilation process, but you can also compile by typing the command lines separately in a shell or "MS-DOS Prompt", or with the help of a suitable batch file. Some GUI's can also be configured to run the compiler and linker as external applications.

The tools package can be requested by writing an email to VLSI Audio Solution Support at the e-mail address `mp3@vlsi.fi` . Please give basic information of your name, position and if you are working for a company (please give company name and department), or if you are a student (please give name of educational institution) or hobbyist etc.

vcc

The VLSI C Compiler. Creates a COFF object file from "C" language source file.

Example:
```
vcc -P130 -O -fsmall-code -I include -o program.o program.c
```

vslink

The linker. Creates a binary program file from multiple COFF object files.

Example:
```
vslink -k -m mem_user -L lib -lc -o program.bin lib/c-spi.o lib/rom1000a.o
program.o
```

vs3emu

The ROM monitor interface. Loads and runs binary program files using RS-232 cable between PC and VS10xx. Also provides standard input/output and file system for debugging C code.

Example:
```
vs3emu -chip vs1000 -s 115200 -l program.bin e.cmd
```

coff2spiboot

Creates bootable EEPROM image from a binary program file.

Example:
```
coff2spiboot -x 0x50 program.bin eeprom.img
```

coff2nandboot

Creates a nand flash compatible boot record file from a binary program file.

Example:
```
coff2nandboot -t 3 -b 8 -s 19 -w 50 -x 0x50 led.bin nand.rec
```

makenandimage

Creates a prommable binary nand flash image from a nand flash compatible boot record file.

Example:
```
makenandimage nand.rec NANDFLSH.IMG
```

3 Examples

3.1 Hello, World!

The first example of writing code for the VS1000A is the traditional "Hello, World!" example, which is compiled and linked. Then the RS-232 ROM monitor interface (vs3emu) is used to load and execute the code.

The contents of the file `hello.c` is:

```
 /*  hello.c :  A Hello World example.  */

#include <stdio.h>

//  main() is the program entry point.  It is entered via a vector,
//  which is statically linked to address 0x0050 in module c.o

void main(void) {
  puts("Hello, World!");
}
```

Compiling

The "hello.c" file is compiled using vcc with a command line such as:

```
vcc -P130 -O -fsmall-code -I include -o hello.o hello.c
```

This creates a coff object file hello.o. The parameteres that were passed to vcc are:

-P130	Treats warning 130 ("can't find prototype") as an error.
-O	Optimize
-fsmall-code	Use 16-bit code model (uses libc16 libraries)
-o hello.o	Output file is hello.o
-I include	subdirectory "`include`" contains include files
hello.c	input file

Linking

Next the hello.o object is linked using the VS1000 memory map, VS1000A ROM content addresses and the relevant VSDSP run-time libraries using a command such as:

```
vslink -k -m mem_user -o hello.bin -L lib -lc lib/c-spi.o lib/rom1000a.o hello.o
```

This produces a loadable object file hello.bin using the parameters:

-k	keep relocation information
-m mem_user	use memory areas specified in file `mem_user`
-o hello.bin	output file is `hello.bin`
-L lib	libraries can be found in subdirectory "`lib`"
-lc	use library `libc.a` (in the -L directory)
lib/c-spi.o	the vsemu and SPI boot compatible C startup module (in subdirectory `lib`). It calls `main()` and returns to ROM code to a point after initializations and SPI boot but before Nand Flash init+boot.
lib/rom1000a.o	address information of the ROM code (in subdirectory `lib`)
hello.o	user compiled module

Loading

There are many ways to load runnable code to VS1000A chips. Code can be loaded automatically during boot-up time from an SPI EEPROM or a NAND flash.

During program development it's usually easiest to load the code using an RS-232 ("COM port") emulator interface, which connects to the RX and TX pins of the VS1000. [1]

The PC side interface is invoked with:

```
vs3emu -chip vs1000 -s 115200 -l hello.bin
```

which instructs the vs3emu interface to use the "vs1000" communication method and default (COM1) port with line speed 115200 bit/s.

The emulator contacts the VS1000 by sending a special character to the COM port. This is handled by the UART receive interrupt on the VS1000. If the VS1000 is running with a 12 MHz crystal, interrupts are enabled and the core is running, it responds with:

```
 VSEMU 2.1 (c)1995-2006 VLSI Solution Oy
Clock 11999 kHz
Using serial port 1, Serial input speed seems to be 115200
COM speed 115200
Waiting for a connection to the board...
Caused interrupt
Chip version "1000"
Stack pointer 0x19e0, bpTable 0x7c0f
User program entry address 0x7398
hello.bin:  includes optional header, 4 sections, 441 symbols
Section 1:  code        page:0 start:80 size:1 relocs:1 fixed
Section 2:  const_x     page:1 start:8096 size:14 relocs:0
Section 3:  main        page:0 start:81 size:14 relocs:2
Section 4:  VS_stdiolib  page:0 start:95 size:50 relocs:13
>
```

[1]This is easiest with a VS1000 Developer Board, but even the VS1000 Demonstration Board could be used in this fashion by building a suitable RS-232 interface board. It would require connecting a MAX3232 or equivalent buffer chip to the "RX" and "TX" pads on the Demonstration Board PCB. Power for the MAX3232 could be taken from the JP1 expansion header.

Next the executing address is set to be 0x0050 (statically linked loading vector for main()) by command `g 0x50` and executed by command `e`. On the screen it should look like:

```
 > g 0x50
> e
Hello, World!
```

This final stage can be automated by writing the commands `g 0x50` and `e` to file `e.cmd` and calling the emulator with the command line

```
vs3emu -chip vs1000 -s 115200 -l hello.bin e.cmd
```

The emulator can be exited by pressing Ctrl-C.

Note

If your board has boot code in the Nand Flash, the Nand Flash boot code runs after main() exits.

Input and Output

This example uses the vs3emu interface to handle C standard I/O (`stdin, stdout`). With it it's possible to write messages to the user and read input from the PC keyboard. Also it's possible to open, read and write files in the PC. The library contains the elementary functions necessary for input and output. In this example, the library function `puts()`, which outputs a line of text and a linefeed to `stdout`, was used.

Since the memory capacity of the chip is limited, the more advanced and memory consuming input/output functions such as printf should not be used. When you need to print out values of variables, it's recomended to use a smaller special function for it. As an example, here is a small function that outputs the value of a 16-bit unsigned integer as a hexadecimal value:

```
#include <stdio.h>
#include <vstypes.h>

__y const char hex[] = "0123456789abcdef";
void puthex(u_int16 a) {
  char tmp[8];
  tmp[0] = hex[(a>>12) & 15];  tmp[1] = hex[(a>>8) & 15];
  tmp[2] = hex[(a>>4)  & 15];  tmp[3] = hex[(a>>0) & 15];
  tmp[4] = ' ';                tmp[5] = '\0';
  fputs(tmp, stdout);
}
```

Also note that if you use `puts` (or any file input/output) in your code, a connection with vs3emu is required. You should carefully remove any such code before porting the code to be loaded via another method than vs3emu such as a boot flash or eeprom. This could be done by surrounding the I/O code with `#ifdef DEBUG` and `#endif` pre-processor directives.

3.2 Making the LEDs blink

The example code below will blink the two LEDs that are connected to VS1000A's SI and SO pins on the Developer Board and the Demonstration Board. Controlling the pins directly requires switching the pin modes from Peripheral control to General Purpose IO control and setting their Output Enable bits to "1".

```
#include <vs1000.h>

/// Busy wait i hundreths of second at 12 MHz clock
auto void BusyWaitHundreths(u_int16 i) {
  while(i--){
    BusyWait10(); // Rom function, busy loop 10ms at 12MHz
  }
}

void main(void) {

  PERIP(GPIO1_MODE) = 0x30; /* UART=peripheral(1) , SPI=GPIO(0) */
  PERIP(GPIO1_DDR) = 0x0c;  /* SI and SO pins (GPIO1[3:2]) are output(1) */

  while(1){
    PERIP(GPIO1_ODATA) = 0x04; /* GPIO1[2] (LQFP pin 24) = 1 */
    BusyWaitHundreths(50);
    PERIP(GPIO1_ODATA) = 0x08; /* GPIO1[3] (LQFP pin 25) = 1 */
    BusyWaitHundreths(50);
  }
}
```

The SPI port pins and UART port pins are controlled by the same I/O controller, I/O controller 1. When disabling peripheral control of the SPI pins, the UART pins (RX, TX) must remain under peripheral control. Otherwise, the connection with vs3emu is lost.

For reference, here are the GPIO1 pin mappings of VS1000A:

VS1000A I/O Controller 1 pins and peripheral functions			
GPIO	**Ident**	LQFP Pin	**Function**
GPIO1[0]	XCS	22	SPI XCS / General-Purpose I/O Port 1, bit 0
GPIO1[1]	SCLK	23	SPI CLK / General-Purpose I/O Port 1, bit 1
GPIO1[2]	SI	24	SPI MISO / General-Purpose I/O Port 1, bit 2
GPIO1[3]	SO	25	SPI MOSI / General-Purpose I/O Port 1, bit 3
GPIO1[4]	TX	26	UART TX / General-Purpose I/O Port 1, bit 4
GPIO1[5]	RX	27	UART RX / General-Purpose I/O Port 1, bit 5

3.3 Adjusting the Player User Interface

The ROM code implements a Vorbis player with a default user interface that has 6 buttons:

- Power/Play/Pause
- Previous/Rewind
- Next/Fast Forward
- Volume -
- Volume +
- EarSpeaker (spatial processing) setting change

In addition to the 6-button interface the ROM contains alternative default key mappings for a 5-button and 4-button user interfaces.

If these are not sufficient, there are two alternatives:

- Create a custom key → event mapping
- Take full control of the player

The ROM function `void KeyEventHandler(enum keyEvent event)` can handle 12 pre-defined player control events:

VS1000A Pre-defined Player Control Events		
Value	Event	**Function**
0	ke_null	Do nothing
1	ke_previous	Play Previous song
2	ke_next	Play Next song
3	ke_rewind	Rewind
4	ke_forward	Fast Forward
5	ke_volumeUp	Volume Up
6	ke_volumeDown	Volume Down
7	ke_earSpeaker	Switch EarSpeaker processing (4 settings)
8	ke_earSpeakerToggle	Toggle EarSpeaker processing (2 settings)
9	ke_randomToggle	Random Play on/off
10	ke_randomToggleNewSong	Play random song
11	ke_pauseToggle	Pause on/off
12	ke_powerOff,	Close and power down

A `KeyMapping` structure controls the relationship between key-presses, long key-presses and events. The structure is an array of pairs

```
struct KeyMapping {
    u_int16 key;         // Key Mask
    enum keyEvent event; // Event
};
```

The following program demonstrates changing the key mapping:

```
// Example on how to change the key mapping of the user interface

#include <vs1000.h>
#include <player.h>

// Define key masks for the buttons on the PCB. This order is
// of the Demonstration Board, leftmost button is "KEY_A"
#define KEY_A 0x0004
#define KEY_B 0x0008
#define KEY_C 0x0001
#define KEY_D 0x0002
#define KEY_E 0x0010

// Define custom key mapping
const struct KeyMapping myKeyMap[] = {
  {KEY_A,                          ke_volumeUp  }, // Key A: Volume step up
  {KEY_A | KEY_LONG_PRESS,         ke_volumeUp  }, // Key A: Volume up continuous
  {KEY_B,                          ke_volumeDown}, // Key B: Volume step dn
  {KEY_B | KEY_LONG_PRESS,         ke_volumeDown}, // Key B: Volume dn continuous
  {KEY_C,                          ke_previous  }, // Key C: Previous song
  {KEY_D,                          ke_next      }, // Key D: Next song
  {KEY_E | KEY_A | KEY_LONG_PRESS,   ke_rewind    }, // Key E with Key A: rewind
  {KEY_E | KEY_B | KEY_LONG_PRESS,   ke_forward   }, // Key E with Key B: fast forward
  {KEY_C | KEY_D | KEY_LONG_ONESHOT, ke_powerOff  }, // Only one event after long press
  {0, ke_null} // End of key mappings
};

// Load own key mapping
void main(){
  currentKeyMap = myKeyMap; // Use own key mapping

  // Note that if there is boot record in NAND, it's run after
  // this point, if this code is run from the emulator
}
```

The KeyEventHandler can also be called directly. For instance if you wish to advance to the next song, you can call

```
KeyEventHandler(ke_next);
```

from your source code. In most cases it takes less code space than changing the `struct player` directly.

The tools package contains further examples on how to adjust the user interface, use the embedded LCD font etc.

3.4 Hooking custom storage controller

Hooks are software jump vectors, that are linked into fixed positions in the VS1000A RAM. Their function is essentially the same as for instance the interrupt vector of a 80x86 processor. For instance, when the player is playing music, it reads a disk sector (512 bytes) of data by calling a function ReadDiskSector(u_int16 *buffer, u_int32 sector). For this call, the linker generates a call to a fixed address 0x000c. In that address (which is in RAM) is a jump instruction to the start address of the ROM function RealReadDiskSector(), which retrieves the data from a logical NAND Flash mapper interface.

By replacing the jump location of the ReadDiskSector() hook vector, it is easy to replace the storage device, which contains the files the player plays. Only the service that delivers a sectorful of data from a storage device is changed while rest of the ROM functionality remains the same.

The image below demonstrates the disk data flow of VS1000:

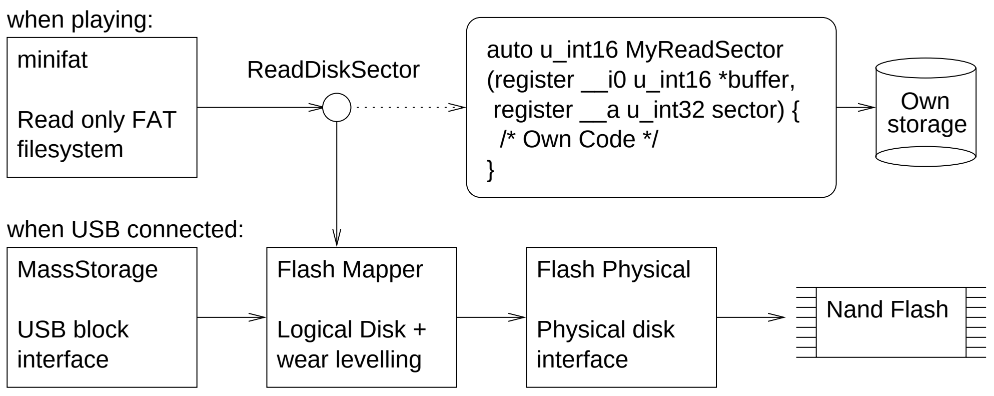

Figure 3.1: Disk Data Flow

Below is an example of hookable disk read function that uses a previously declared `EEReadBlock()` function to read 512 bytes to `*buffer` and returns 0 signifying no error:

```
auto u_int16 MyReadDiskSector(register __i0 u_int16 *buffer,
                              register __a  u_int32  sector) {
  EEReadBlock(sector+FAT_START_SECTOR, buffer);
  return 0;
}
```

This can then be hooked to the ReadDiskSector hook by calling

```
   SetHookFunction((u_int16)ReadDiskSector, MyReadDiskSector);
```

in `main()` or some other convenient function.

The above method is most convenient for preprogrammed storage devices. If you need write access to your own storage device, you need to write control code for it yourself, e.g. for downloading a disk image over a serial port etc.

ReadDiskSector is for reading only

As the name suggests, the ReadDiskSector() hook is meant only for reading data. This limits its usage to the "player" mode only (when the VS1000A is in player mode, it does not write to the logical disk).

If you want to attach your own device to the USB bus as a mass storage device, you need to write a mapper interface that has functions for reading and writing+erasing 512-byte sectors. Then you need to write a function that publishes the interface with name `map`, initializes the USB handler (probably by calling `InitUSB(USB_MASS_STORAGE)`) and then calls `UsbHandler()` in a busy loop until the USB is detached.

