Application Note

AN_289

FT51A Programming Guide

Version 1.1

Issue Date: 2016-09-19

This document provides a guide for using FT51A firmware libraries supplied by FTDI and writing applications.

Future Technology Devices International Limited (FTDI)
Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
Web Site: http://ftdichip.com

Table of Contents

1 Introduction

This guide documents the registers and internal architecture of the FT51A. It also covers the firmware libraries and samples provided for the FT51A by FTDI.

1.1 Overview

The FT51A series of devices provides a USB device interface, a built-in USB hub and an 8051 compatible microcontroller. The 8051 compatible component is referred to as the 'core'.

There is 16kB of program storage in MTP (Multiple Time Programmable) memory, 16kB of Shadow RAM (from where code is run), 8kB of data RAM and 128 bytes of internal RAM.

Details of the device are fully documented in the FT51A Series Datasheets which can be obtained from the FTDI website. http://www.ftdichip.com/Products/ICs/FT51.html.

Additionally there are the following hardware interfaces:

- GPIO
- UART
- PWM
- SPI Master and Slave
- I2C Master and Slave
- FT245 Parallel
- ADC
- Additional Timers

The FT51A has an internal USB Full Speed device controller that is register compatible with an FT122. An internal on-chip USB hub can optionally be enabled to allow a single downstream port from the FT51A.

1.2 Features

The firmware libraries for the FT51A have the following features:

- Abstracted access to USB functions for simple implementation of device emulation.
- Functions for performing basic access to ADC, SPI Master, I2C Master, I2C Slave and UART hardware interfaces.
- Macros and definitions for hardware related features.
- Additional libraries for LCD devices, DFU (Device Firmware Update), TMC (Test and Measurement).

The use of the firmware libraries is shown in the sample applications:

- DFU Firmware update,
- Keyboard and Mouse demos,
- Test and Measurement Class,
- Interfacing to FT800 demos,
- Interfacing to LCD screen.

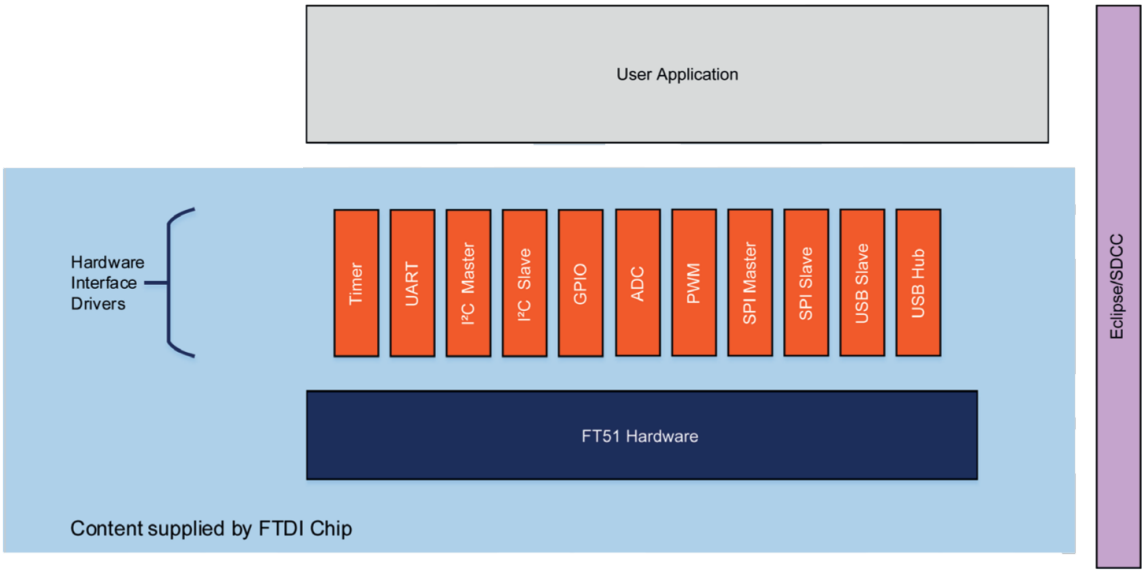
1.3 Scope

This guide is intended for developers who are creating applications, extending FTDI provided applications or implementing example applications for the FT51A.

In the reference of the FT51A, an "application" refers to firmware that runs on the FT51A; "libraries" are source code provided by FTDI to help users access specific hardware features of the chip.

The FT51A Tools are currently only available for Microsoft Windows, and are tested on Windows 7 and Windows 8.1.

The following diagram shows the overall system structure:



2 Hardware Reference

The FT51A has an 8051 compatible core. There are extended Special Function Registers (SFRs) to enable access to the registers of all the peripherals and modules. Certain registers are accessed directly through SFRs and others are accessed through I/O ports addressed though SFRs.

The SFR map is shown in Table 2.1.