The complete example code is below. It uses 253 words of program RAM out of the 1968 words available for plugins.

```
// storage.c :  Plug-in for playing from intel "S33" serial flash eeprom.
// For this example, a QH25F640S33B8 chip is connected to SI, SO, SCLK, XCS.

#include <stdlib.h>
#include <vs1000.h>


#define SPI_EEPROM_COMMAND_READ_STATUS_REGISTER  0x05
#define SPI_EEPROM_COMMAND_READ  0x03


//macro to set SPI to MASTER; 8BIT; FSYNC Idle => xCS high
#define SPI_MASTER_8BIT_CSHI   PERIP(SPI0_CONFIG) = \
        SPI_CF_MASTER | SPI_CF_DLEN8 | SPI_CF_FSIDLE1

//macro to set SPI to MASTER; 8BIT; FSYNC not Idle => xCS low
#define SPI_MASTER_8BIT_CSLO   PERIP(SPI0_CONFIG) = \
        SPI_CF_MASTER | SPI_CF_DLEN8 | SPI_CF_FSIDLE0

//macro to set SPI to MASTER; 16BIT; FSYNC not Idle => xCS low
#define SPI_MASTER_16BIT_CSLO  PERIP(SPI0_CONFIG) = \
        SPI_CF_MASTER | SPI_CF_DLEN16 | SPI_CF_FSIDLE0



void InitSpi() {
  SPI_MASTER_8BIT_CSHI;
  PERIP(SPI0_FSYNC) = 0;        // Frame Sync is used as an active low xCS
  PERIP(SPI0_CLKCONFIG) = SPI_CC_CLKDIV * (1-1);   // Spi clock divider = 1
  PERIP(GPIO1_MODE) |= 0x1f;    // Set SPI pins to be peripheral controlled
}


void EESingleCycleCommand(u_int16 cmd){
  SPI_MASTER_8BIT_CSHI;
  SPI_MASTER_8BIT_CSLO;
  SpiSendReceive(cmd);
  SPI_MASTER_8BIT_CSHI;
}


/// Wait for not_busy (status[0] = 0) and return status
u_int16 EEWaitGetStatus(void) {
  u_int16 status;
  SPI_MASTER_8BIT_CSHI;
  SPI_MASTER_8BIT_CSLO;
  SpiSendReceive(SPI_EEPROM_COMMAND_READ_STATUS_REGISTER);
  while ((status = SpiSendReceive(0)) & 0x01)
    ; //Wait until ready
  SPI_MASTER_8BIT_CSHI;
  return status;
}
```

```
/// Read a block from EEPROM
/// \param blockn number of 512-byte sector 0..32767
/// \param dptr pointer to data block
u_int16 EEReadBlock(u_int16 blockn, u_int16 *dptr) {
  EEWaitGetStatus();                     // Wait until EEPROM is not busy
  SPI_MASTER_8BIT_CSLO;                  // Bring xCS low
  SpiSendReceive(SPI_EEPROM_COMMAND_READ);
  SpiSendReceive(blockn>>7);             // Address[23:16] = blockn[14:7]
  SpiSendReceive((blockn<<1)&0xff);      // Address[15:8]  = blockn[6:0]0
  SpiSendReceive(0);                     // Address[7:0]   = 00000000
  SPI_MASTER_16BIT_CSLO;                 // Switch to 16-bit mode
  { int n;
    for (n=0; n<256; n++){
      *dptr++ = SpiSendReceive(0);       // Receive Data
    }
  }
  SPI_MASTER_8BIT_CSHI;                  // Bring xCS back to high
  return 0;
}


// Disk image is prommed to EEPROM at sector 0x80 onwards, leaving
// the first 64 kilobytes (1 erasable block) free for boot code
#define FAT_START_SECTOR 0x80


// This function will replace ReadDiskSector() functionality
auto u_int16 MyReadDiskSector(register __i0 u_int16 *buffer,
                              register __a u_int32 sector) {

  PERIP(GPIO1_MODE) |= 0x1f;    // Set SPI pins to be peripheral controlled
  EEReadBlock(sector+FAT_START_SECTOR, buffer);

  return 0;
}


// Initialize SPI and hook in own disk read function.
// This example plays ogg files from a FAT image that has been
// previously written to a serial EEPROM.

void main(void) {
  InitSpi();

  // Hook in own disk sector read function
  SetHookFunction((u_int16)ReadDiskSector, MyReadDiskSector);

} // Return to ROM code.  Player will now play from EEPROM
```

3.5 Setting your own USB descriptors

Each USB device has a Vendor ID and a Product ID, which are 16-bit numbers that the operating system uses for determining which device driver to load for the device. Additionally most USB devices have a vendor name and model name strings that the operating system can display to the user. All USB string descriptors are 16-bit Unicode strings (UTF-16).

VS1000A's ROM code holds VLSI's Vendor ID and Product ID. For prototyping you can use an unused Vendor ID and Product ID, but when you ship products to customers, you must use your own Vendor ID and Product ID. A Vendor ID can be obtained from the USB Implementers Forum, Inc.'s web site, `http://www.usb.org` .

To comply with USB Mass Storage Specification, each device that is shipped out to customers should have a unique serial number in the USB descriptors. Windows uses this serial number e.g. for storing device parameters in the system registry.

VS1000A's ROM is written so that it's easy to change these descriptors without having to touch the rest of the USB code. This example shows how you can change the Device Descriptor, which holds the Vendor ID and Product ID, and the Vendor/Model/SerialNumber string descriptors.

`USB.descriptorTable[6]` holds pointers to the descriptors. They are overwritten by the ROM code in various locations, but a system hook vector called `DecodeSetupPacket` can be used to set pointers to your own descriptors each time a SETUP packet is received from the PC. This way you can be sure that VS1000A always responds with the updated descriptors.

Descriptor data format

Mostly because the USB has its roots in the 8-bit oriented PC (80x86) architecture, all USB traffic is transmitted byte by byte. When values that have more than 8 bits, such as 16-bit integers or 32-bit integers, are transmitted, they are transmitted in the little-endian ("Little End First") format, where the least significant (last) byte of a multi-byte value is sent first.

VS_DSP, however, is a natively 16-bit architecture that only handles 16-bit values. Thus all data in VS_DSP must be stored as signed or unsigned 16 (or 32) bit values. To maintain USB compatibility, care must be taken to transmit descriptors in the correct byte order. In practice this means that descriptors should be stored in tables of byte-swapped 16-bit unsigned integers as in the example below.

The serial number is a string of (at least) 12 characters from set {"0123456789ABCDEF"}. All strings are stored in 16-bit Unicode format. The example code creates a new serial number string descriptor `mySerialNumberStr`. The last 8 characters are generated in the `main()` function from `u_int32 mySerialNumber`, which should be unique for each device. You could generate it from e.g. the serial number of the storage memory your product has. The first 4 characters ("1234" in the descriptor) could be fixed for a specific program version etc.

```
// usbdesc.c :  Example for changing USB descriptors
// We will hook DecodeSetupPacket so that it sets our string descriptors
// each time a Setup packet (USB device/class request) is sent.

#include <vs1000.h>
#include <usb.h>

#define VENDOR_NAME_LENGTH 6
const u_int16 myVendorNameStr[] = {
  ((VENDOR_NAME_LENGTH * 2 + 2) << 8) | 0x03,
  'M' << 8,
  'y' << 8,
  'C' << 8,
  'o' << 8,
  'r' << 8,
  'p' << 8
};

#define MODEL_NAME_LENGTH 6
const u_int16 myModelNameStr[] = {
  ((MODEL_NAME_LENGTH * 2 + 2) << 8) | 0x03,
  'G' << 8,
  'a' << 8,
  'd' << 8,
  'g' << 8,
  'e' << 8,
  't' << 8
};

#define SERIAL_NUMBER_LENGTH 12
u_int16 mySerialNumberStr[] = {
  ((SERIAL_NUMBER_LENGTH * 2 + 2) << 8) | 0x03,
  '1' << 8, // You can
  '2' << 8, // put any
  '3' << 8, // numbers you
  '4' << 8, // like here (over the '1' '2' '3' and '4')
  0x3000, 0x3000, 0x3000, 0x3000, // Last 8 digits of serial
  0x3000, 0x3000, 0x3000, 0x3000  // number will be calculated here
};

// This is the new Device Descriptor.  See the USB specification!
// Note that since VS_DSP is 16-bit Big-Endian processor,
// tables MUST be given as byte-swapped 16-bit tables for USB compatibility!
// This device descriptor template is ok for mass storage devices.
const u_int16  myDeviceDescriptor [] = {
  0x1201, 0x1001, 0x0000, 0x0040,
  0x3412,   // byte-swapped Vendor ID (0x1234) Get own from usb.org!
  0x4523,   // byte-swapped Product ID (0x2345)
  0x5634,   // byte-swapped Device ID (0x3456)
  0x0102, 0x0301
};
```

```
// When a USB setup packet is received, install our descriptors
// and then proceed to the ROM function RealDecodeSetupPacket.
void MyDecodeSetupPacket(void){
  USB.descriptorTable[DT_VENDOR] = myVendorNameStr;
  USB.descriptorTable[DT_MODEL]  = myModelNameStr;
  USB.descriptorTable[DT_SERIAL] = mySerialNumberStr;
  USB.descriptorTable[DT_DEVICE] = myDeviceDescriptor;
  RealDecodeSetupPacket();
}

const u_int16  bHexChar16[] = { // swapped Unicode hex characters
  0x3000, 0x3100, 0x3200, 0x3300, 0x3400, 0x3500, 0x3600, 0x3700,
  0x3800, 0x3900, 0x4100, 0x4200, 0x4300, 0x4400, 0x4400, 0x4500
};

void main(void) {
  u_int16 i;
  u_int32 mySerialNumber = 0x1234abcd;  // Unique serial number

  // Put unique serial number to serial number descriptor
  for (i=5; i<13; i++){
    mySerialNumberStr[i]=bHexChar16[mySerialNumber>>28];
    mySerialNumber <<= 4;
  }

  // Hook in function that will load new descriptors to USB struct
  SetHookFunction((u_int16)DecodeSetupPacket, MyDecodeSetupPacket);

} // Return to ROM code.
```

3.6 Booting from SPI EEPROM

VS1000A supports loading boot-up code from an SPI EEPROM such as the 25LC640. The ROM code checks the state of XCS pin during boot-up. If XCS is high, the code attempts to read a boot record from the EEPROM using the SI, SO, SCLK and XCS pins. In addition to the 16-bit addressing of SPI eeproms such as the 25LC640, the ROM also supports 24-bit addressing of some larger EEPROMS (possibly up to 16 megabytes).

A program that is to be loaded using the SPI EEPROM must be linked with `c-spi.o` object module. The `c-spi.o` module can also be used with running the code from vs3emu, but not from the nand flash.

The `coff2spiboot` tool can be used to create a bootable EEPROM image from the linker output file with a command such as:

```
  coff2spiboot -x 0x50 led.bin eeprom.img
```

This reads the previously compiled program `led.bin` and creates a binary eeprom image `eeprom.img`, which can be programmed to an SPI EEPROM with an EEPROM programmer.

A valid boot record starts with identifier 0x564C5349 ('V','L','S','I') and contains blocks of binary data that are to be stored at specified addresses. A boot record that is loaded via the SPI bus must have an execution command as the last block. Description of the block format is in the datasheet, if it should be needed for some special purpose.

Using a VS1000 Developer Board as an eeprommer

Also a VS1000 Developer Board can be used to program the SPI EEPROM, using the vs3emu file interface. The next pages contain an example program that reads the `eeprom.img` file and writes it to a 25LC640 EEPROM. The promming routine is compiled normally to a binary program `prommer.bin`. It can be run with vs3emu with a command such as:

```
  vs3emu -chip vs1000 -s 115200 -l prommer.bin e.cmd
```

If the file `eeprom.img` is found on the local directory, the contents is programmed to the EEPROM and you should see output such as

```
   25LC640 EEPROM promming routine for VS1000A
   Trying to open eeprom.img
   Programming...
   Sector 0000 ...
   Reading first 2 words of EEPROM: 564c 5349  ("VLSI"), which is a valid VLSI boot
id.
   Done.
```

```
// VS1000A EEPROM Writer Program
// Reads eeprom.img file from PC via vs3emu cable and programs it to EEPROM.

#define MY_IDENT "25LC640 EEPROM promming routine for VS1000A"

#include <stdio.h>
#include <stdlib.h>
#include <vs1000.h>
#include <minifat.h>

__y const char hex[] = "0123456789abcdef";
void puthex(u_int16 a) {
  char tmp[8];
  tmp[0] = hex[(a>>12)&15];  tmp[1] = hex[(a>>8)&15];
  tmp[2] = hex[(a>>4)&15];   tmp[3] = hex[(a>>0)&15];
  tmp[4] = ' ';              tmp[5] = '\0';
  fputs(tmp, stdout);
}

#define SPI_EEPROM_COMMAND_WRITE_ENABLE  0x06
#define SPI_EEPROM_COMMAND_WRITE_DISABLE  0x04
#define SPI_EEPROM_COMMAND_READ_STATUS_REGISTER  0x05
#define SPI_EEPROM_COMMAND_WRITE_STATUS_REGISTER  0x01
#define SPI_EEPROM_COMMAND_READ  0x03
#define SPI_EEPROM_COMMAND_WRITE 0x02

//macro to set SPI to MASTER; 8BIT; FSYNC Idle => xCS high
#define SPI_MASTER_8BIT_CSHI   PERIP(SPI0_CONFIG) = \
        SPI_CF_MASTER | SPI_CF_DLEN8 | SPI_CF_FSIDLE1

//macro to set SPI to MASTER; 8BIT; FSYNC not Idle => xCS low
#define SPI_MASTER_8BIT_CSLO   PERIP(SPI0_CONFIG) = \
        SPI_CF_MASTER | SPI_CF_DLEN8 | SPI_CF_FSIDLE0

//macro to set SPI to MASTER; 16BIT; FSYNC not Idle => xCS low
#define SPI_MASTER_16BIT_CSLO  PERIP(SPI0_CONFIG) = \
        SPI_CF_MASTER | SPI_CF_DLEN16 | SPI_CF_FSIDLE0

void SingleCycleCommand(u_int16 cmd){
  SPI_MASTER_8BIT_CSHI;
  SpiDelay(0);
  SPI_MASTER_8BIT_CSLO;
  SpiSendReceive(cmd);
  SPI_MASTER_8BIT_CSHI;
  SpiDelay(0);
}

/// Wait for not_busy (status[0] = 0) and return status
u_int16 SpiWaitStatus(void) {
  u_int16 status;
  SPI_MASTER_8BIT_CSHI;
  SpiDelay(0);
  SPI_MASTER_8BIT_CSLO;
  SpiSendReceive(SPI_EEPROM_COMMAND_READ_STATUS_REGISTER);
  while ((status = SpiSendReceive(0xff)) & 0x01){
    SpiDelay(0);
  }
  SPI_MASTER_8BIT_CSHI;
  return status;
}

void SpiWriteBlock(u_int16 blockn, u_int16 *dptr) {
  u_int16 i;
  u_int16 addr = blockn*512;

  for (i=0; i<32; i++){
    SingleCycleCommand(SPI_EEPROM_COMMAND_WRITE_ENABLE);
    SPI_MASTER_8BIT_CSLO;
    SpiSendReceive(SPI_EEPROM_COMMAND_WRITE);
    SPI_MASTER_16BIT_CSLO;
    SpiSendReceive(addr);
    {
      u_int16 j;
      for (j=0; j<16; j++){ //Write 16 words (32 bytes)
        SpiSendReceive(*dptr++);
      }
    }
    SPI_MASTER_8BIT_CSHI;
    SpiWaitStatus();
    addr+=32;
  }
}


u_int16 SpiReadBlock(u_int16 blockn, u_int16 *dptr) {
  SpiWaitStatus();
  SPI_MASTER_8BIT_CSLO;
  SpiSendReceive(SPI_EEPROM_COMMAND_READ);
  SpiSendReceive((blockn<<1)&0xff);     // Address[15:8]  = blockn[6:0]0
  SpiSendReceive(0);                    // Address[7:0]   = 00000000
  SPI_MASTER_16BIT_CSLO;
  {
    u_int16 i;
    for (i=0; i<256; i++){
      *dptr++ = SpiSendReceive(0);
    }
  }
  SPI_MASTER_8BIT_CSHI;
  return 0;
}


// This routine programs the EEPROM.
// The minifat module has a memory buffer of 512 bytes (minifatBuffer)
// that is used here as temporary memory.
// The routine does not verify the data that is written, but after
// programming, the eeprom start is checked for a VLSI boot id.
```

```
void main(void) {

  FILE *fp;

  SPI_MASTER_8BIT_CSHI;
  PERIP(SPI0_FSYNC) = 0;
  PERIP(SPI0_CLKCONFIG) = SPI_CC_CLKDIV * (12-1);
  PERIP(GPIO1_MODE) |= 0x1f; /* enable SPI pins */
  PERIP(INT_ENABLEL) &= ~INTF_RX; //Disable UART RX interrupt

  puts("");
  puts(MY_IDENT);
  puts("Trying to open eeprom.img");

  if (fp = fopen ("eeprom.img", "rb")){ // Open a file in the PC
    u_int16 len;
    u_int16 sectorNumber=0;

    puts("Programming...");
    while ((len=fread(minifatBuffer,1,256,fp))){
      fputs("Sector ",stdout); puthex(sectorNumber); puts("...");
      SpiWriteBlock(sectorNumber, minifatBuffer);
      sectorNumber++;
    }
    fclose(fp);  // Programming complete.

    minifatBuffer[0]=0;
    fputs("Reading first 2 words of EEPROM: ",stdout);
    SpiReadBlock(0,minifatBuffer);

    puthex(minifatBuffer[0]);
    puthex(minifatBuffer[1]);

    fputs(" (\"",stdout);
    putchar(minifatBuffer[0]>>8); putchar(minifatBuffer[0]&0xff);
    putchar(minifatBuffer[1]>>8); putchar(minifatBuffer[1]&0xff);

    if ((minifatBuffer[0]==0x564c) && (minifatBuffer[1]==0x5349)){
      puts("\"), which is a valid VLSI boot id.");
    } else {
      puts("\"), which is NOT a valid VLSI boot id!");
    }
    puts("Done.");

  }else{
    puts("File not found\n");
  }

  PERIP(INT_ENABLEL) |= INTF_RX; //Re-enable UART RX interrupt
  while(1)
    ; //Stop here
}
```

3.7 Booting from NAND FLASH

If a nand flash chip is connected to the byte-wide bus interface of VS1000, it can also be used for booting the VS1000. VS1000A supports natively most single-level cell, single-chip-select NAND flashes such as the NAND128W3A2 from ST (small page) or K9F2G08U0M from Samsung (large page).

3.7.1 Nand Flash startup sequence and structure

The nand flash boot is attempted after EEPROM boot. First the I/O voltages are set according to the input state of GPIO0[7] pin. Then VS1000A attempts to read the first block of 512 bytes of the nand flash with 8 different access methods, using the nand flash interface with only CS1 chip select. The access methods cover small and large page flashes with 4, 5 or 6 address bytes.

Because different types of nand flash chips differ in the access methodology, using a nand flash is somewhat more complicated than using an eeprom. To ensure proper operation, a nand flash chip must be programmed with a valid VLSI ID record in the beginning of block 0. VS1000A looks for the ID record and adjusts the nand access parameters according to the ID record information.

If the VLSI boot id 'V"L"S"I' (0x564C5349) is successfully read in the beginning of block 0, the ID record is considered valid. The next words of the ID record specify the overall size, erasable block size, number of address bytes, block size and speed grade of the nand flash chip in question in the format specified in the VS1000 datasheet.

The rest of the 512-byte block can be programmed with additional boot code. It can contain a small patch code that fits in the sector itself, or a chain loader for loading a larger user program.

The first erasable area of a nand flash chip is reserved for boot data. The filesystem that contains the songs to be played and is visible to the PC as a USB disk starts at a further offset after the boot area and for security reasons is separate from the boot area.

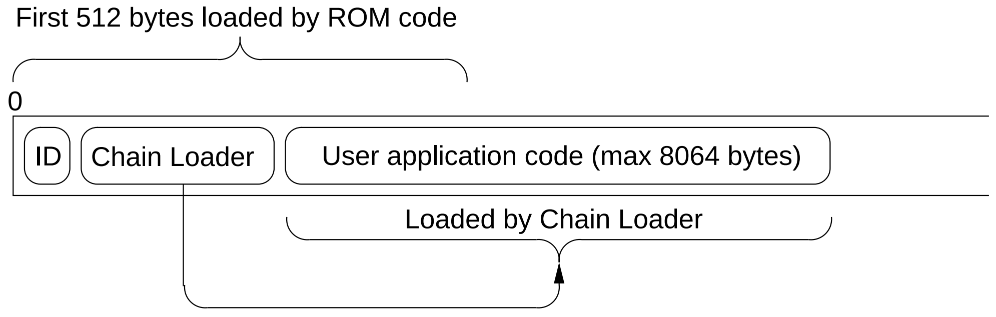
First 512 bytes loaded by ROM code
0
ID
Chain Loader
User application code (max 8064 bytes)
Loaded by Chain Loader

Figure 3.2: Structure of the beginning of a Nand Flash in VS1000A

3.7.2 Preparing a nand flash image

A program that is to be loaded using the nand flash must be linked with `c-nand.o` with a command line such as:

```
vslink -k -m mem_user -o led.bin -L lib -lc lib/c-nand.o lib/rom1000a.o led.o
```

The binary program `led.bin` must then be converted into a nand boot record using the `coff2nandboot` program with a command line such as:

```
  coff2nandboot -t 3 -b 8 -s 19 -w 50 -x 0x50 led.bin nand.rec
```

The program `coff2nandboot` creates a nand boot record starting with a VLSI ID record. The parameters `-t 3 -b 8 -s 19 -w 50` specify that the target nand flash chip used

- is of Type 3 (Large Page, 5-byte address)
- has an erasable block size of $2^8 \times 512$ bytes (128 KiB)
- has an overall size of $2^{19} \times 512$ bytes (256 MiB)
- needs 50 ns wait states

The parameters `-x 0x50 led.bin nand.rec` instruct that

- executable code starts at address 0x0050
- linked program image is in `led.bin`
- boot record should be written to `nand.rec`

Output such as the following can be expected from `coff2nandboot`:

```
NandType:  3 Large-Page 5-byte addr, 128kB blocks, 256MB flash
I: 0x0050-0x0086 In:  222, out:  222
In:  222, out:  228
```

The above parameters are ok for the K9F2G08, which is installed in some of the Demonstration Boards shipped by VLSI. Others have NAND128W and for those a suitable command line is `coff2nandboot -t 0 -b 5 -s 15 -x 0x50 led.bin nand.rec`

There is one final step before a programmable nand flash image is obtained: the tool `makenandimage` inserts a VS1000A compatible chain loader to nand.rec and outputs a binary boot image `nand.img`.

```
  makenandimage nand.rec NANDFLSH.IMG
```

```
makenandimage:  inserting chain loader for 'nand.rec'.
```

The resulting binary file `NANDFLSH.IMG` can be prommed to the beginning of a nand flash with a nand flash programmer.

3.7.3 Using the VS1000 Demostration/Developer Board as a nand flash writer

Because it would be troublesome to remove a nand flash chip that is soldered to a PCB for programming, the VS1000 contains a number of ways to update the flash contents. The nand flash contents can be updated by

- programming the nand flash off-pcb using a prommer
- running a flasher program via the vs3emu emulator interface (requires RS-232)
- running a flasher program via an SPI EEPROM
- running a flasher program via the VS1000 USB mass storage backdoor

The last option is most convenient for players that don't have RS-232 port, such as the VS1000 Demonstration Board. When the VS1000 is switched to USB Mass Storage mode by attaching the USB cable when GPIO0[6] is low, it creates a logical drive that is presented to the USB host as a removable disk.

A special thing happens when the ROM software can't detect a nand flash chip (by reading the VLSI boot ID as explained earlier). In that case, the software creates a RAM disk of a few kilobytes. This can be detected by the disk being empty and having a size of only about 16 kilobytes. (The RAM disk also has the identifier signature "`VLSIFATDISK`" but that is normally not shown by Windows.)

This feature can be used for initial programming of the nand flash since at the first boot-up of a new VS1000 device with an empty nand flash, the VLSI ID is not yet programmed into the nand flash and thus the RAM disk appears. Later on, when the nand flash is programmed and its contents need to be updated, the nand flash detection can be prevented by pulling CS1 low when powering up the VS1000. In the Demonstration Board this can be done by shorting TP2 and CS1 pads on the Developer Board PCB. When connected to the PC, the RAM disk appears and the short should be removed. The user can now copy files to the RAM disk using Windows/Unix etc.

A special file named `VS1000_A.RUN` can now be copied to the RAM disk. When the USB cable is removed, **without turning off power**, the VS1000A loads and runs a boot record from that file. The maximum length of the boot record is 512 bytes.

For updating the flash contents, `VS1000_A.RUN` should contain a flasher program, that reads another file named `NANDFLSH.IMG` from the RAM disk and writes its contents to the beginning of the nand flash. When the VS1000 boots up the next time, with CS1 pulled high, it uses and boots up from the nand flash with the updated software.

The software tools package for VS1000 contains the above VS1000_A.RUN file. Its source code is shown on the next page as an example of more complicated (and powerful) VS1000 programming that uses the integrated ROM code library.

```
// Program for flashing first sector of a compatible Nand Flash chip
// from file NANDFLSH.IMG on the RAMDISK.
// Since this program is run from file VS1000_A.RUN in the ramdisk, *map already
// points to an existing ramdisk, so OpenFile() etc work from the ramdisk.

#include <vs1000.h>
#include <minifat.h>
#include <vsNand.h>
extern __y u_int16 mallocAreaY[]; /* for ramdisk */
extern u_int16 mallocAreaX[];     /* for ramboot */
extern struct FsNandPhys fsNandPhys;
extern struct FsPhysical *ph;

void main(void) {
  register int j = 0;
  ph = &fsNandPhys.p; // Physical disk is nand flash handler in ROM
  mallocAreaY[29] = 0x3220; // Force disk image to be FAT12

  if (InitFileSystem() == 0) { // Reinitialize file system in FAT12 mode

    static const u_int32 bootFiles[] = { FAT_MKID('I','M','G'), 0 };
    minifatInfo.supportedSuffixes = bootFiles; // Only read .IMG files

    if (OpenFile(0) < 0) { // Open first .IMG file on ramdisk
      j = ReadFile(mallocAreaX, 0, 2*0x1000) / 2;
      if (j==0) goto fail; // Could not read from the file
    } else goto fail; // OpenFile() did not find any .IMG file from the ramdisk

    // File is now read to mallocAreaX and j contains its length.
    ((struct FsNandPhys *)ph)->nandType = mallocAreaX[2]; //nandType from imgfile
    ((struct FsNandPhys *)ph)->waitns = 200; //Set 200 ns wait states

    if (ph->Erase(ph, 0)){ // Call ROM routine to erase flash
      goto fail; // In case of erase failure
    }

    // Call ROM routine to write sector, goto fail if chip reports write error
    if (ph->Write(ph, 0, (j+255)/256, mallocAreaX, NULL) == 0) goto fail;

    /* Programming done, do special LED blink */
    while(1){
      PERIP(GPIO1_ODATA) = 0x04; /* GPIO1[2] (LQFP pin 24) = 1 */
      for (j=0; j<10; j++) BusyWait10();
      PERIP(GPIO1_ODATA) = 0x08; /* GPIO1[3] (LQFP pin 25) = 1 */
      for (j=0; j<100; j++) BusyWait10();
    } // Continue the blinking forever
  }

fail:
  PERIP(GPIO1_ODATA) = 0x08; // in fail condition constantly light LED 2
  while(1)
    ;
}
```

3.8 Additional examples

Version 1.33 of VS1000 Developer Toolkit (vskit133.zip) is launched alongside with the launching of VS1000B, an updated version of VS1000A. It has many additional example files to study. These are mainly for VS1000B.

The code that is to be run with VS1000B or VS1000C must be compiled from the vs1000bc directory. Code for VS1000A must be compiled from the vs1000a directory. For information on how to compile the various binaries for VS1000B or VS1000C, refer to the command batch file `BUILD.BAT` in directory vskit133/vs1000bc and README.TXT file in directory vskit133.

3.8.1 playloop.c

`playloop.c` is an example on how to take direct control of the player, e.g. replace the main playing loop. This example programs the player so that each file is repeated continuously until user selects "next" or "previous" song (by pressing the appropriate button).

3.8.2 display.c

`display.c` is the code, which is preprogrammed to the nand flash of the VS1000 Developer Board (the one that has the oled display). It contains examples about how to get information about the currently playing file and update a graphical display based on that information. It's also an example of how to access the internal font ROM of vs1000.

3.8.3 execdemo1.c and execdemo2.c

These files contain an example of an `Exec()` function that loads new user program from the filesystem. The idea is that a large system can consist of many programs that are stored in the filesystem in the nand flash.

Since the internal filing system of VS1000 ROM works by enumerating all files with allowable extensions, the choice of which file to run is made by the file's extension only. The idea is that the filesystem contains files "EXECFILE.PR1" and "EXECFILE.PR2". First file is selected by running the first file with extension "PR1" and the second file is run by running the first file with extension "PR2". These are created by renaming a Nand-Flash bootable file such as NAND128W.IMG that is created by the build script.

Additionally, parameter passing between programs is demonstrated: Parameter passing can be done by mapping variables to known same locations in different images. In a large project this can be done using the assmebler and ORG directive cleanly. But in a small project there are some tricks such as this one: We use some memory area used by ROM code that is not active, to store the variables. Normally the Bass Boost is not active, it's not activated by ROM code, though user has the option of activating it from software.

We use a 64-word table "s_int16 __y btemp[64]" to store variables that can be seen across different execs.

Since Exec() overwrites the malloc and user program areas, it's generally not possible to return from Exec() to any other place than the main() function of the newly loaded file. Thus each file must contain the Exec() function and exit via loading another program via Exec(). Typically a project would contain "MAIN.PRG" loaded by boot code. It would then run other programs such as "PROGRAM.PR1", "PROGRAM.PR2" etc. Each of them would return by execing to MAIN.PRG file.

Exec() -loadable binaries must be linked with a version of the startup library that resets the stack, otherwise a stack overflow will eventually occur. The command batch file BUILD.BAT prepares a properly linked file with file name EXECFILE.PRG, which can be renamed and dropped to the VS1000 flash disk.

3.8.4 power.c

This file is an example on how to adjust the internal regulator voltages. The file contains the default settings in ROM.

Note that the 3 regulators: CORE, IO and ANALOG have different maximum and minimum voltages and thus the set values will result in different voltages for each regulator. Note also that it's possible to adjust voltages over the limits: it's possible to set e.g. such a low voltage that the core doesn't work (at some clock frequency) or such high voltages that they are harmful to the chip or other chips powered by VS1000. Be careful in your adjustments!

3.8.5 nandprog.c

This program can be used to reset or program the boot area of the nand flash on a developer board. Please see README.TXT file in the developer toolkit for more information.

3.9 Using an external display

The VS1000 can be interfaced easily to an external display controller using the SPI bus. Since all LCD controllers don't have an embedded character generator, the VS1000 includes a ROM font that can be used to draw alphanumeric characters and symbols.

The ROM font contains

- ASCII symbols 32...127
- Half-width katakana symbols
- Special symbols (play,pause,stop,speaker,usb,cabinet,...)

Additional symbols can be defined in RAM.

The low bytes of `u_int16 fontData[]` contain the low end ASCII shapes and variable-width symbols:

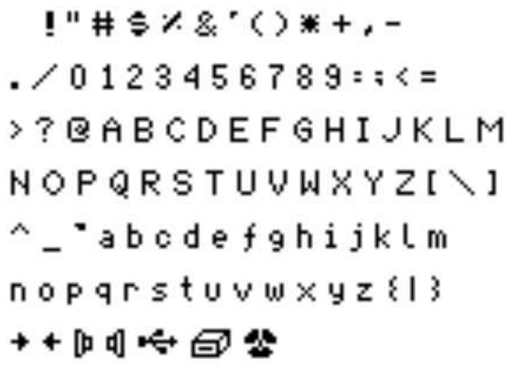

Figure 3.3: VS1000 variable-width symbols

`u_int16 fontPtrs[]` contains the starting offsets of pixel data for each character.

The high bytes of `u_int16 fontData[]` contain katakana and fixed width special symbols:

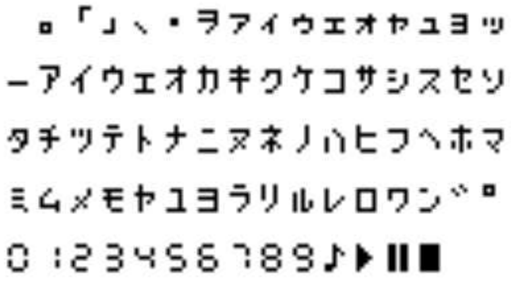

Figure 3.4: VS1000 fixed width symbols

For more information, see files `romfont.txt` and `display.c` in the Developer Toolkit.

4 Peripheral documentation

4.1 VS1000 System Controller

4.1.1 General

The System Controller controls various global aspects of VS1000 function such as the system clock and voltages and I/O pin modes.

4.1.2 Registers

The System Controller is accessed through 2 registers, SCI_SYSTEM and SCI_STATUS.