SFRs	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x80	P0	SP	DPL0	DPH0	DPL1	DPH1	DPS	PCON
0x88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	
0x90	P1	EIF			IO_DATA_9			
0x98	SCON0	SBUF0	IO_ADDR_0 H	IO_ADDR_0 L	IO_DATA_0	IO_ADDR_1 H	IO_ADDR_1 L	IO_DATA_1
0xA0	P2							
0xA8	IE	IO_ADDR_2 H	IO_ADDR_2 L	IO_DATA_2				
0xB0	P3	IO_ADDR_3 H	IO_ADDR_3 L	IO_DATA_3				
0xB8	IP	IO_ADDR_4 H	IO_ADDR_4 L	IO_DATA_4				
0xC0								
0xC8	T2CON	T2IF	RCAP2L	RCAP2H	TL2	TH2		
0xD0	PSW	IO_ADDR_5 H	IO_ADDR_5 L	IO_DATA_5				
0xD8		IO_ADDR_6 H	IO_ADDR_6 L	IO_DATA_6				
0xE0	ACC	IO_ADDR_7 H	IO_ADDR_7 L	IO_DATA_7				
0xE8	EIE	STATUS						
0xF0	B	I2CSOA	I2CSCR	I2CSBUF	I2CMSA	I2CMCR	I2CMBUF	I2CMTP
0xF8	EIP	IO_ADDR_8 H	IO_ADDR_8 L	IO_DATA_8	FT122_CMD	FT122_DATA	IO_ADDR_9 H	IO_ADDR_9 L

Table 2.1 FT51A SFR Map

2.1 Hardware Access

The SFRs contain registers to allow direct access to the USB Full Speed device controller, I2C Master and I2C Slave peripherals.

There are 10 sets of I/O ports that permit access to the registers of the ADC, PWM, SPI Master, SPI Slave, UART FTDI, 245 FIFO, DMA, Timers, Watchdog and IOMUX.

Table 2.2 summarises the methods required to access each module.

Name	Method
ADC	I/O
PWM	I/O
SPI Master	I/O
SPI Slave	I/O
I2C Master	SFR
I2C Slave	SFR
UART FTDI	I/O
UART DCD	SFR
245 FIFO	I/O
DMA Controller	I/O

Name	Method
GPIO FTDI	I/O
GPIO	SFR
AIO	SFR
Debugger	SFR
Device Control	I/O
IOMUX	I/O
Timers 0, 1, 2	SFR
Timers A, B, C, D	I/O
Watchdog FTDI	I/O
Watchdog 8051	SFR
USB Device Hub Port	SFR

Table 2.2 FT51A Peripherals

2.1.1 Registers Accessed by SFR

For peripherals and modules addressed directly through the SFRs, the SDCC compiler provides a "__sfr" keyword to allow their registers to be used like variables. For example, specify `__sfr __at (0x80) P0;` to allow access to port 0 via P0 variable. Refer to the SDCC documentation for further information.

2.1.2 Registers Accessed through I/O Ports

To access a register via the I/O port method, the address of the register has to first be written to one of the `IO_ADDR_x` SFRs; then the data can be read from, or written to, the matching `IO_DATA_x` SFR.

The I/O port address space is 9 bits, 0x000 to 0x1FF. Therefore the `IO_ADDR_x` SFRs have a high and a low byte. The high byte is normally zero because only the IO Cell Controller is located above the address 0xFF.

The SFRs contain 10 separate I/O ports. Writing to the address register for one port does not interfere with an address written previously for a different port.

Example macros for writing and reading I/O ports are presented below:

```
#define WRITE_IO_REG(address, data)                                  \
do                                                                   \
{                                                                    \
    IO_ADDR_9_H = (unsigned char)((unsigned int)(address) >> 8); \
    IO_ADDR_9_L = (unsigned char)(address);                          \
    IO_DATA_9 = (data);                                              \
}                                                                    \
while (0)

#define READ_IO_REG(address, data)                                   \
do                                                                   \
{                                                                    \
    IO_ADDR_9_H = (unsigned char)((unsigned int)(address) >> 8); \
    IO_ADDR_9_L = (unsigned char)(address);                          \
    (data) = IO_DATA_9;                                              \
}                                                                    \
while (0)
```

2.1.3 Register Descriptions

The hardware and peripheral descriptions in this chapter include register maps which define the initial state of the registers, their behaviour and provide a description of the bit fields.

Bit type is the behaviour of the bit when accessed. It can be read only, read and write or write to clear. The mnemonics used in this chapter are defined in Table 2.3.

Type	Definition
R	Read Only
R/W	Read/Write
W1C	Write '1' to Clear
RFU	Reserved for Future Use
W1T	Write '1' to Trigger, Reads as '0'

Table 2.3 Register Bit Type Definitions

The initial state of each register is given in the Reset column of the register descriptions.

2.2 Device Control Registers

These registers control and provide status on the FT51A device. They are collectively referred to as the 'top-level' registers.

Address	Register Name	Description
0x00	DEVICE_CONTROL_REGISTER	Device Control Registers
0x01	SYSTEM_CLOCK_DIVIDER	System Clock Divider
0x02	TOP_USB_ENABLE	USB Top-Level Control Register
0x03	PERIPHERAL_INT0	Peripheral Interrupt Status 0
0x04	PERIPHERAL_IEN0	Peripheral Interrupt Enable 0
0x05	PERIPHERAL_INT1	Peripheral Interrupt status 1
0x06	PERIPHERAL_IEN1	Peripheral Interrupt Enable 1
0x09	PIN_CONFIG	Debugger State and BDC mode
0x2B	MTP_CONTROL	MTP Memory Control
0x2C	MTP_ADDR_L	MTP Lower Address
0x2D	MTP_ADDR_U	MTP Upper Address
0x2E	MTP_PROG_DATA	MTP Write Data
0x36	MTP_CRC_CTRL	16-bit CRC enable of MTP memory
0x37	MTP_CRC_RESULT_L	16-bit CRC Result Lower Byte
0x38	MTP_CRC_RESULT_U	16-bit CRC Result Lower Byte
0x34	PIN_PACKAGE_CONFIG	Device package Information
0x39	TOP_SECURITY_LEVEL	Device Security Status Register

Table 2.4 Device Control Register Addresses

In addition to the standard interrupts generated by the 8051 core, the FT51A supports other modules and peripherals as sources. These interrupts can be queried in a hierarchical manner.