SCI_SYSTEM: System Power and Clock Control

SCI_SYSTEM Bits		
Name	**Bits**	**Description**
SCISYSF_CLKDIV	15	Divide Clock by 2 (for 24 MHz xtal)
SCISYSF_AVDD	14:10	Analog and Usb Voltage setting 2.5V - 3.6V
SCISYSF_IOVDD	9:5	I/O Voltage setting 1.8V - 3.3V
SCISYSF_CVDD	4:0	Core Voltage setting 1.25V - 2.7V

SCI_SYSTEM controls the internal voltage regulator and clock divider of VS1000A. Setting the clock divider while PLL is not used (clock multiplier = 1) makes the system run at considerably slower clock rate, conserving the system power.

Setting bad voltage values can cause malfuntion and/or even physically harm the device or, in case of IOVDD, even other devices attached to the I/O Pins.

The default values in reset are:

Default Regulator Output Voltages		
Net	**Default Value**	**Description**
AVDD	2.6 V	Analog and Usb Voltage
IOVDD	1.8 V	I/O Voltage
CVDD	1.8 V	Core Voltage

The Core VDD is directly routed to the DSP core and peripheral logic. AVDD and IOVDD are routed by the PCB, allowing PCB layout to generate fixed AVDD and

IOVDD voltages (for AVDD and IOVDD there are separate pins for regulator output and chip input).

USB powering

When USB is active, USB requires at least 2.5V, which is more than the default IOVDD value, so the USB voltage is taken from AVDD output of the internal voltage regulator.

SCI_STATUS: System Flags

The SCI_STATUS register is the second System Controller register. It is used to control and read the state of several system level peripherals.

SCI_STATUS Bits (r/w)		
Name	**Bits**	**Description**
SCISTF_SLOW_CLKMODE	15	Divide XTALI by 256
SCISTF_USB_DN_OUT	14	D- pin output state in GPIO mode
SCISTF_USB_DP_OUT	13	D+ pin output state in GPIO mode
SCISTF_USB_DDR	12	Drive D+/D- pins directly as GPIO
SCISTF_VCM_OVERLOAD	11	VCM pin overload, CBUF disconnected
SCISTF_VCM_DISABLE	10	Disable VCM protection
SCISTF_USB_DP	9	State of D+ pin
SCISTF_USB_DN	8	State of D- pin
SCISTF_USB_DIFF_ENA	7	Enable USB data input
SCISTF_USB_PULLUP_ENA	6	Activate 1.5kOhm D+ pull-up resistor
SCISTF_REGU_POWERLOW	5	Regulator input too low for good AVDD
SCISTF_REGU_POWERBUT	4	State of Power Button ("Play/Pause") pin
SCISTF_ANADRV_PDOWN	3	Analog Output Driver power down control
SCISTF_ANA_PDOWN	2	Analog Core (bias) power down control
SCISTF_REGU_CLOCK	1	Clock in new regulator voltage values
SCISTF_REGU_SHUTDOWN	0	Regulator Shutdown control

USB detection

USB detection and device attachment/detachment are handled using the System Controller. Actual USB data traffic is handled using the USB peripheral itself.

It is suggested that bot the D+ and D- pins have a 1 megaohm pull-up resistor on the PCB. This makes both D+ and D- pins weakly bias to "1" state when the device is not connected to a USB port. When the USB cable is attached, the 15 kilo-ohm pull-down resistors of the host USB hub pull D+ and D- low, pulling the pins to "0" state. Thus detecting SCISTF_USB_DN = 0 indicates USB cable connect.

Upon detecting the connection of the USB cable, software should drive the system clock to 48 MHz (XTALI=12.000MHz, Clock Multiplier 4.0x) and wait for clock to stabilize before setting SCISTF_USB_PULLUP_ENA high, which activates the integrated 1.5 kilo-ohm pull-up resistor of D+, signaling the PC to start enumeration of the USB device.

4.1.3 Conserving Power

Three main factors affect the power requirement of any CMOS device: Clock frequency, voltage and leakage. Of these, clock frequency has the greatest effect to power consumption.

The Clock frequency of VS1000 is controlled by

- The XTALI input (crystal oscillator)
- The System Controller
- The PLL (Phase Locked Loop) Controller (Clock multiplier)

The System Controller's role in clock control is providing two clock dividers between the crystal oscillator output and the analog block and the PLL controller. First there is a divide-by-2 block, which is controlled by SCISTF_SLOW_CLKMODE. After that there is a divide-by-256 block, which is controlled by SCISYSF_CLKDIV.

The divide-by-2 block is normally used when there is a 24 MHz crystal connected to the XTALI/XTALO pins (normally a 12 MHz crystal is used). Setting SCISTF_SLOW_CLKMODE affects all system frequencies, including the PLL, but it does not prohibit using PLL.

It should be noted that the analog block requires 12 Mhz from System Controller for proper operation.

The divide-by-256 block is used to considerably cut down power consumption. This is especially useful when some basic operation is needed (such as the capability to recover from USB suspend or resume after PAUSE mode) but battery life needs to be extended.

The PLL must not be used when divide-by-256 is active. The PLL tries and fails to lock to a frequency below PLL minimum. Switch off PLL (set 1 x clock multiplier) before setting SCISYSF_CLKDIV.

If divide-by-256 is activated without first switching the analog drivers off, the DAC sigma-delta modulator noise (which is part of normal sigma-delta operation) drops down to audible frequencies, which is undesired. To overcome this, set SCISTF_ANADRV_PDOWN before activating SCISYSF_CLKDIV. You should also write 0 to DAC_LEFT DAC_RIGHT to further diminish digital noise and power consumption. Remember to restore the values before resuming playback.

If playback will resume directly after recovering from the power down state, it is not recommended to set SCISTF_ANA_PDOWN since restoring the bias voltages of the analog block can result in a power-up pop sound. If that is not relevant (such as in a USB suspend condition,) SCISTF_ANA_PDOWN should be asserted to further minimize power consumption. Also setting the AVDD, DVDD and CVDD to a lower level will diminish power consumption.

The divide-by-2 and divide-by-256 blocks can be active at the same time, resulting in a master clock that is divided by 512. With the standard 12 MHz crystal, this results in a system clock of just above 23 kHz (23437.5 Hz).

4.1.4 I/O Pin Routing

The System Controller controls the I/O pins of the device, routing signals to/from the peripherals such as a serial port or GPIO controller.

GPIOn_MODE Bits		
Name	**Bits**	**Description**
periph/gpioX	15:0	bit vector; 1=peripheral 0=GPIO

GPIO0_MODE and GPIO1_MODE control output signal routing for the I/O pins. Most pins are multiplexed between general purpose input/output and a peripheral function. Pins are controlled by peripheral functions by default. Writing "0" to a bit in GPIOn_MODE enables direct control over the pin.

Regardless of GPIOn_MODE register value, the input data (1/0 state of pin) can always be read from the GPIOn_IDATA register (See section: Interruptable General Purpose IO).

Switching a pin to GPIO mode can be used to disable data flow from a pin to a peripheral function. The following peripheral input signal values are set when the corresponding pin is in GPIO mode:

Peripheral Function Input Signal Values When pin is in GPIO Mode		
GPIO	**Function**	**Value**
GPIO0[7:0]	Nand Flash data input	00000000
GPIO0[8]	Nand Flash Ready	1
GPIO1[0]	SPI Slave Select	1
GPIO1[1]	SPI Clock	1
GPIO1[2]	SPI MISO	1
GPIO1[3]	SPI MOSI	1
GPIO1[5]	UART Receive	1

4.1.5 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for the System Controller:

At boot-up time, if pin D7 (pin number 12 in LQFP package) is biased high, the ROM software raises IOVDD from 1.8V to 3.3V. If it's biased low, IOVDD remains at 1.8V. The pin should not be left floating.

The default core voltage has been raised to 2.2V in VS1000B.

The ROM code expects a 12.000 MHz crystal input.

4.2 PLL controller v1.0 2006-05-10

4.2.1 General

The Phase-Locked Loop (PLL) controller is used to generate clock frequencies that are higher than the incoming (crystal-based) clock frequency. The PLL output is used by the CPU core and some peripherals.

Configurable features include:

- VCO Enable/Disable
- Select VCO or input clock to be output clock
- Route VCO frequency to output pin
- Select PLL clock multiplier

4.2.2 DAC Interpolator control

The DAC interpolator frequency control and PLL controller are controlled using the same register pair FREQCTLH and FREQCTLL. Output sample rate is derived from the rollover frequency of a 20-bit interpolator accumulator. Its accumulation rate is specified by ifreq.

The maximum value for ifreq is 0x80000. Note that the DAC (and thus also the interpolator) clock is not controlled by the PLL (see "VS1000 System Controller" and "Overview of VS1000A Clocking").

4.2.3 Registers

Nand flash controller user registers can be divided to three groups: the nand flash interface control registers, the dsp interface control registers and the ECC control/logging registers. Register map is shown in the next table.

Interpolator Rate (low part)

FREQCTLL bits		
Name	**Bits**	**Description**
ifreq[15:0]	15:0	Bits 15..0 of the interpolator accumulation rate

Interpolator Rate (high part) and PLL control

FREQCTLH bits		
Name	**Bits**	**Description**
pll-lock-read	13	0=lock failed since last test
pll-lock-test	12	1:Sets pll-lock-read to 1 to start lock test
vco-out-ena	11	Route VCO to GPIO pin (VS1000:second cs pin)
use-pll	9	1:System clock is VCO / 0:System clock is inclk
pll-in-divide	8	divide inclk by 2 (for 1.5, 2.5 or 3.5 x clk)
pll-ratectl	7:4	PLL rate control
ifreq[19:16]	3:0	Bits 19..16 of the interpolator accumulation rate

For comprehensive reference on the function of the clock routing bits, see section "Overview of VS1000A Clocking" below.

At the core of the PLL controller is the VCO, a high frequency oscillator, whose oscillation frequency is adjusted to be an integer multiple of some input frequency. As the name "Phase-Locked Loop" suggests, this is done by comparing the phase of the input frequency against the phase of a signal which is derived from the VCO output through frequency division.

If the system is stable, e.g. the comparison phase difference remains virtually zero, the PLL is said to be "in lock". This means that the output frequency of the VCO is stable and reliable.

The PLL locked status can be checked by generating a high-active pulse (writing first "1" , then "0") to pll-lock-test and reading pll-lock-read. Pll-lock-read is set to "1" along with the high level of pll-lock-test and to "0" whenever the PLL falls out of lock. So if the "1" remains in pll-lock-read, PLL is in sync.

The PLL controller gets its input clock from the System Controller and its operation optimized for frequencies around 12..13 MHz. If you activate clock dividers in the System Controller to get a slow master clock, you should turn the PLL off before (also switch off analog before setting a clock of less than 10 MHz).

Note that USB requires 48.0 MHz for packet sending/receiving.

It's recommended to change the PLL rate in small steps and wait for the PLL to stabilize after each change. For diagnostic purposes, the PLL clock output (VCO) can be routed to an I/O pin so it can be scanned with an oscilloscope.

Bits [7:4] (pll-ratectl) control PLL multiplication rate. PLL multiplier is (pll-ratectl + 1). When pll-ratectl is 0, the VCO is powered down and output clock is forced to be input clock (same as use-pll = 0).

4.2.4 Overview of VS1000A Clocking

Below is a diagram showing the basic layout of the clock signal paths in VS1000A:

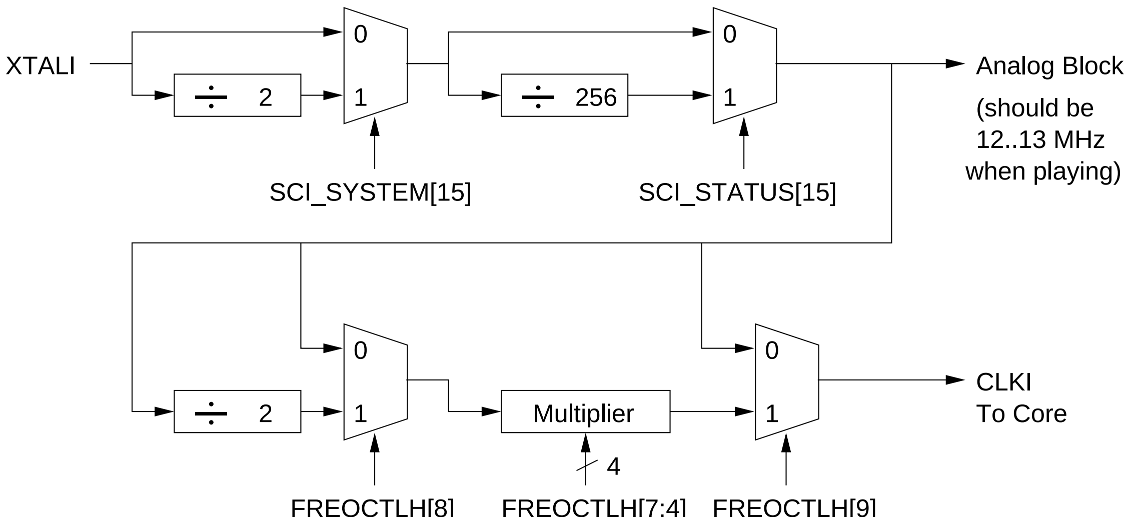

Figure 4.1: VS1000A Clocking

With a 12.0000 megahertz crystal, the following core clock speeds are within limits:

Core Frequency Calculation XTALIN=12.000 MHz					
Register Values					**Result**
SCI_SYSTEM[15]	SCI_STATUS[15]	FREQCTLH[9]	FREQCTLH[8]	FREQCTLH[7:4]	Registers: SCI_SYSTEM[15] XTALI divide by 2 SCI_STATUS[15] XTALI divide by 256 FREQCTLH[9] Use PLL FREQCTLH[8] Divide PLL input clock by 2 FREQCTLH[7:4] PLL rate control
1	1	0	0	0000	0.02344 MHz (23.438 kHz) (Lower CVDD possible)
0	1	0	0	0000	0.04688 MHz
1	0	0	0	0000	6 MHz
0	0	0	0	0000	12 MHz
0	0	1	1	0010	18 MHz
0	0	1	0	0001	24 MHz
0	0	1	1	0100	30 MHz
0	0	1	0	0010	36 MHz
0	0	1	1	0110	42 MHz
0	0	1	0	0011	48 MHz (required by USB, maximum used by ROM code)
0	0	1	1	1000	54 MHz
0	0	1	0	0100	60 MHz (Possible with high CVDD but not recommended)

Note that higher frequencies have higher CVDD requirements and frequencies above 54 MHz are not recommended for production use.

4.2.5 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for PLL:

The clock rate is selected to be 12 MHz by default, 48 MHz when USB is connected and variable between 12 and 48 MHz when Ogg Vorbis is playing. Interpolator rate is set to select sample rate of 44.100 kHz when in the USB audio mode. When Vorbis is playing, the sample rate is set to the sample rate specified in the Ogg file (within 1 Hz steps).

4.3 Interruptable General Purpose IO (VS1000A) v1.0 2002-04-23

4.3.1 General

This chapter describes the interrupt-capable 16-bit general-purpose I/O block for VS_DSP.

Note that in VS1000, pin function is partly handled also by the System Controller: GPIOn_MODE register bits control whether output data for a GPIO pin is taken from a peripheral function (mode="1") or the GPIO controller (mode="0").

4.3.2 Registers

Interruptable General I/O registers, prefix GPIOx_				
Reg	**Type**	**Reset**	**Abbrev**	**Description**
0	r/w	0	DDR	Data direction
1	r/w	0	ODATA	Data output
2	r	0	IDATA	Data input (I/O pin state)
3	r/w	0	INT_FALL	Falling edge interrupt enable
4	r/w	0	INT_RISE	Rising edge interrupt enable
5	r/w	0	INT_PEND	Interrupt pending source
6	w	0	SET_MASK	Data set (→ 1) mask
7	w	0	CLEAR_MASK	Data clear (→ 0) mask
8	r/w	0	BIT_CONF	Bit engine config 0 and 1
9	r/w	0	BIT_ENG0	Bit engine 0 read/write
10	r/w	0	BIT_ENG1	Bit engine 1 read/write

Data Direction GPIOx_DDR

The data direction register (DDR) configures the directions of each of the 16 I/O pins. A bit set to 1 in the DDR turns the corresponding I/O pin to output mode, while a bit set to 0 sets the pin to input mode. The register is set to all zeros in reset, i.e. all pins are inputs by default. The current state of the DDR can also be read.

Output Data GPIOx_ODATA

A write sets the data register value. Change in bits that are configured as outputs are reflected in the outputs. A read returns the state of data register value.

Note: configuring a pin as input should not reset the state of the corresponding data register bit. If the data register is first written 0xffff and then all pins are configured as outputs by writing 0xffff to DDR, all outputs should go to the high state.

This operation enables free selection of polarity for outputs, e.g. after reset a pull-up keeps a control line high, the data register bit is set to 1 and after this the DDR bit is set to 1 enabling the output.

When a data register bit is set to 0, it is easy to use the I/O pin as open-drain-style output by changing the direction: as input the line state is 1 by a pull-up, as output the line is pulled low by the driver.

Possible delays must be documented.

Input Data GPIOx_IDATA

The actual logical levels of the I/O pins are seen in the input data register. Possible delays must be documented.

Falling Edge Interrupt Enable GPIOx_INT_FALL

If a bit the falling edge interrupt enable register (INT_FALL) is set to 1, a falling edge in the corresponding pin (even when configured as output) will set the corresponding bit in the interrupt pending source register (INT_PEND).

Rising Edge Interrupt Enable GPIOx_INT_RISE

If a bit the rising edge interrupt enable register (INT_RISE) is set to 1, a rising edge in the corresponding pin (even when configured as output) will set the corresponding bit in the interrupt pending source register (INT_PEND).

Interrupt Pending Source GPIOx_INT_PEND

If any of the bits in the interrupt pending source register (INT_PEND) are set, an interrupt request is generated. Bits in INT_PEND can be cleared by writing a 1-bit to the bit that is to be cleared.

Note: the interrupt request will remain asserted until all INT_PEND bits are cleared.

Data Set Mask GPIOx_SET_MASK

A bit mask is written to the data set mask register. All bits that are set in the mask also set the corresponding bit in the data output register. Other bits retain their old values. I.e. a logical-OR operation is performed between the data output register old value and the mask and the result is written to the data output register.

Data Clear Mask GPIOx_CLEAR_MASK

A bit mask is written to the data clear mask register. All bits that are set in the mask clear the corresponding bit in the data output register. Other bits retain their old values. I.e. a logical-AND operation is performed between the data output register old value and the mask's inverse and the result is written to the data output register.

Bit Engine Config GPIOx_BIT_CONF

The bit engine config register (BIT_CONF) selects a mapping between an I/O bit and a data output/input register bit for each of the bit engine registers.

GPIOx_BIT_CONF Bits		
Name	**Bits**	**Description**
GPIO_BE_DAT1	15:12	Data bit selection (0..15) for bit engine 1
GPIO_BE_IO1	11:8	I/O bit selection (0..15) for bit engine 1
GPIO_BE_DAT0	7:4	Data bit selection (0..15) for bit engine 0
GPIO_BE_IO0	3:0	I/O bit selection (0..15) for bit engine 0

Bit Engine 0 Read/Write GPIOx_BIT_ENG0

When writing a value to the bit engine 0 register, the data bit specified in the configuration register is copied to the data output register bit specified in the same register.

When reading a value from the bit engine 0 register, the data input register bit specified in the configuration register is copied to the data bit specified in the same register, other bits read out as 0.

Bit Engine 1 Read/Write GPIOx_BIT_ENG1

GPIOx_BIT_ENG1 works just like GPIOx_BIT_ENG0.

4.3.3 VS1000A GPIO Pin Mappings

VS1000A I/O Controller 0 pins and peripheral functions			
GPIO	**Ident**	LQFP Pin	**Function**
GPIO0[0]	NFDIO0	2	Nand-flash IO0 / General-purpose IO Port 0, bit 0
GPIO0[1]	NFDIO1	3	Nand-flash IO1 / General-purpose IO Port 0, bit 1
GPIO0[2]	NFDIO2	4	Nand-flash IO2 / General-purpose IO Port 0, bit 2
GPIO0[3]	NFDIO3	5	Nand-flash IO3 / General-purpose IO Port 0, bit 3
GPIO0[4]	NFDIO4	9	Nand-flash IO4 / General-purpose IO Port 0, bit 4
GPIO0[5]	NFDIO5	10	Nand-flash IO5 / General-purpose IO Port 0, bit 5
GPIO0[6]	NFDIO6	11	Nand-flash IO6 / General-purpose IO Port 0, bit 6
GPIO0[7]	NFDIO7	12	Nand-flash IO7 / General-purpose IO Port 0, bit 7
GPIO0[8]	NFRDY	13	Nand-flash READY / General-purpose IO Port 0, bit 8
GPIO0[9]	NFRD	14	Nand-flash RD / General-purpose IO Port 0, bit 9
GPIO0[10]	NFCE	15	Nand-flash CE / General-purpose IO Port 0, bit 10
GPIO0[11]	NFWR	20	Nand-flash WR / General-purpose IO Port 0, bit 11
GPIO0[12]	NFCLE	16	Nand-flash CLE / General-purpose IO Port 0, bit 12
GPIO0[13]	NFALE	17	Nand-flash ALE / General-purpose IO Port 0, bit 13
GPIO0[14]	CS2	21	General-purpose IO Port 0, bit 14

VS1000A I/O Controller 1 pins and peripheral functions			
GPIO	**Ident**	LQFP Pin	**Function**
GPIO1[0]	XCS	22	SPI XCS / General-Purpose I/O Port 1, bit 0
GPIO1[1]	SCLK	23	SPI CLK / General-Purpose I/O Port 1, bit 1
GPIO1[2]	SI	24	SPI MISO / General-Purpose I/O Port 1, bit 2
GPIO1[3]	SO	25	SPI MOSI / General-Purpose I/O Port 1, bit 3
GPIO1[4]	TX	26	UART TX / General-Purpose I/O Port 1, bit 4
GPIO1[5]	RX	27	UART RX / General-Purpose I/O Port 1, bit 5

4.3.4 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for GPIO pins:

4.4 Interrupt Controller v1.0 2002-04-23

The interrupt controller is used to forward interrupt requests from peripherals to VSDSP. The 32 interrupt sources are vectorized, i.e. the VS_DSP core jumps to a different address according to the 5-bit interrupt vector value. There are three levels of priority for simulteneous requests and a global disable available for all of the sources.

For an interrupt handler written in C, an assembly language stub that re-enables interrupts before RETI, should be written. The assembly language stub should call the C language handler routine.

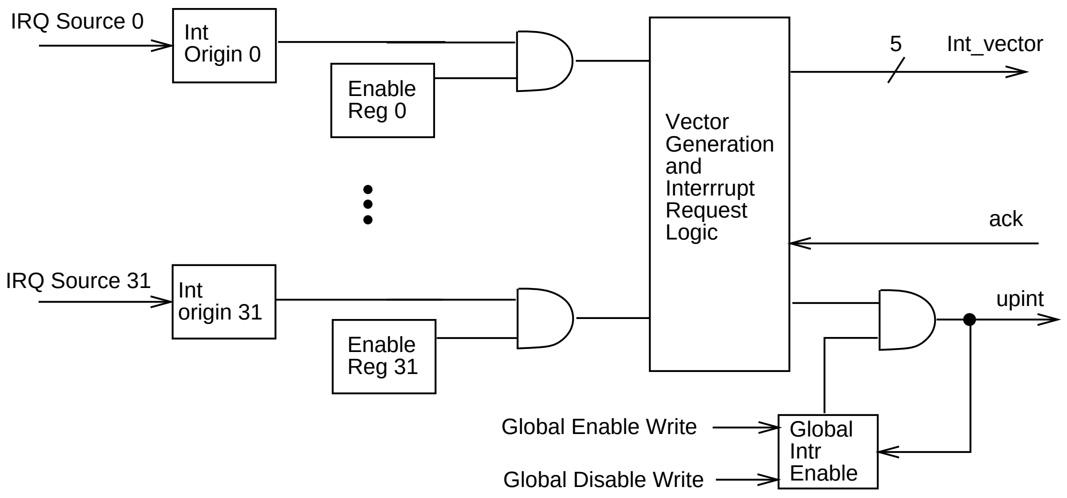

Figure 4.2: Interrupt Controller Block Diagram

4.4.1 Registers

Interrupt Controller registers, prefix INT_				
Reg	**Type**	**Reset**	**Abbrev**	**Description**
0	r/w	0	ENABLEL0	Interrupt Enable Low 0
1	r/w	0	ENABLEL1	Interrupt Enable Low 1
2	r/w	0	ENABLEH0	Interrupt Enable High 0
3	r/w	0	ENABLEH1	Interrupt Enable High 1
4	r/w	0	ORIGIN0	Interrupt Origin 0
5	r/w	0	ORIGIN1	Interrupt Origin 1
6	r	0	VECTOR[4:0]	Interrupt Vector
7	r/w	0	ENCOUNT[2:0]	Interrupt Enable Counter
8	w	0	GLOB_DIS[-]	Interrupt Global Disable
9	w	0	GLOB_EN[-]	Interrupt Global Enable

- Enable registers, which contain enable/disable bits for each interrupt source. Bit pairs configure the interrupt priority and disable.
- Origin registers, which contain the source flags for each interrupt. A request from an interrupt source sets the corresponding bit. A bit is automatically reset when a request for the source is generated.
- Enable counter register, which contains the value of the General Interrupt Enable counter, and two registers for increasing and decreasing the value.

Enable INT_ENABLE[L/H][0/1]

Interrupt enable registers selectively masks interrupt sources. Enable registers 0 contain sources 0..15 and enable registers 1 contain sources 16..31. Each source has two enable bits: one in the enable low and one in the enable high register. If both bits are zero, the corresponding interrupt source is not enabled, otherwise the bits select the interrupt priority.

High	Low	Priority
0	0	Source disabled
0	1	Priority 1
1	0	Priority 2
1	1	Priority 3

Priorities only matter when the interrupt controller decides which interrupt to generate for the core next. This happens whenever two interrupt sources request interrupts at the same time, or when interrupts become enabled after an interrupt handler routine or part of code where the interrupts have been disabled.

Origin INT_ORIGIN[0/1]

If an interrupt source requests an interrupt, the corresponding bit in the interrupt origin register (ORIGIN0 or ORIGIN1) will be set to '1'. If an interrupt source is enabled (using ENABLE registers), the interrupt controller generates an interrupt request signal for VSDSP with the corresponding vector value. The bit in the origin registers is reset automatically after the interrupt is requested.

If the source is not enabled, the processor can read the origin register state and perform any necessary actions without using interrupt generation, i.e. polling of the interrupt sources is also possible. The bits in the interrupt origin registers can be cleared by writing '1' to them.

A read from the interrupt origin register returns the register state.

A write to the interrupt origin register clears bits in the interrupt origin register. All '1'-bits in the written value cause the corresponding bits in the interrupt origin register to be cleared. All zero-bits cause the corresponding bits in the interrupt origin register to keep their state. For example writing a value 0x00ff will clear the lowest eight bits in the interrupt origin register, while leaving the upper bits as-is.

Vector INT_VECTOR

The last generated vector value can be read from the vector register.

Enable Counter INT_ENCOUNT

The global interrupt enable/disable is used to control whether an interrupt request is sent to the processor or not. Whenever this 3-bit counter value is non-zero, interrupt requests are not forwarded to VSDSP. The counter is increased by one whenever the interrupt controller generates an interrupt request for VSDSP, thus disabling further interrupts.

When read, the enable counter register returns the counter value.

Don't write directly to INT_ENCOUNT. Use INT_GLOB_DIS and INT_GLOB_EN to manipulate the value of this register.

Global Disable INT_GLOB_DIS

A write (of any value) to global disable register increases the global interrupt enable/disable counter by one. If the counter is zero, interrupt signal generation is enabled. When the interrupt arbitrator generates an interrupt request for VS_DSP core, it automatically increases the counter. The user must write to the global enable register (once) to enable interrupts.

If an interrupt is generated in the same cycle as a write to global disable register, the interrupt enable counter is increased by two.

Global Enable INT_GLOB_EN

A write (of any value) to global enable register decreases the global interrupt enable/disable counter by one. If the counter is zero, interrupt generation is enabled.

The user must write to this register once in the end of the interrupt handler to enable further interrupts. This should be done in assembly language.

4.4.2 VS1000A Interrupt Sources

VS1000A Interrupt Sources		
Name	**Vector**	**Source**
INTV_DAC	0	Digital to Analog Converter
INTV_SPI	1	Serial Peripheral Interface
INTV_USB	2	Universal Serial Bus
INTV_NFLSH	3	Byte-wide Bus (Nand Flash) Controller
INTV_TX	4	UART Transmit
INTV_RX	5	UART Receive
INTV_TIM0	6	Timer 0 underflow
INTV_TIM1	7	Timer 1 underflow
INTV_REGU	8	Input Voltage Monitor
INTV_GPIO0	9	I/O Pin Controller 0
INTV_GPIO1	10	I/O Pin Controller 1

4.4.3 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for interrupts:

Timer 0 interrupt is used for software real time counter.

UART RX interrupt is used for the ROM monitor

DAC interrupt is used for loading samples to the DAC.

4.5 SPI v1.3 2005-06-09

4.5.1 General

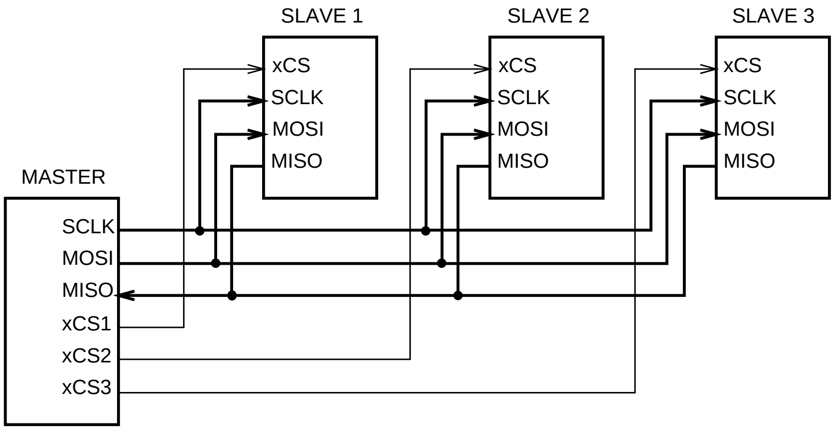

Figure 4.3: SPI Bus

SPI is a serial bus interface that allows for simple serial communication between one host and potentially several slaves. As depicted in Figure 4.3, four different signals are required for implementing SPI:

- SCLK (Master Serial Clock): a static serial clock, offered by the master.
- MOSI (Master Out / Slave In): Master's output data. This output is always driven by the master.
- MISO (Master In / Slave Out): Slave's output data. By default, all slaves on the bus are in high impedance state. When the slave's chip select is activated, it turns MISO to an output, and when it starts receiving SCLKs, it behaves as defined in the slave's specification.
- xCS (Chip Select): Every slave requires its own chip select. Without the chip select signal, a slave may not listen to what happens on the SPI bus.

Although widely used, SPI is not a real standard. Because of this, there are many different implementations, more or less compatible with each other. Also, a very similar de-facto standard, SSI, is in wide use with e.g. D/A converters. Again, there exists another de-facto standard very close to this, Microwire. Thus, if one wants to make an SPI/SSI/Microwire master device that works with all kinds of different slaves, it must be well configurable.

SPI Block Compatibility		
Format	**Master**	**Slave**
SPI	Yes	Yes
SSI	Yes	Yes
Microwire	Yes	No

4.5.2 The SPI Block

The SPI block can implement both a master and slave SPI mode. Figure 4.4 shows the two different physical connections for the modes. Chip Select extensions in master mode allow for implementing several SSI variants. Also, Microwire master mode may be implemented with this same arrangement.

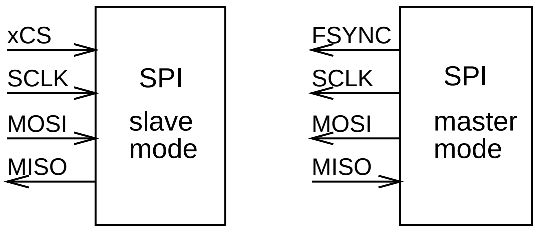

Figure 4.4: SPI Pins

The SPI block is quite flexible, and allows for many different SPI configurations. Input and output clock edges may be set independently, and the whole clock may be inverted. In master mode, it is possible to delay reading a value for a given number of clock cycles after a given clock edge, making it possible to make SPI implementations that are not dependent of the output clock edge of a slave device, with the price of decreased maximum SPI speed.

The most typical SPI configuration is such that 8-bit transfers are written MSB first to the bus at falling clock edges, and read at a rising clock edges. When a transfer is not active, the clock is low. This case is presented in Figure 4.5 (SPI_CF_CLKOPOL=1).