Once the top-level interrupt source is known by reading `PERIPHERAL_INT0` or `PERIPHERAL_INT1` the interrupt status registers in the pertinent module can then be investigated to determine the low-level interrupt source.

To clear an interrupt, first the low-level interrupt with the module should be cleared, followed by the high-level interrupt in the `PERIPHERAL_INT0` or `PERIPHERAL_INT1` registers.

Interrupt handler routines may need to check if a particular interrupt source is enabled in `INTERRUPT_EN_0` or `PERIPHERAL_IEN1` before acting on the interrupt.

To perform a reset of the entire device the `top_soft_reset` bit in `DEVICE_CONTROL_REGISTER` must be set, followed by the `reset_8051` bit in the `SYSTEM_CLOCK_DIVIDER` register.

2.2.1 DEVICE_CONTROL_REGISTER

Bit Position	Bit Field Name	Type	Reset	Description
7..2	RFU	R	0	Reserved
1	`top_dev_en`	R/W	0	This bit MUST be set to allow write access to all top-level registers.
0	`top_soft_reset`	R/W	0	When set will cause a reset of the entire device. This bit will always read as zero

Table 2.5 Device Control Register

The Device Control register provides top-level write enable and reset functions for all top-level registers on the FT51A device. This encompasses only the registers described in this chapter and not any 8051 core registers or other module's registers.

Write access to the top-level registers is enabled by setting the `top_dev_en` bit to 1. Clearing this bit will disable write access.

To reset all top-level registers, a 1 is written to the `top_soft_reset` bit. The module clears this bit when a reset is performed and will therefore always read as '0'.

2.2.2 SYSTEM_CLOCK_DIVIDER

Bit Position	Bit Field Name	Type	Reset	Description
7..5	RFU	R	0	Reserved
4	`reset_8051`	R/W	0	Set to reset the 8051 core. This will cause the 8051 state and registers to be reset. The program counter will return to its RESET value 0x0000. All other modules and peripherals except the top-level registers will be reset.
3	`system_stop_request`	R/W	0	For reduced power consumption. When set will stop all internal clocks and place the chip in a low power state. Alternatively use PCON SFR (more below).
2..1	`clk_sys_divisor`	R/W	0	<table><tr><th>1</th><th>0</th><th>Clock division</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>1</td><td>0</td><td>4</td></tr><tr><td>1</td><td>1</td><td>8</td></tr></table>
0	hub_suspend_en	R/W	0	Allow the hub to enter suspend mode.

Table 2.6 System and Clock Divider Register

Note: When requesting a low power state and to obtain the lowest possible current consumption the User must ensure all pad IO controls have no pull ups or downs enabled, and are configured as an input. Also ensure that the external VCC3V3 is not under any load conditions.

Note: When running with clock division set to divide-by-8 certain functions are affected: debugger access is NOT possible; UART cannot run at 3M BAUD. A minimum of divide-by-4 is advised for such operations.

Note: Setting PCON SFR bit 0, so called Power Management Mode (PMM), reduces power consumption by externally dividing the clock signal provided to the microcontroller, causing it to operate at a reduced speed. When PMM is invoked, the external pin called PMM is set into logic 1. It signalizes to external divider that CLK frequency should be divided by 256. Note that all internal functions, on-board timers (including serial port baud rate generation), watchdog timer, and software timing loops will run at the reduced speed.

PMM is entered and exited by setting the PMM bit (PCON.0). In addition, use of the switchback feature is possible to affect a return from PMM to the full speed mode. This allows both hardware and software to cause an exit from PMM. It is the responsibility of the software to test for UART activity before attempting to change speed, as a modification of the clock divider bits during a UART operation will corrupt the data.

The switchback feature allows a system to burst to a faster mode when required by an external event. Enable this feature by setting the PCON bit 2, a qualified interrupt (interrupt which has occurred and been acknowledged) or serial port reception or transmission cause the microcontroller to return to full speed mode. An interrupt must be enabled and not blocked by a higher priority interrupt. Software should manually return the microcontroller to PMM after the event is completed. The following sources can trigger the Switchback:

- external interrupt 0/1,
- serial start bit detected, UART,
- transmit buffer loaded, UART,
- reset,
- external reset.

2.2.3 TOP_USB_ENABLE

Bit Position	Bit Field Name	Type	Reset	Description
7..6	RFU	R	0	Reserved
5	hub_compd_dev	R/W		Controls the way the hub module identifies itself during enumeration : 1 – Hub is part of a compound device 0 – Hub is not part of a compound device
4	hub_remote_wakeup_en	R/W	0	1- Enable remote wakeup: Hub will respond to a host get_status command with an ACK 0 - Disable remote wakeup: Hub will respond to a host get_status command with a STALL
3	hub_stsnzdatahsk	R/W	0	On receipt of a non-zero-length data packet, device will: 1 – Hub returns a STALL 0 – Hub returns an ACK
2	hub_ext_localpwrsrc	R/W		The setting of this bit is used as a power status flag which is returned in response to a Host Get_Status command : 1 – External power source 0 – Bus powered
1	hub_enable	R/W	0	Write to this bit to enable or disable the Hub: 1 – Hub enabled 0 – Hub disabled
0	ft122_enable	R/W	0	Write to this bit to enable or disable USB functionality: 1 – USB enabled 0 – USB disabled