Master Mode

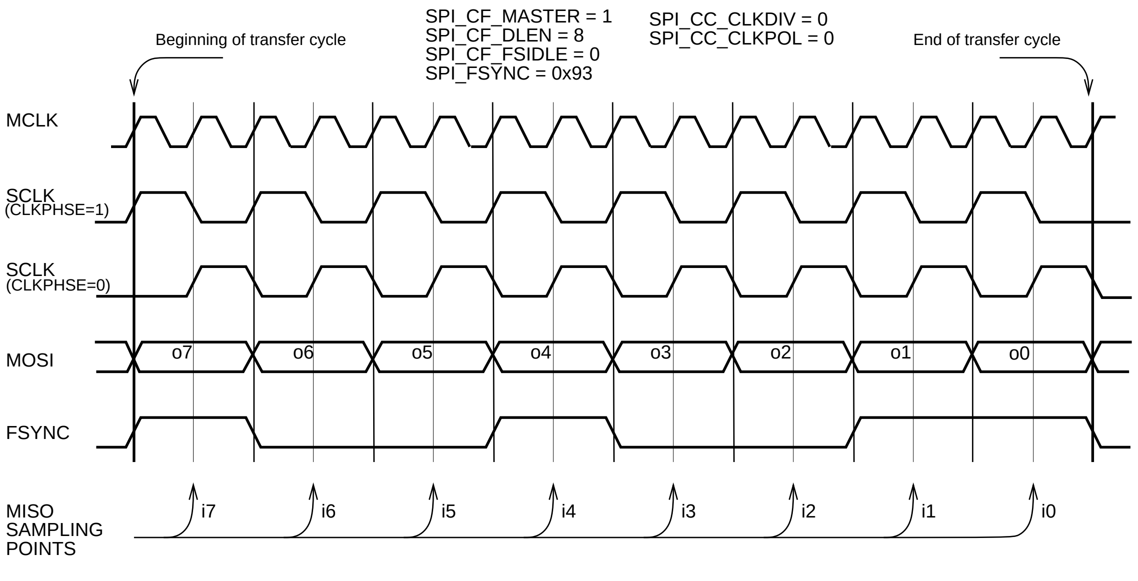

Figure 4.5: Example SPI Timing, Master Mode

In master mode, the SPI clock SCLK is created in the SPI block. MOSI, SCLK and FSYNC are in output mode, and MISO in input mode. No pins are in high impedance state.

The highest speed that can be expected to work is $f_s = \frac{1}{2} \times f_m$, where f_s is the SPI speed and f_m is the SPI block's input clock.

If more than one slave devices are to be used, each device requires a separate chip select signal. Chips selects are intended to be implemented with general I/O pins.

FSYNC is mainly intended to be used for SSI device synchronization purposes. If it is not needed for synchronization, it can also be used to implement one chip select. This approach makes it possible to create a chip select that is automatically deasserted when a transfer is finished.

Slave Mode

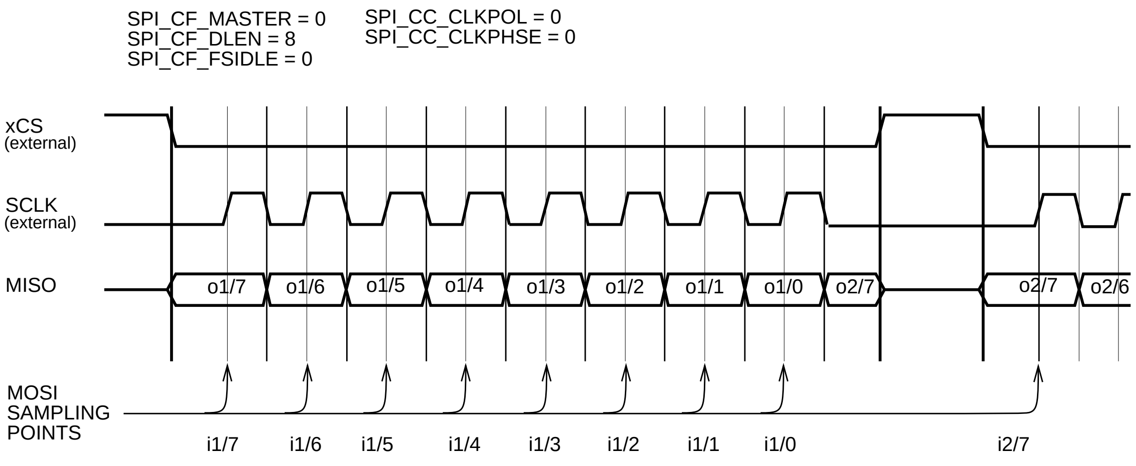

Figure 4.6: Example SPI Timing, Slave Mode

In slave mode, the SPI clock SCLK is created externally. MOSI, SCLK and xCS are inputs, and MISO is only an output when xCS is active. Otherwise MISO is high impedance, as can be seen in Figure 4.6. The high impedance state is handled outside the SPI block (with gpio control).

In slave mode, the external clock, SCLK is used for latching input bits asynchronously to the master clock MCLK.

The highest recommended input clock speed is slightly lower than $f_s = \frac{1}{2} \times f_m$, where f_s is the input SPI speed and f_m is the SPI block's input clock. The highest operable input clock speed depends on the SPI block's input clock speed, on the core clock speed, and on the software.

There are three receive modes:

1. Interrupted xCS mode
2. Falling edge xCS mode

3. Rising edge xCS mode

In interrupted xCS mode the clock is only listened to if xCS is active. Reception starts when xCS state changes from high to low. If xCS is deasserted in the middle of the transfer, the reception is aborted.

In falling edge xCS mode reception starts when xCS state changes from high to low, but transfer is not aborted if xCS changes from low to high mid-transfer. If another high to low transition is encountered during the transfer of SPI_CF_DLEN+1 bits, the partially received data is moved to the data register, SPI_ST_BREAK is set, interrupt 0 request is sent, and a new transfer is initiated.

Rising edge xCS mode works like the falling edge xCS mode, except that the polarity of the synchronization is reversed.

4.5.3 Registers

SPI registers, prefix SPIx_				
Reg	**Type**	**Reset**	**Abbrev**	**Description**
0	r/w	0	CONFIG[10:0]	Configuration
1	r/w	0	CLKCONFIG	Clock configuration
2	r/w	0	STATUS[7:0]	Status
3	r/w	0	DATA	Sent / received data
4	r/w	0	FSYNC	SSI Sync data in master mode
5	r/w	0	DEFAULT	Data to send (slave) if SPI_ST_TXFULL='0'

Main Configuration SPIx_CONFIG

SPIx_CONFIG Bits		
Name	**Bits**	**Description**
SPI_CF_SRESET	11	SPI software reset
SPI_CF_RXFIFOMODE	10	'0' = interrupt always when a word is received, '1' = Interrupt only when FIFO register full or CS deasserted with receive register full
SPI_CF_RXFIFO	9	Receive FIFO enable
SPI_CF_TXFIFO	8	Transmit FIFO enable
SPI_CF_XCSMODE	7:6	xCS mode in slave mode
SPI_CF_MASTER	5	Master mode
SPI_CF_DLEN	4:1	Data length in bits
SPI_CF_FSIDLE	0	Frame sync idle state

SPI_CF_XCSMODE selects xCS mode for slave operation. '00' is interrupted xCS mode, '10' is falling edge xCS mode, and '11' is rising edge xCS mode.

SPI_CF_MASTER sets master mode. If not set, slave mode is used.

SPI_CF_DLEN+1 is the length of SPI data in bits. Example: For 8-bit data transfers, set SPI_CF_DLEN to 7.

SPI_CF_FSIDLE contains the state of FSYNC when SPI_ST_TXRUNNING is clear. This bit is only valid in master mode.

Clock Configuration SPIx_CLKCONFIG

SPIx_CLKCONFIG Bits		
Name	**Bits**	**Description**
SPI_CC_CLKDIV	9:2	Clock divider
SPI_CC_CLKPOL	1	Clock polarity selection
SPI_CC_CLKPHASE	0	Clock phase selection

In master mode, SPI_CC_CLKDIV is the clock divider for the SPI block. The generated SCLK frequency $f = \frac{f_m}{2\times(c+1)}$, where f_m is the master clock frequency and c is SPI_CC_CLKDIV. Example: With a 12 MHz master clock, SPI_CC_CLKDIV=3 divides the master clock by 4, and the output/sampling clock would thus be $f = \frac{12MHz}{2\times(3+1)} = 1.5MHz$.

SPI_CC_CLKPOL reverses the clock polarity. In master mode, the inverter is implemented as the last thing in the output clock data chain. In slave mode, it is implemented as the first thing in the input clock data chain. See Figure 4.7 for details. If SPI_CC_CLKPOL is clear the data is read at rise edge and written at fall edge if SPI_CC_CLKPHASE is clear. When SPI_CC_CLKPHASE is set the data is written at rise edge and read at fall edge.

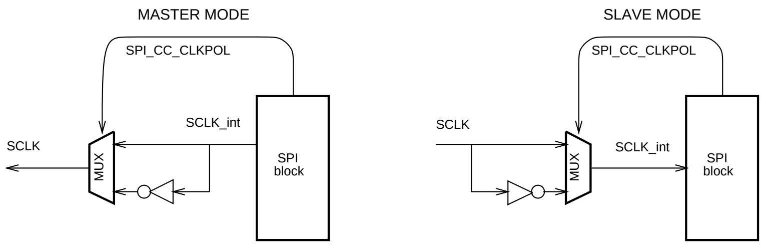

Figure 4.7: Normal and Reverese SPI Clock Polarity

SPI_CC_CLKPHASE defines the data clock phase. If clear the first data is written when xcs is asserted and data is sampled at first clock edge (rise edge when SPI_CC_CLKPOL = 0 and fall edge if SPI_CC_CLKPOL = 1). If SPI_CC_CLKPHASE is set the first data is written a the first data clock edge and sampled at second.

Status SPIx_STATUS

SPIx_STATUS Bits		
Name	**Bits**	**Description**
SPI_ST_RXFIFOFULL	7	Receiver FIFO register full
SPI_ST_TXFIFOFULL	6	Transmitter FIFO register full
SPI_ST_BREAK	5	Chip select deasserted mid-transfer
SPI_ST_RXORUN	4	Receiver overrun
SPI_ST_RXFULL	3	Receiver data register full
SPI_ST_TXFULL	2	Transmitter data register full
SPI_ST_TXRUNNING	1	Transmitter running
SPI_ST_TXURUN	0	Transmitter underrun

SPI_ST_BREAK is set in slave mode if chip select was deasserted in interrupted xCS mode or a starting edge is encountered in xCS edge modes while a data transfer was in progress. This bit has to be cleared manually.

SPI_ST_RXORUN is set if a received byte overwrites unread data when it is transferred from the receiver shift register to the data register. This bit has to be cleared manually.

SPI_ST_RXFULL is set if there is unread data in the data register.

SPI_ST_TXFULL is set if the transmit data register is full.

SPI_ST_TXRUNNING is set if the transmitter shift register is in operation.

SPI_ST_TXURUN is set if an external data transfer has been initiated in slave mode and the transmit data register has not been loaded with new data to shift out. This bit has to be cleared manually.

Note: Because TX and RX status bits are implemented as separate entities, it is relatively easy to make asynchronous software implementations, which do not have to wait for an SPI cycle to finish.

Data SPIx_DATA

SPIx_DATA[SPI_CF_DLEN:0] may be written to whenever SPI_ST_TXFULL is clear. In master mode, writing will initiate an SPI transaction cycle of SPI_CF_DLEN+1 bits. In slave mode, data is output as soon as suitable external clocks are offered. Writing to SPIx_DATA sets SPI_ST_TXFULL, which will again be cleared when the data word was put to the shift register. If SPI_ST_TXRUNNING was clear when SPIx_DATA was written to, data can immediately be transferred to the shift register and SPI_ST_TXFULL won't be set at all.

When SPI_ST_RXFULL is set, SPIx_DATA may be read. Bits SPI_CF_DLEN:0 contain the received data. The rest of the 16 register bits are set to 0.

SSI Synchronization SPIx_FSYNC

SPIx_FSYNC is meant for generation of potentially complex synchronization signals, including several SSI variants as well as a simple enough automatic chip select signal. SPIx_FSYNC is only valid in master mode.

If a write to SPIx_DATA is preceded by a write to SPIx_FSYNC, the data written to SPIx_FSYNC is sent to FSYNC with the same synchronization as the data written to SPIx_DATA is written to MOSI. When SPI_ST_TXRUNNING is clear, the value of SPI_CF_FSIDLE is set to FSYNC.

If SPIx_DATA is written to without priorly writing to SPIx_FSYNC, the last value written to SPIx_FSYNC is sent.

SPIx_FSYNC is double-buffered like SPIx_DATA.

4.5.4 Interrupts

The SPI block has one interrupt.

Interrupt 0 request is sent when SPI_ST_BREAK is asserted, or when SPI_ST_TXFULL or SPI_ST_TXRUNNING is cleared. This allows for sending data in an interrupt-based routine, and turning chip select off when the device becomes idle.

4.5.5 Changes from 1.2

A default data register is added. If in slave mode there is no data to send when it is needed (SPI_ST_TXFULL is '0'), the default data is sent (and SPI_ST_TXURUN is set like before).

In addition to receive and transmit data registers another set of FIFO registers are added. In normal mode these are not used. If SPI_CF_TXFIFO is set, two words can be waiting while a third one is in transmit. An interrupt is generated when SPI_ST_TXFULL becomes '0' (like before).

If SPI_CF_RXFIFO is set, RX FIFO register holds another received word while a third one is in receive. When SPI_DATA is read and SPI_ST_RXFIFO is '1', the FIFO register value is returned, otherwise the receive register value is returned.

Status register should be writable by user, i.e. it must be possible to clear the state of FIFO and transmit/receive register indicators.

The clock configuratio register operations has changed. This device uses the common SPI clocking configuration modes where data clock's polarity and phase can be inverted.

4.5.6 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for SPI:

At boot-up the SPI chip select is checked: if it's pulled high, SPI boot is attempted.

When SPI is not active, the default player application uses the SPI data lines as LED controls.

4.5.7 Effect of Clock Multiplier

Note that the clock multiplier affects SPI speed. In VS1000A ROM you can read the current clock multiplier setting in global variable clockX. Here's a line of code that sets the SPI clock speed taking the clock multiplier into account:

```
PERIP(SPI0_CLKCONFIG) = SPI_CC_CLKDIV * (clockX - 1);
```

4.6 Byte-wide bus/Nand Flash controller v1.0 2006-05-10

4.6.1 General

The byte-wide bus peripheral implements a nand flash controller with vsdsp peripheral bus interface. The peripheral can be configured for different speed/size memory devices. The device has internal ECC calculation which provides the error detection data for the dsp.

Operation as a Nand Flash controller requires that dsp controls the command latch enable (CLE), address latch enable (ALE) and memory chip enable (CEx) pins directly (as GPIO).

The peripheral provides clocked byte transfers of 1..32 bytes from an integrated buffer memory freeing the DSP from having to generate clocking for each transferred byte. The peripheral also provides standard Error Correcting Code (ECC) calculation for 1..512 byte blocks.

Configurable features include:

- Programmable address cycles from 1 to 32
- Programmable wait states from 0 to 63 (i.e. Read/write pulse time)
- ECC calculation disable/enable
- Interrupt request disable/enable
- Chip select write mode continuous/byte-at-a-time (for LCDs)
- 1 - 512 byte blocks ECC calculation (in 16-bit words)
- Programmable burst transactions from 1 to 32 bytes

4.6.2 Block Diagram

Nand flash controller consists of the memory interface signal generation unit and ECC calculation and logging unit. These units can operate separately from each other.

The peripheral implements a memory mapped interface that generates the control signals for flash memory read/write operation. It also calculates and logs the parity bit information from one read/write block. The block size is not limited but the byte counter is only 9 bits. Reads/writes can be done one at a time or from a 32-byte data buffer in bursts from 1 to 32 bytes at a time. Block diagram with the main registers is shown in the next figure.

The architecture has timing control logic which controls the flash operation delay of each write/read. This logic controls the wex, rex and cex signal toggling. Wex and rex pulses are always symmetric. Without wait states each write/read cycle takes two master clock cycles. When waitstates are set to 1 each cycle takes 2+2 master clock cycles. I.e. Each

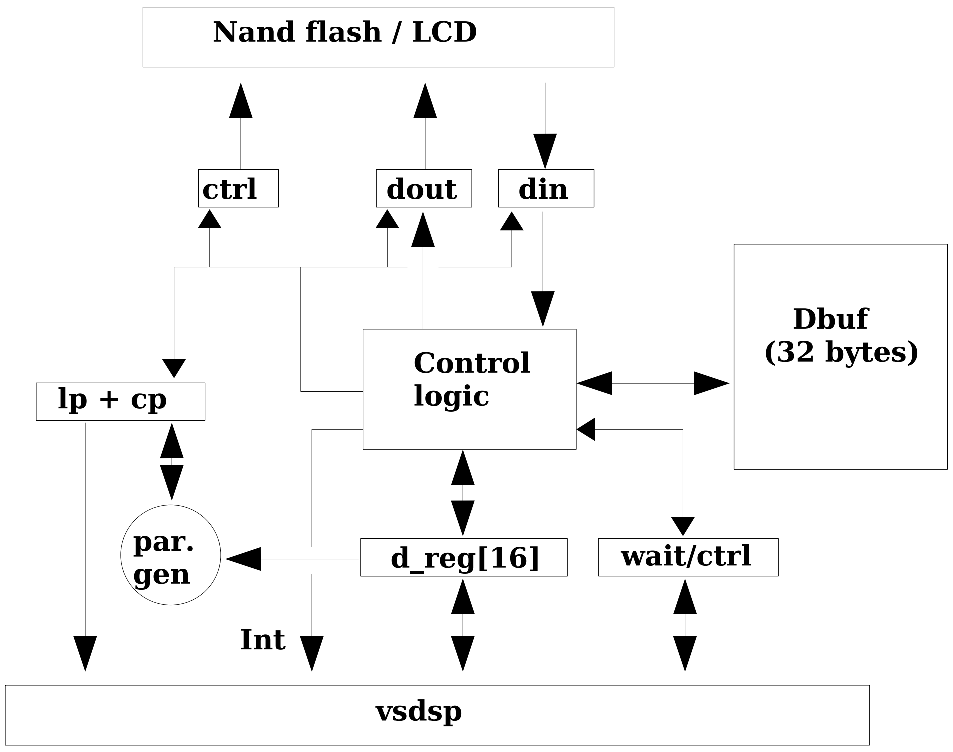

Figure 4.8: Byte-wide Bus Controller Block Diagram.

operation takes (waitstates+1) * 2 master clock cycles. Waitstates can be set from 0 to 63 (6-bit register). For LCDs the chip select in write mode can be set to toggle between bytes.

Buffer memory operation

The 32-byte buffer memory consists of 16 addresses, 16 bits each. In the byte-wide bus operations, the high 8 bits (MSB) are transferred first.

4.6.3 Registers

Nand flash controller user registers can be divided to three groups: the nand flash interface control registers, the dsp interface control registers and the ECC control/logging registers. Register map is shown in the next table.

Byte-wide bus peripheral registers			
Offset	**Type**	**Register**	**Function**
0	rw	CTRL[8:0]	Byte-wide Bus (Nand Flash) Controller Control
1	r	LPL[15:0]	Calculated Line Parity for 512-byte block
2	r	CP_LPH[7:0]	Calculated Column Parity for 512-byte block
3	rw	DATA[15:0]	Buffer Data read/write register
4	rw	NFIF[12:0]	Buffer-to-Physical Interface Control
5	rw	DSPIF[7:0]	Buffer-to-DSP Interface Control
6	r	ECC_CNT[7:0]	Error Correction Code counter

Control register

NFLSH_CTRL bits		
Name	**Bits**	**Description**
lcd-ce-mode	8	Chip select operation mode in read/write cycles
int-enable	7	Interrupt enable
nf-sreset	6	Resets the controller
waitstates	5:0	Number of wait states in read/write cycles

Waitstates delays the read/write operation by (1+n)+(1+n) master clock cycles where n is the number of wait states. I.e. The flash read/write enable low and high times are both delayed.

Line and Column parity registers

NFLSH_LPL bits		
Name	**Bits**	**Description**
lpl	15:0	Low part (bits 15:0) of Line Parity

NFLSH_CP_LPH bits		
Name	**Bits**	**Description**
cp	7:2	Calculated Column Parity bits (6 bits)
lph	1:0	High part (bits 17:16) of Line Parity

Lp and cp calculate the parity bits as descibed in Samsung's Application Note for NAND Flash Memory (Revision 2.0). The parity calculation can be used with or without actually accessing any physical Nand Flash device. A nand operation can be active during ECC calculation but it must be from/to the data buffer.

When ECC is enabled (ecc-ena=1), each read and write to the dreg register updates the ECC. ECC is calculated from the 16-bit values of dreg register.

The ECC generation uses the Hamming ECC principle. In case of 528 byte page nand flash (small page) a 24 bit ECC is generated. This gives a performance of 2 bit detection and 1 bit correction. For 2112 byte page nand flash memories (large page) the calculation can be done in 512 byte sections.

Data register

NFLSH_DATA bits		
Name	**Bits**	**Description**
dreg	15:0	Data read/write register. Can be used with or without ECC.

All data transfers to/from the are done through this register. The operation of NFLSH_DATA depends from dsp-ena-dbuf, dsp-rd-wrx and nf-rd-wrx. When dsp-rd-wrx is set the register samples the data buffer (from pointer address dsp-dbuf-pntr) or the nand flash input register (when dsp-ena-dbuf is low).

Data buffer reads/writes can be done in 16 consecutive clock cycles. It must be noted that when the read mode (dsp-rd-wrd set) is selected it takes one clock cycle for the control to transfer the first word from data buffer to dreg. Therefore it is recommended that the read mode is set (+ ecc reset/enable/disable) as the nand flash operation is started.

Interface control towards physical pins

NFLSH_NFIF bits		
Name	**Bits**	**Description**
nf-byte-cnt	12:8	Rx / tx byte counter, hardware sends nf-byte-cnt + 1 bytes
nf-use-dbuf	7	write from buffer(1) or dreg register(0)
nf-dbuf-pntr	6:2	pointer address of the data buffer for next read/write
nf-do-op	1	nand flash interface start operation bit (resets when done)
nf-rd-wrx	0	read(1)/write(0) selection

NFIF control register can only be written in idle state. Current nand flash operation can be terminated by setting the nf-sreset bit of the control register. When all bytes are read/written an interrupt is given (if enabled)

Interface control towards DSP

NFLSH_DSPIF bits		
Name	**Bits**	**Description**
dsp-dbuf-pntr	7:4	Data buffer pointer for next operation
dsp-ena-dbuf	3	Use data buffer for operations ('1' = enabled)
dsp-rd-wrx	2	Dsp read/write selection ('1' = read)
ecc-ena	1	Ecc calculation enable
ecc-sreset	0	Ecc register reset bit (zeroed after one cycle)

When dsp-ena-dbuf is 0, the 32-byte buffer memory is not changed.

ECC counter register

NFLSH_ECC_CNT bits		
Name	**Bits**	**Description**
ecc-cnt	7:0	Calculated ecc words (data is processed in 16-bit format)

Ecc-cnt register counts the 16-bit words that are read or written to dreg. This information is required when lpl, lph and cp are calculated. The register is updated only when the ecc is enabled (ecc-ena = '1'). In write operation the register is updated one clock cycle after the write took place (as the data is being moved to the data buffer) and in the read operation it is updated in the same clock cycle.

4.6.4 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for the Nand Flash controller:

At boot-up the Nand Flash chip select is checked: if it's pulled high, Nand flash scan is attempted. For proper recognition of the Nand Flash, a number of access methods are used to attempt to read the first 512 bytes of the Nand Flash chip and look for an 8-byte NandType record from the beginning of the block. The NandType record should confirm the proper access method number for the Flash in question and specify the device size and erasable block size of the Flash chip (see datasheet).

The remaining 504 bytes of the first block can contain a VSDSP boot record, which can be used to load data to X RAM or I RAM and optionally execute code to extend or replace firmware functionality on chip.

If the ROM does not have a proper access method for the Nand Flash, but reading at least the first block is successful with one of the ROM methods, the boot record can be used to load an access method to RAM so the bootup can be continued.

Nand Flash access methodology

VS1000A writes to the nand flash in blocks of 512 (data) + 16 (spare) bytes. single-level cell (SLC) large page flashes (block size 2112) are mostly ok with this, but multi-level cell (MLC) have problems with this so those are not supported by the ROM code. VS1000A ROM contains own wear levelling algorithm and logical-to-physical block mapper that greatly extends the life of the nand flash chips.

4.7 Timers v1.0 2002-04-23

4.7.1 General

There are two 32-bit timers that can be initialized and enabled independently of each other. If enabled, a timer initializes to its start value, written by a processor, and starts decrementing every clock cycle. When the value goes past zero, an interrupt is sent, and the timer initializes to the value in its start value register, and continues downcounting. A timer stays in that loop as long as it is enabled.

A timer has a 32-bit timer register for down counting and a 32-bit TIMER1_LH register for holding the timer start value written by the processor. Timers have also a 2-bit TIMER_ENA register. Each timer is enabled (1) or disabled (0) by a corresponding bit of the enable register.

4.7.2 Registers

Timer registers, prefix TIMER_				
Reg	**Type**	**Reset**	**Abbrev**	**Description**
0xC030	r/w	0	CONFIG[7:0]	Timer configuration
0xC031	r/w	0	ENABLE[1:0]	Timer enable
0xC034	r/w	0	T0L	Timer0 startvalue - LSBs
0xC035	r/w	0	T0H	Timer0 startvalue - MSBs
0xC036	r/w	0	T0CNTL	Timer0 counter - LSBs
0xC037	r/w	0	T0CNTH	Timer0 counter - MSBs
0xC038	r/w	0	T1L	Timer1 startvalue - LSBs
0xC039	r/w	0	T1H	Timer1 startvalue - MSBs
0xC03A	r/w	0	T1CNTL	Timer1 counter - LSBs
0xC03B	r/w	0	T1CNTH	Timer1 counter - MSBs

Configuration TIMER_CONFIG

TIMER_CONFIG Bits		
Name	**Bits**	**Description**
TIMER_CF_CLKDIV	7:0	Master clock divider

TIMER_CF_CLKDIV is the master clock divider for all timer clocks. The generated internal clock frequency $f_i = \frac{f_m}{c+1}$, where f_m is the master clock frequency and c is TIMER_CF_CLKDIV. Example: With a 12 MHz master clock, TIMER_CF_DIV=3 divides the master clock by 4, and the output/sampling clock would thus be $f_i = \frac{12MHz}{3+1} = 3MHz$.

Configuration TIMER_ENABLE

TIMER_ENABLE Bits		
Name	**Bits**	**Description**
TIMER_EN_T1	1	Enable timer 1
TIMER_EN_T0	0	Enable timer 0

Timer X Startvalue TIMER_Tx[L/H]

The 32-bit start value TIMER_Tx[L/H] sets the initial counter value when the timer is reset. The timer interrupt frequency $f_t = \frac{f_i}{c+1}$ where f_i is the master clock obtained with the clock divider (see Chapter 4.7.2 and c is TIMER_Tx[L/H].

Example: With a 12 MHz master clock and with TIMER_CF_CLKDIV=3, the master clock $f_i = 3MHz$. If TIMER_TH=0, TIMER_TL=99, then the timer interrupt frequency $f_t = \frac{3MHz}{99+1} = 30kHz$.

Timer X Counter TIMER_TxCNT[L/H]

TIMER_TxCNT[L/H] contains the current counter values. By reading this register pair, the user may get knowledge of how long it will take before the next timer interrupt. Also, by writing to this register, a one-shot different length timer interrupt delay may be realized.

4.7.3 Interrupts

Each timer has its own interrupt, which is asserted when the timer counter underflows.

4.7.4 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for timers:

Timer 0 is used as the System Timer, updating a software real time counter that is used for all timing of the ROM routines.

Timer 1 is free for user applications.

4.8 UART v1.11 2007-03-16

4.8.1 General

RS232 UART implements a serial interface using rs232 standard.

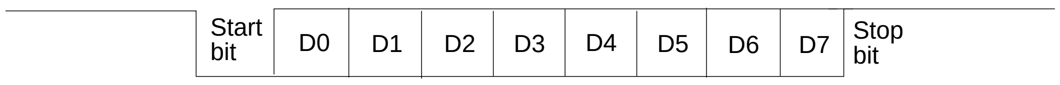

Figure 4.9: RS232 Serial Interface Protocol

When the line is idling, it stays in logic high state. When a byte is transmitted, the transmission begins with a start bit (logic zero) and continues with data bits (LSB first) and ends up with a stop bit (logic high). 10 bits are sent for each 8-bit byte frame.

4.8.2 Registers

UART registers, prefix UARTx_				
Reg	**Type**	**Reset**	**Abbrev**	**Description**
0xC028	r	0	STATUS[3:0]	Status
0xC029	r/w	0	DATA[7:0]	Data
0xC02A	r/w	0	DATAH[15:8]	Data High
0xC02B	r/w	0	DIV	Divider

Status UARTx_STATUS

A read from the status register returns the transmitter and receiver states.

UARTx_STATUS Bits		
Name	**Bits**	**Description**
UART_ST_FRAMERR	4	Framing Error (stop bit was 0)
UART_ST_RXORUN	3	Receiver overrun
UART_ST_RXFULL	2	Receiver data register full
UART_ST_TXFULL	1	Transmitter data register full
UART_ST_TXRUNNING	0	Transmitter running

UART_ST_FRAMERR is set at the time of stop bit reception. When reception is functioning normally, stop bit is always "1". If, however, "0" is detected at the line input at the stop bit time, UART_ST_FRAMERR is set to "1". This can be used to detect "break" condition in some protocols.

UART_ST_RXORUN is set if a received byte overwrites unread data when it is transferred from the receiver shift register to the data register, otherwise it is cleared.

UART_ST_RXFULL is set if there is unread data in the data register.

UART_ST_TXFULL is set if a write to the data register is not allowed (data register full).

UART_ST_TXRUNNING is set if the transmitter shift register is in operation.

Data UARTx_DATA

A read from UARTx_DATA returns the received byte in bits 7:0, bits 15:8 are returned as '0'. If there is no more data to be read, the receiver data register full indicator will be cleared.

A receive interrupt will be generated when a byte is moved from the receiver shift register to the receiver data register.

A write to UARTx_DATA sets a byte for transmission. The data is taken from bits 7:0, other bits in the written value are ignored. If the transmitter is idle, the byte is immediately moved to the transmitter shift register, a transmit interrupt request is generated, and transmission is started. If the transmitter is busy, the UART_ST_TXFULL will be set and the byte remains in the transmitter data register until the previous byte has been sent and transmission can proceed.

Data High UARTx_DATAH

The same as UARTx_DATA, except that bits 15:8 are used.

Divider UARTx_DIV

UARTx_DIV Bits		
Name	**Bits**	**Description**
UART_DIV_D1	15:8	Divider 1 (0..255)
UART_DIV_D2	7:0	Divider 2 (6..255)

The divider is set to 0x0000 in reset. The ROM boot code must initialize it correctly depending on the master clock frequency to get the correct bit speed. The second divider (D_2) must be from 6 to 255.

The communication speed $f = \frac{f_m}{(D_1+1)\times(D_2)}$, where f_m is the master clock frequency, and f is the TX/RX speed in bps.

4.8.3 Interrupts and Operation

Transmitter operates as follows: After an 8-bit word is written to the transmit data register it will be transmitted instantly if the transmitter is not busy transmitting the previous byte. When the transmission begins a TX_INTR interrupt will be sent. Status bit [1] informs the transmitter data register empty (or full state) and bit [0] informs the transmitter (shift register) empty state. A new word must not be written to transmitter data register if it is not empty (bit [1] = '0'). The transmitter data register will be empty as soon as it is shifted to transmitter and the transmission is begun. It is safe to write a new word to transmitter data register every time a transmit interrupt is generated.

Receiver operates as follows: It samples the RX signal line and if it detects a high to low transition, a start bit is found. After this it samples each 8 bit at the middle of the bit time (using a constant timer), and fills the receiver (shift register) LSB first. Finally if a stop bit (logic high) is detected the data in the receiver is moved to the reveive data register and the RX_INTR interrupt is sent and a status bit[2] (receive data register full) is set, and status bit[2] old state is copied to bit[3] (receive data overrun). After that the receiver returns to idle state to wait for a new start bit. Status bit[2] is zeroed when the receiver data register is read.

RS232 communication speed is set using two clock dividers. The base clock is the processor master clock. Bits 15-8 in these registers are for first divider and bits 7-0 for second divider. RX sample frequency is the clock frequency that is input for the second divider.

4.8.4 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for the UART:

UART receive is by default tied to the ROM monitor, which enables debugging via a serial cable using vsemu command line tool and IDE environment available from VLSI.

The default communication speed of the UART is 115200 bit/s with a 12 MHz crystal.