Table 2.7 USB Control Register

2.2.4 PERIPHERAL_INT0

Bit Position	Bit Field Name	Type	Reset	Description
7..5	RFU	R	0	Reserved
4	dma3_irq	R/W1C	0	Set when the memory contents have been successfully copied. Write '1' to clear interrupt.
3	dma2_irq	R/W1C	0	Set when the memory contents have been successfully copied. Write '1' to clear interrupt.
2	dma1_irq	R/W1C	0	Set when the memory contents have been successfully copied. Write '1' to clear interrupt.
1	dma0_irq	R/W1C	0	Set when the memory contents have been successfully copied. Write '1' to clear interrupt.
0	watchdog_irq	R/W1C	0	Set when a watchdog RESET is generated after a timeout. Write '1' to clear interrupt.

Table 2.8 Interrupt Status 0 Register

2.2.5 PERIPHERAL_IEN0

Bit Position	Bit Field Name	Type	Reset	Description
7..5	RFU	R	0	Reserved
4	dma3_irq_ien	R/W	0	Set to enable the dma3 interrupt.
3	dma2_irq_ien	R/W	0	Set to enable the dma2 interrupt.
2	dma1_irq_ien	R/W	0	Set to enable the dma1 interrupt.
1	dma0_irq_ien	R/W	0	Set to enable the dma0 interrupt.
0	watchdog_irq_ien	R/W	0	Set to enable the watchdog reset interrupt.

Table 2.9 Interrupt Enable 0 Register

2.2.6 PERIPHERAL_INT1

Bit Position	Bit Field Name	Type	Reset	Description
7	fifo_245_irq	R/W1C	0	Set when the 245 FIFO has generated an interrupt. Write '1' to clear interrupt.
6	timer_irq	R/W1C	0	Set when the TIMER has generated an interrupt. Write '1' to clear interrupt.
5	pwm_irq	R/W1C	0	Set when the PWM has generated an interrupt. Write '1' to clear interrupt.
4	spi_slave_irq	R/W1C	0	Set when the SPI slave has generated an interrupt. Write '1' to clear interrupt.
3	spi_master_irq	R/W1C	0	Set when the SPI master has generated an interrupt. Write '1' to clear interrupt.
2	uart_irq	R/W1C	0	Set when the UART has generated an interrupt. Write '1' to clear interrupt.
1	io_cell_controller_irq	R/W1C	0	Set after the completion of an ADC conversion. Write '1' to clear interrupt.
0	ft122_irq	R/W1C	0	Set when the USB has generated an interrupt after a timeout. Write '1' to clear interrupt.

Table 2.10 Interrupt Status 1 Register

2.2.7 PERIPHERAL_IEN1

Bit Position	Bit Field Name	Type	Reset	Description
7	`fifo_245_irq_ien`	R/W	0	Set to enable the 245 FIFO interrupt.
6	`timer_irq_ien`	R/W	0	Set to enable the TIMER interrupt.
5	`pwm_irq_ien`	R/W	0	Set to enable the PWM interrupt.
4	`spi_slave_irq_ien`	R/W	0	Set to enable the SPI_SLAVE interrupt.
3	`spi_master_irq_ien`	R/W	0	Set to enable the SPI_MASTER interrupt.
2	`uart_irq_ien`	R/W	0	Set to enable the UART interrupt.
1	`io_cell_controller_irq_ien`	R/W	0	Set to enable the ADC interrupt.
0	`ft122_irq_ien`	R/W	0	Set to enable the USB interrupt.

Table 2.11 Interrupt Enable 1 Register

2.2.8 PIN_CONFIG

Bit Position	Bit Field Name	Type	Reset	Description
7..1	RFU	R	0	Reserved
0	`vbus_detect_mode`	R/W	0	When set shall enable Battery Charge Detection mode.

Table 2.12 Pin Config Register

2.2.9 MTP_CONTROL

The MTP area can be written with program code that is copied into the Shadow RAM at power-on or reset.

Bit Position	Bit Field Name	Type	Reset	Description
7	copy_mtp_2_ram	R/W	0	When set to 1 this shall copy the contents of MTP block in to Shadow RAM. The core is held in RESET and code will start from address 0x0000 on completion of the copy. The bit is cleared on completion of the copy therefore the bit shall always read as zero.
6	flash_mtp_mem	R/W	0	When set to 1 this shall copy the contents of the Shadow RAM in to MTP memory. The bit is cleared on completion of the copy.
5	mtp_byte_prog_done	R1C	0	Set when MTP BYTE write has completed (See registers 0x2C, 0x2D, 0x2E). Cleared upon reading. Note: This bit does NOT indicate completion of a copy to MTP (see bit 6)
4	mtp_mem_wr_failure	R	0	Set when a write fail occurs. A fail can occur if the MTP byte write fails to update to the new value within a set programming time defined internally.
3..0	RFU	R	0	Reserved

Table 2.13 MTP Control Register

When the `copy_mtp_2_ram` operation is performed it is advised to read the `mtp_byte_prog_done` register flag in `MTP_CONTROL` for completion. Then the status of the write can then be read from the `mtp_mem_wr_failure` bit.

2.2.10 MTP_ADDR_L, MTP_ADDR_U and MTP_PROG_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	Data	R/W	0	Low byte of MTP Address

Table 2.14 MTP Address (Lower) Register

Bit Position	Bit Field Name	Type	Reset	Description
7..6	Reserved	R/W	0	
5..0	Data	R/W	0	High bytes of MTP Address

Table 2.15 MTP Address (Upper) Register

Bit Position	Bit Field Name	Type	Reset	Description
7..0	Data	R/W	0	

Table 2.16 MTP Data Register

The `MTP_ADDR_L`, `MTP_ADDR_U` and `MTP_PROG_DATA` registers allow byte write access to the MTP.

The address registers should be written to with the address in the MTP to be modified. A write to the `MTP_PROG_DATA` register will initiate the MTP byte write sequence.

It is advised to read the `mtp_byte_prog_done` register flag in `MTP_CONTROL` for completion. Then the status of the write can then be read from the `mtp_mem_wr_failure` bit.

Byte programming the MTP cannot be performed on the 63 bytes at 0x3FC0 to 0x3FFE. These are protected. However, writing to byte 0x3FFF is permitted. See `TOP_SECURITY_LEVEL` register.

2.2.11 MTP_CRC_CTRL, MTP_CRC_RESULT_L and MTP_CRC_RESULT_U

Bit Position	Bit Field Name	Type	Reset	Description
7..1	RFU	R	0	Reserved
0	`CRC_EN`	R/W		When set shall perform a CRC of the MTP array (not including top 64 bytes) and returns the 16-bit result into the MTP_CRC_RESULT registers.

Table 2.17 MTP CRC Control Register

Bit Position	Bit Field Name	Type	Reset	Description
7..0	Data	R/W	0	LOWER byte of 16-bit CRC result.

Table 2.18 MTP CRC Result (Lower) Register

Bit Position	Bit Field Name	Type	Reset	Description
7..0	Data	R/W	0	UPPER byte of 16-bit CRC result.

Table 2.19 MTP CRC Result (Upper) Register

2.2.12 PIN_PACKAGE_CONFIG

There are 3 package pin configurations available, 48/44, 32 or 28 pins. This is a read-only register that encodes the package type.

Bit Position	Bit Field Name	Type	Reset	Description
7..6	Pin Configuration	R	0	<table><tr><th>7</th><th>6</th><th>Package pins</th></tr><tr><td>0</td><td>0</td><td>28</td></tr><tr><td>0</td><td>1</td><td>RFU</td></tr><tr><td>1</td><td>0</td><td>32</td></tr><tr><td>1</td><td>1</td><td>48/44</td></tr></table>
5..0	RFU	R	0	Reserved

Table 2.20 Pin Package Type Register

2.2.13 TOP_ SECURITY_LEVEL

There are 3 security levels built into the FT51A.

Security Level 0 (SL0) – No security. Reads and writes possible from Program Memory.

Security Level 1 (SL1) – Reads are blocked via debugger to certain address sectors.

Security Level 2 (SL2) – Debugger is completely disabled.

This is a read only register and reflects the security level of the chip.

Bit Position	Bit Field Name	Type	Reset	Description
7..5	RFU	R	0	Reserved
4	`Global_Bit`	R	0	Set to protect sector: Start Address: 0x0000 End Address: 0x3FBF 0= Sector level security applied as per bits[3:0]. 1= All sectors are SL2. Security bits[3:0] are overridden.
3	`Sector_4`	R	0	Set to protect sector: Start Address: 0x3000 End Address: 0x3FBF 0= Sector level is SL0. 1= Sector level is SL1.
2	`Sector_3`	R	0	Set to protect sector: Start Address: 0x2000 End Address: 0x2FFF 0= Sector level is SL0. 1= Sector level is SL1.
1	`Sector_2`	R	0	Set to protect sector: Start Address: 0x1000 End Address: 0x1FFF 0= Sector level is SL0 1= Sector level is SL1.
0	`Sector_1`	R	0	Set to protect sector: Start Address: 0x0000 End Address: 0x0FFF 0= Sector level is SL0 1= Sector level is SL1.

Table 2.21 Top Level Security Register

The security level is set in the top MTP byte at address 0x3FFF. This allows the level of security for the chip to be set.

The value stored in MTP has the same bitmap as this register. It is copied into this register after an external reset or power on.

The top byte can only be written through a `flash_mtp_mem` operation from the `MTP_CONTROL` register. This copies the entire Shadow RAM into MTP.

The security bits are categorised as write forward, as once a particular level has been set it is not possible to go back to a lower security level.

If the `flash_mtp_mem` operation is used then Shadow RAM byte 0x3FFF must contain the security requirements for the chip. The address 0x3FFF should be reserved for this purpose.

Do not modify the values stored at Shadow RAM addresses 0x3FF8, 0x3FF9 and 0x3FFA. These contain factory programmed values.

2.3 SPI Master

The Serial Peripheral Interface Bus is an industry standard communications interface. Devices communicate in Master / Slave mode, with the Master initiating the data transfer.

The SPI Master module has seven signals:

- Clock
- 4 Slave Select lines (numbered 0 to 3)
- MOSI (master out – slave in)
- MISO (master in – slave out).

The SPI Master protocol by default does not support any form of handshaking and the only available mode is unmanaged. Data is clocked out of the Master and clocked in from the Slave simultaneously.

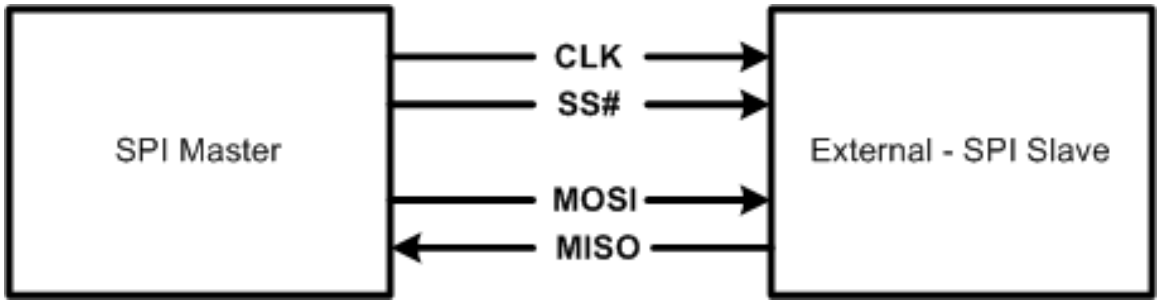


Figure 2.1 SPI Master Schematic Diagram

The SPI interface has 4 unique modes of clock phase (CPHA) and clock polarity (CPOL), known as Mode 0, Mode 1, Mode 2 and Mode 3. Table 2.29 summarizes these modes.

The registers associated with the SPI Master are outlined in Table 2.22. These are accessed using `IO_ADDR_x` SFRs to set the address, and the corresponding `IO_DATA_x` SFR to read and write the data.

I/O Address	Register Name	Description
0x50	SPI_MASTER_CONTROL	SPI Master top level control register
0x51	SPI_MASTER_TX_DATA	Transmit data register
0x52	SPI_MASTER_RX_DATA	Receive data register
0x53	SPI_MASTER_IEN	Interrupt Enable register
0x54	SPI_MASTER_INT	Interrupt Status register
0x55	SPI_MASTER_SETUP	Setup register
0x56	SPI_MASTER_CLK_DIV	Clock divisor register
0x57	SPI_MASTER_DATA_DELAY	Data delay register
0x58	SPI_MASTER_SS_SETUP	Slave select setup register
0x59	SPI_TRANSFER_SIZE_L (LOWER)	Transfer size setup register – lower byte
0x5A	SPI_TRANSFER_SIZE_U (UPPER)	Transfer size setup register – upper byte
0x5B	SPI_MASTER_TRANSFER_PENDING	Transfer pending register

Table 2.22 SPI Master Register Addresses

The SPI Master module uses four wire interfaces: MOSI, MISO, CLK and SS#. There are four SS# lines to control four different SPI Slave devices. The connection diagram is shown in Figure 2.1.

The main purpose is to send data from main memory to the attached SPI slave, and or to receive data and send it to main memory. The SPI Master is controlled by the internal CPU using memory mapped I/O registers. It operates from the main system clock, though sampling of input data and transmission of output data is controlled by the SPI clock (CLK).

An SPI transfer can only be initiated by the SPI Master and begins with the slave-select signal (SS#) being asserted by setting the `spi_ss_n` bit in SPI_MASTER_SETUP register. This is followed by a data byte being clocked out with the master supplying CLK. Once the master has transferred the desired number of bytes, it terminates the transaction by de-asserting slave-select. The SPI Master can abort a transfer at any time by clearing the `spi_ss_n` bit to de-assert slave-select.

The CPU may control data transfer using the interrupts/status register `SPI_MASTER_INT`. Data received by the SPI Master can be read from the `SPI_MASTER_RX_DATA` register, and data to be sent out is written to the `SPI_MASTER_TX_DATA` register. In the case of data being sent from the Master, there are bits indicating when there is space to write into the Tx holding register. Also, there is a Tx-overrun bit (which is set when the user attempts to write data to a full Tx register), a bit indicating whether the state machine is busy processing a transfer, and a Tx-done interrupt when a byte has been sent. In the case of data received by the Master, the RX-full interrupt indicates new data is available, and the Rx-overrun bit indicates that data has been received when the Rx register was full.

The SPI Master module also supports transfers of predefined data packets. This performs automatic control over SS#. The size of the transfer is specified in `SPI_TRANSFER_SIZE_U` and `SPI_TRANSFER_SIZE_L` registers, and a completed transfer is indicated by the

`transfer_size_done_int` interrupt in SPI_MASTER_INT. The `SPI_MASTER_DATA_DELAY` register must be non-zero to use this method.

Data transfers can also be controlled by DMA: more details can be found in the DMA section of this document.

2.3.1 SPI_MASTER_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	`spi_master_dev_en`	R/W	0	Enable SPI Master
0	`spi_master_soft_reset`	R/W	0	Reset SPI Master

Table 2.23 SPI Master Control Register

The SPI Master Control register provides top-level enable and reset functions for the SPI Master module.

The SPI Master module is enabled by setting the `spi_master_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `spi_master_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.3.2 SPI_MASTER_TX_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	Data	R/W	0	Byte of data to clock out on SPI Master bus.

Table 2.24 SPI Master Transmit Register

This register contains data to transmit from the Master to the Slave. Writing to this register will start an SPI Master Write if a Slave Select (SS#) line is asserted.

2.3.3 SPI_MASTER_RX_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	Data	R	0	Byte of data to clocked in on SPI Master bus.

Table 2.25 SPI Master Receive Register

Data transmitted from the Slave to the Master is stored in this register. This will contain the data clocked from the Slave during the previous Master Write.

2.3.4 SPI_MASTER_IEN

Bit Position	Bit Field Name	Type	Reset	Description
7..6	RFU	R	0	Reserved
5	`transfer_size_done_ien`	R/W	0	When set will enable `transfer_size_done_int`
4	`rx_oe_ien`	R/W	0	When set will enable `rx_oe_int`
3	`rx_full_ien`	R/W	0	When set will enable `rx_full_int`
2	`tx_oe_ien`	R/W	0	When set will enable `tx_oe_int`
1	`tx_done_ien`	R/W	0	When set will enable `tx_done_int`
0	`hold_txe_ien`	R/W	0	When set will enable `hold_txe_int`

Table 2.26 SPI Master Interrupt Enable Register

This register will enable or disable interrupts from the SPI Master module. When enabled, an interrupt in the `SPI_MASTER_INT` register will lead to a top-level peripheral interrupt in the spi_master_irq bit in the `PERIPHERAL_INT1` register.

2.3.5 SPI_MASTER_INT

Bit Position	Bit Field Name	Type	Reset	Description
7..6	RFU	R	0	Reserved
5	`transfer_size_done_int`	R/W1C	0	Indicates when a transmission of the `SPI_MASTER_TRANSFER_SIZE` bytes has completed.
4	`rx_oe_int`	R/W1C	0	Indicates an Rx overrun error when data is received and `SPI_MASTER_RX_DATA` is still full. If this occurs the new data is discarded.
3	`rx_full_int`	R/W1C	0	Indicates a Rx data register interrupt that `SPI_MASTER_RX_DATA` has new data to be read out.
2	`tx_oe_int`	R/W1C	0	Indicates a Tx overrun error when data is written to `SPI_MASTER_TX_DATA` while the register is still full. If this occurs the old data is overwritten.
1	`tx_done_int`	R/W1C	0	Indicates when a transmission has completed. Set when the data in `SPI_MASTER_TX_DATA` has been sent.
0	`hold_txe_int`	R/W1C	0	Indicates a Tx holding register interrupt. Set when the holding register is empty.

Table 2.27 SPI Master Interrupt Status Register

The status of each SPI Master module interrupt is read from this register. When an interrupt is enabled and the interrupt is active then a top level peripheral interrupt in the `spi_master_irq` bit in the `PERIPHERAL_INT1` register is set.

Clearing an interrupt bit is achieved by writing a 1 to the corresponding bit field. Writing a zero has no effect.

2.3.6 SPI_MASTER_SETUP

Bit Position	Bit Field Name	Type	Reset	Description
7..4	RFU	R	0	Reserved
3	spi_ss_n	R/W	1	SPI Slave Select. Used when the CPU wishes to control a SS# signal. When LOW is sets SS# active, when HIGH it set it inactive.
2	lsbfirst	R/W	0	When HIGH, data is transferred LSB first. When LOW, data is transferred MSB first.
1	cpol	R/W	0	SPI Clock Polarity (CPOL) Bit - selects the polarity of the SPI clk.
0	cpha	R/W	0	SPI Clock Phase (CPHA) Bit - selects the phase of the SPI clk.

Table 2.28 SPI Master Setup Register

When transmitting data between SPI modules, both modules must be using the same CPOL and CPHA values. A change to either of these bits aborts a transmission in progress and returns the SPI system into an idle state.

Combined, the CPOL and CPHA settings make 4 modes that are listed in Table 8.

Mode 0 and 1: CPOL = 0, the base (inactive) level of SCLK is 0.

When CPHA = 0, data is clocked in on the rising edge of SCLK, and data is clocked out on the falling edge of SCLK.

When CPHA = 1, data is clocked in on the falling edge of SCLK, and data is clocked out on the rising edge of SCLK

Mode 2 and 3: CPOL =1, the base (inactive) level of SCLK is 1.

When CPHA = 0, data is clocked in on the falling edge of SCLK, and data is clocked out on the rising edge of SCLK

When CPHA =1, data is clocked in on the rising edge of SCLK, and data is clocked out on the falling edge of SCLK.

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Table 2.29 SPI Master Mode Numbers

The SPI Slave Select signal is enabled or disabled with the `spi_ss_n` bit. The `spi_ss_n` bit should NOT be used in conjunction with SPI Transfer Size register.

Note: When `spi_ss_n` is de-asserted the `cpol` bit should NOT be toggled at the same time.

Note: The `lsbfirst` bit does not affect the order in which data is stored in the rx and tx registers, simply the order in which data is transmitted and received.

2.3.7 SPI_MASTER_CLK_DIV

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`clk_div`	R/W	0	Clock divider register to determine the frequency of the SPI clk signal.

Table 2.30 SPI Master Clock Divisor Register

The SPI Master clock can operate up to one half of the CPU system clock:

CPU running at 48Mhz would set the SPI maximum clock to 24Mhz

CPU running at 24Mhz would set the SPI maximum clock to 12Mhz

CPU running at 12Mhz would set the SPI maximum clock to 6hMz

The SPI Master clock frequency can be calculated:

Fsclk = (Fclk / 2) / div

Fsclk - SPI Master clock frequency.
Fclk – CPU system clock frequency.
div – Clock divider.

If the CPU runs at 48MHz a divider value of both 0 and 1 will result in an SPI clock frequency of 24MHz.

2.3.8 SPI_MASTER_DATA_DELAY

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`data_delay`	R/W	0	Inserts a fixed delay between SS# going active and the first SCLK cycle. The value of this register is the number of SCLK periods to delay. For example, if SCLK is 100 kHz, a value of 255 gives a delay of 2.55 ms.

Table 2.31 SPI Master Data Delay Register

It is recommended that this value is non-zero when using the Transfer Size feature for automatic control over SS#.

2.3.9 SPI_MASTER_SS_SETUP

Bit Position	Bit Field Name	Type	Reset	Description
7..3	RFU	R	0	Reserved
2..1	`ss_route`	R/W	0	SPI Slave Select Route. Two bits set the active slave select out of the 4 different slave selects.
0	`ss_idle_state`	R/W	1	SPI Slave Select Idle State.

Table 2.32 SPI Master Slave Select Setup

When setting the value of `ss_idle_state`: '1' sets an idle state of high, therefore SS# is active low; '0' is an idle state of low, therefore SS# is active high.

2.3.10 SPI_MASTER_TRANSFER_SIZE

The SPI_MASTER_TRANSFER_SIZE register contains 16 bits and is split over 2 registers; `SPI_MASTER_TRANSFER_SIZE_L` at 0x69, and `SPI_MASTER_TRANSFER_SIZE_U` at 0x6A.

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`xfer_size_l`	R/W	0	Lower 8 bits of transfer size register.

Table 2.33 SPI Master Transfer Size (Lower) Register

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`xfer_size_u`	R/W	0	Upper 8 bits of transfer size register.

Table 2.34 SPI Master Transfer Size (Upper) Register

This allows the hardware to auto-control the assertion and de-assertion of slave select. Set the lower byte first and then the upper byte. The `SPI_MASTER_DATA_DELAY` must be programmed with a non-zero value when using these registers for automatic control over SS#.

Setting both bytes to 0 will abort a transfer in process.

If the transfer size is non-zero then the `spi_ss_n` bit in Section 0 should be set to '1' (inactive).

2.3.11 SPI_MASTER_TRANSFER_PENDING

Bit Position	Bit Field Name	Type	Reset	Description
7..1	RFU	R	0	Reserved
0	`transfer_pending`	R	0	The live status of the SPI Master. Set to '1' when the SPI Master is busy servicing a prior request.

Table 2.35 SPI Master Transfer Pending Register

The `transfer_pending` bit reports the status of the SPI Master in real time.

2.3.12 Use Cases

The SPI Master can be used in the following ways:

As a polled interface – writing a single byte at a time to the transmit register and reading single byte responses from the receive register.

As an interrupt driven interface – an interrupt handler sends multiple bytes of data to the transmit register and receives a corresponding amount of data from the receive register.

A DMA driven interface – two DMA engines are configured to send data to, and receive data from, the SPI slave. Program code is not required to perform any actions during a transfer.

2.3.12.1 Interface Setup

To setup the SPI Master, go through the following steps:

1. Reset the SPI Master in the `SPI_MASTER_CONTROL` register.
2. Enable the SPI Master in the `SPI_MASTER_CONTROL` register.
3. Set the required frequency of SCLK via the clock divisor in the `SPI_MASTER_CLK_DIV` register.
4. Setup the SPI Master Mode and bit order as required in the `SPI_MASTER_SETUP` register.
5. Leave the Slave Select line inactive.
6. Set the desired Slave Select line and idle state in the `SPI_MASTER_SS_SETUP` register.
7. The interface is now ready to use.

Enabling and Disabling Slave Select:

Manual control over SS# is performed by enabling or disabling SS# under program control via the `spi_ss_n` bit of the `SPI_MASTER_SETUP` register. The `SPI_MASTER_TRANSFER_SIZE_L` and `SPI_MASTER_TRANSFER_SIZE_U` registers must be zero for this method.

Alternatively, automatic control (where the amount of data to transfer is known in advance) is done by leaving the Slave Select line inactive and programming the number of bytes to transfer into the `SPI_MASTER_TRANSFER_SIZE` registers. SS# is enabled when data is next written to the `SPI_MASTER_TX_DATA` register and disabled automatically. To correctly enable SS# at the start of a transfer program a non-zero value must be programmed into the `SPI_MASTER_DATA_DELAY` register.

Here is an example SPI Master setup that uses WRITE_IO_REG macro defined in 5.1.2.

```
WRITE_IO_REG(0x0050, 0x01); // Reset to a known state
WRITE_IO_REG(0x0050, 0x02); // Enable SPI Master device before any setup not after.
WRITE_IO_REG(0x0056, 0x60); // Divide the FT51A system clock by 0x60
WRITE_IO_REG(0x0058, 1 << 0x00    // Set SS Idle State to High,
                   | 0 << 0x01); // Set SS number to 0
WRITE_IO_REG(0x0055, 0 << 0x00    // Set SCLK phase to 0
                   | 0 << 0x01    // Set SCLK polarity to 0
                   | 0 << 0x02    // Set data order to MSB
```

```
                    | 1 << 0x03); // Set SS high, i.e. deactivate it
```

2.3.12.2 Polled Interface

The SPI Master device is used as a polled interface sending a command to an SPI slave and reading a response.

1. Enable the Slave Select method required.
2. For each byte of data:
    a. Write a byte to send to the slave into the `SPI_MASTER_TX_DATA` register.
    b. Poll the `SPI_MASTER_TRANSFER_PENDING` register until the pending bit clears.
    c. Read a byte from the slave from the `SPI_MASTER_