4.9 Universal Serial Bus Controller v1.0 2006-01-05

4.9.1 General

The Universal Serial Bus Controller handles USB 1.1 data traffic at a 12 Mbit/s signalling speed.

The USB device can handle traffic for the control endpoint (0) plus three input and output endpoints. Bulk, Isochronous and Interrupt transfer modes are supported at Full Speed (12 Mbit/s). The maximum packet size is 1023 bytes.

4 kilobytes of X data memory are used as the USB packet buffer: 2 KiB for incoming packets (X:0x2C00-0x2FFF) and 2 KiB for outgoing packets (X:0x3000-0x33FF). The input buffer is a ring buffer with incoming packets consisting of a start word and n data words. The output buffer has 16 possible start locations for outgoing packets at 128-byte (64-address) intervals (note that all data addressing in VS1000 is based on 16-bit words).

4.9.2 Registers

Universal Serial Bus Controller Registers

Address	**Register**	**Function**
0xC080	USB_CONFIG	USB Device Config
0xC081	USB_CONTROL	USB Device Control
0xC081	USB_STATUS	USB Device Status
0xC082	USB_RDPTR	Receive buffer read pointer
0xC083	USB_WRPTR	Receive buffer write pointer
0xC088	USB_EP_SEND0	EP0IN Transmittable Packet Info
0xC089	USB_EP_SEND1	EP1IN Transmittable Packet Info
0xC08A	USB_EP_SEND2	EP2IN Transmittable Packet Info
0xC08B	USB_EP_SEND3	EP3IN Transmittable Packet Info
0xC090	USB_EP_ST0	Flags for endpoints EP0IN and EP0OUT
0xC091	USB_EP_ST1	Flags for endpoints EP1IN and EP1OUT
0xC092	USB_EP_ST2	Flags for endpoints EP2IN and EP2OUT
0xC093	USB_EP_ST3	Flags for endpoints EP3IN and EP3OUT

USB_CONFIG - USB Device Config 0xC080

USB_CONFIG bits		
Name	**Bits**	**Description**
reset	15	Reset Active
dtogg-host	14	Reset value of host data toggle (set to 0)
dtogg-device	13	Reset value of device data toggle (set to 0)
debug12-11	12:11	Debug bits (set to 0)
dtogg-errctl	10	Data Toggle error control (set to 0)
reserved9	9	Reserved (set to 0)
rstusb	8	Reset receiver (set to 0)
usb-enable	7	Enable USB
usb-address	6:0	Current USB address

USB_CONTROL - USB Device Control 0xC081

USB_CONTROL bits		
Name	**Bits**	**Description**
USB_STF_BUS_RESET	15	Interrupt mask for bus reset
USB_STF_SOF	14	Interrupt mask for start-of-frame
USB_STF_RX	13	Interrupt mask for receive data
USB_STF_TX_READY	12	Interrupt mask for transmitter holding register empty
USB_STF_TX_EMPTY	11	Interrupt mask for transmitter empty (idle)
USB_STF_NAK	10	Interrupt mask for NAK packet sent to host
usb-configured	0	Configured. 0→1 transition loads dtogg-host and dtogg-device

Software should write "1" to usb-configured bit when completing the USB Chapter 9 Set_Configuration request. Setting this bit loads all device and host side data toggle registers with the defaults set at the dtogg-host and dtogg-device bits at the USB_CONFIG register. The dtogg-host and dtogg-device bits should normally always be "0".

VS1000A ROM does not use the USB interrupt.

USB_STATUS - USB Device Status 0xC082

USB_STATUS bits		
Name	**Bits**	**Description**
USB_STF_BUS_RESET	15	Bus reset occurred
USB_STF_SOF	14	Start-of-frame
USB_STF_RX	13	Receive data
USB_STF_TX_READY	12	Transmitter holding register empty
USB_STF_TX_EMPTY	11	Transmitter empty (idle)
USB_STF_NAK	10	NAK packet sent to host
USB_STF_SETUP	7	Setup packet received
USB_STM_LAST_EP	3:0	Endpoint number of last rx/tx transaction

The USB_STM_LAST_EP can be used mainly for debugging purposes, final software should be able to work without it.

USB_RDPTR - Receive buffer read pointer 0xC083

USB_RDPTR bits		
Name	**Bits**	**Description**
USB_RDPTR	15:0	Packet Read Pointer

This buffer marks the index position of the last word that the DSP has successfully read from the receive packet buffer. DSP should control this register and update the position after each packet it has read from the receive buffer. After reset this register is zero.

USB_WRPTR - Receive buffer write pointer 0xC084

USB_WRPTR bits		
Name	**Bits**	**Description**
USB_WRPTR	15:0	Packet Write Pointer

After a packet has been received from the PC, the USB hardware updates this pointer to the receive buffer memory. USB_WRPTR is index location of the next free word location in the USB receive buffer. When USB_RDPTR equals to USB_WRPTR, the packet input buffer is empty. After reset this register is zero.

USB_EP_SENDn - EPnIN Transmittable Packet Info 0xC088..0xC08B

USB_EP_SENDn bits		
Name	**Bits**	**Description**
txpkt-ready	15	Packet ready for transmission
start-addr	13:10	Starting location of packet
length	9:0	Length of packet in bytes (0..1023)

When the DSP has written a packet into the transmit buffer, that is ready to be transmitted to the PC by an endpoint, the DSP signals the USB firmware by setting the value of the USB_EP_SENDn register of the endpoint that should transmit the packet (USB_EP_SEND0 for endpoint 0, USB_EP_SEND1 for endpoint 1 etc).

The txpkt-ready bit should be set to "1" by the DSP. When the packet information (not contents) is loaded to the internal Transmit Holding Register of the endpoint, txpkt-ready bit is set to "0" by the hardware. Note that this does not indicate that the packet is sent to the PC, merely that it is ready for sending when the PC next requests "IN" data for that endpoint. Scanning the txpkt-ready bit merely allows software to prepare the next packet to be sent even before the previous packet has been sent to the PC.

The start-addr field is index to a 64-word boundary in the transmit buffer memory area. The actual memory location that start-addr corresponds to is calculated by:

```
packet start address = USB_SEND_MEM + (start-addr × 64)
```

which in VS1000A corresponds to address X:0x3000 for start-addr=0, X:0x3040 for start-addr=1 etc.

USB_EP_STn - Flags for endpoints EPnIN and EP0nUT 0xC090..0xC093

USB_EP_STn bits		
Name	**Bits**	**Description**
EPnOUT (PC → Device) endpoint (0 .. 3) flags		
out-type	15:14	00=bulk 01=interrupt 11=isochronous
out-enable	14:13	1=enabled 0=disabled
out-forcestall	12	Force STALL
out-stall-sent	11	At least 1 STALL sent
reserved	10:8	Set to 0
EPnIN (Device → PC) endpoint (0 .. 3) flags		
in-type	7:6	00=bulk 01=interrupt 11=isochronous
in-enable	5	1=enabled 0=disabled
in-forcestall	4	Force STALL
in-stall-sent	3	At least 1 STALL sent to PC
in-nak-sent	2	At least 1 NAK sent to PC
in-xmit-empty	1	Transmitter empty
reserved	0	Set to 0

4.9.3 Receiving Packets from PC (EP0OUT, EP1OUT, ... , EP3OUT)

The USB hardware handles all necessary token (ACK, NAK, IN, OUT, SETUP, STALL) sending and receiving. The software sees only the data packet contents plus some state information about the sent tokens.

Reception

All received packets for all endpoints arrive to the same 2 KiB (1 KiW) ring buffer memory in X address space. This maximizes memory usage efficiency, but leads to one important side-effect: The USB specification dictates that an incoming SETUP transfer to the control endpoint must be the first packet to be processed at all times.

For instance the PC might issue a SETUP control request to the control endpoint before the software has had time to process a data packet that has previously arrived to a data endpoint. In such a case, the software should ignore the pending data and handle the SETUP packet instead.

For achieving this functionality, the hardware can test the USB_STF_SETUP bit at the USB_CONTROL register. If it is "1", all packets until the last received SETUP packet need to be truncated and the last SETUP packet processed. A reasonably fast USB implementation should be able to achieve this without problems, but delays of several milliseconds (such as for sending debug messages etc) can cause problems with this clause, which result in "random hang-ups" of the USB communication with the PC. If care is taken to process the packets in the correct order, most (if not all) USB transactions can perfectly well cope with delays of several seconds. In practice the PC waits patiently for several seconds if the data you send is "correct," e.g. what the PC expects, but very quickly responds to any unexpected data by issuing a bus reset.

Software can detect a received packet by scanning the USB_RDPTR and USB_WRPTR registers. When their values differ, there is a packet ready for processing in the input buffer.

USB_RDPTR points now to a header word. The actual packet data words are in the buffer memory after the header word. The packet header word has the following structure:

Packet header word bits		
Name	**Bits**	**Description**
crc-err	15	1=CRC error detected
setup	14	1=SETUP packet, 0=DATA packet
endpoint	13:10	Endpoint to which the packet is addressed to
pktlength	9:0	Length of packet in bytes

This is immediately followed by (pktlength+1)/2 data words, MSB first.

A quick routine can access the contents directly in the buffer memory, or choose to copy the packet contents to another location in memory. In either case, the software should update the value of USB_RDPTR, indicating that the packet is no longer needed.

The USB hardware automatically NAK's incoming data packets if there is less than 40 words space left in the buffer memory. In this situation the hardware still accepts SETUP packets. If receiving a packet would cause the USB_WRPTR to be overrun by USB_RDPTR, e.g. there is no more room for even the SETUP packet, even the SETUP packets are NAK'ed.

Sending Packet to PC (EP0IN, EP1IN, ... , EP3IN)

To send a USB packet, software must prepare the packet to the transmit buffer area, starting at a 64-word boundary. The data is to be stored in Big Endian format, e.g. the first byte to be sent should be in the most significant 8 bits of the first word. Next, software should load the USB_EP_SEND register of the chosen endpoint with the start location selector and size of the transmittable packet (in bytes). The most significant bit of the register (txpkt-ready) should be set to "1".

When the internal Transmit Holding Register for the endpoint is ready, the value of the USB_EP_SEND register is loaded to the internal Transmit Holding Register and txpkt-ready bit of the USB_EP_SEND register is set to "0". This indicates that the packet is queued for transmission and the USB_EP_SEND register can be loaded with information about the next sendable packet (if any).

To get information about when the packet has actually been transmitted to the PC, the Transmitter Idle (in-xmit-empty) bit of the endpoint's USB_EP_ST register can be polled (or the corresponding interrupt used).

How to know that the PC is expecting data

During software development, when protocol matters can be still somewhat unclear, it is sometimes difficult to know when the PC actually is expecting you to send a packet to some endpoint. In the USB hardware there is a feature to assist in finding out this information: the endpoint's in-nak-sent bit of the endpoint's USB_EP_ST register. Using this bit can avoid a common pitfall: loading a transmitter register with packet that is never actually requested by a PC. That would cause the packet information to remain in the transmitter register (until next USB reset), which again would cause the packet to be sent as an answer to the **next** request of the PC, causing unexpected results.

Stalling

STALL is a special condition on the USB bus, which more or less states, that "I can't handle this data packet now nor in the future". For example when the software needs to STALL reception of data from PC, software should set out-forcestall to "1" and out-stall-sent to "0". The hardware will then wait for the next OUT token from the PC and respond with STALL token. Bit out-stall-sent indicates that a STALL token has been successfully transmitted to the PC.

A more common case of stalling regards the handling of control requests (SETUP messages sent to the control endpoint). In case of receiving an unsupported request, the device should respond wit a stall. Since the control endpoint must remain open for the next request for the PC and stalling a control request should be a rare event, a possible way to handle this is:

- In the USB_EP_ST0 register, set out-forcestall to "1" and out-stall-sent to "0"
- Busy loop until out-stall-sent is "1" OR a USB reset occurs OR a time-out occurs
- Set out-forcestall to "0"

In a normal case this would send a single STALL to the control endpoint and leave the endpoint open for the next request.

If an endpoint's (other than 0) Halt feature is set (USB Chapter 9 standard request), the endpoint should be stalled (forcestall set to "1").

Mass storage class device can use STALL to end a bulk transfer [Axelson, J.: USB Mass Storage].

4.9.4 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for the USB:

- Endpoint 0: USB Standard Requests (USB Chapter 9 functionality)
- Endpoint 1 OUT: USB Speakers
- Endpoint 2 OUT: Mass Storage Class (PC → VS1000)
- Endpoint 3 IN: Mass Storage Class (VS1000 → PC)

Depending on the state of GPIO0:6 during boot-up, the descriptors sent to the PC select either Audio or Mass Storage functionality.

Augmenting the ROM functionality

Changing only the descriptors is easy since the descriptors are accessed via a descriptor pointer table in RAM that consists of 6 pointers to X memory:

USB.descriptorTable entries	
Index	**Function**
[0]	String Descriptor 0 (Language Index)
[1]	String Descriptor 1 (Manufacturer: "VLSI")
[2]	String Descriptor 2 (Model: "VS1000")
[3]	String Descriptor 3 (Serial Number: "100010001003")
[4]	Device Descriptor
[5]	Configuration Descriptors

Because the configuration descriptor is actually a set of descriptors, its size is stored in USB.configurationDescriptorSize. For other descriptors, the size is taken from the descriptor itself.

Note: A good storage driver should overwrite the serial number string descriptor with a unique one. For a NAND flash, this could be done easily in the first sector's optional boot code. Since the USB.descriptorTable default values are loaded at each USB init (attacj), the most straightforward way to do this would be to hook the DecodeSetupPacket() function to load USB.descriptorTable[3] and call the RealDecodeSetupPacket() in ROM.

USB-related software hooks are:

- void USBHandler() - USB task handler
- void DecodeSetupPacket() - handles SETUP packets to EP0OUT
- void MSCPacketFromPC() - handles mass storage class command packets
- void ScsiTaskHandler() - handles (pending) disk operations

Hooking means replacing a ROM function with a RAM function by setting the hook vector address. This is normally used to augment or replace functionality of the ROM code. In most cases, the original ROM function can be called after handling some special case in the RAM function. The ROM functions are called by using a function name with prefix "Real".

Hooking: Example

This example augments the USBHandler to blink LED 2 if there is an uncomplete disk write operation.

```
void USBHandler(void);
void RealUSBHandler(void);

void MyUSBHandler(void) {
  if (SCSI.mapperNextFlushed == -1) {
    USEX(GPIO1_ODATA) &= ~LED2;
  } else {
    USEX(GPIO1_ODATA) ^= LED2;
  }
  RealUSBHandler(); /* Call original ROM function */
}
```

The hook can be loaded by calling

```
SetHookFunction((u_int16)USBHandler, MyUSBHandler);
```

or by setting the hook vector directly in the boot record (via a Set X Memory directive).

Used memory areas

The USB transmitting routines in VS1000A ROM are limited to transmitting packets of max. 64 bytes. Only the first 512 bytes (addresses X:0x3000-0x30FF) of transmit packet memory is used, leaving 1536 bytes (768 words) of X memory (addresses X:3100-0x33ff) free for other uses. However, in future revisions of the chip (VS1000B etc..) this memory may not be available.

4.10 Watchdog v1.0 2002-08-26

4.10.1 General

The watchdog consist of a watchdog counter and some logic. After reset, the watchdog is inactive. The counter reload value can be set by writing to WDOG_CONFIG. The watchdog is activated by writing 0x4ea9 to register WDOG_RESET. Every time this is done, the watchdog counter is reset. Every 65536'th clock cycle the counter is decremented by one. If the counter underflows, it will activate vsdsp's internal reset sequence.

Thus, after the first 0x4ea9 write to WDOG_RESET, subsequent writes to the same register with the same value must be made no less than every 65536×WDOG_CONFIG clock cycles.

Once started, the watchdog cannot be turned off. Also, a write to WDOG_CONFIG doesn't change the counter reload value.

After watchdog has been activated, any read/write operation from/to WDOG_CONFIG or WDOG_DUMMY will invalidate the next write operation to WDOG_RESET. This will prevent runaway loops from resetting the counter, even if they do happen to write the correct number. Writing a wrong value to WDOG_RESET will also invalidate the next write to WDOG_RESET.

Reads from watchdog registers return undefined values.

4.10.2 Registers

Watchdog, prefix WDOG_				
Reg	**Type**	**Reset**	**Abbrev**	**Description**
0xC020	w	0	CONFIG	Configuration
0xC021	w	0	RESET	Clock configuration
0xC022	w	0	DUMMY[-]	Dummy register

4.10.3 VS1000A ROM code usage

The ROM code in VS1000A has the following usage for the watchdog: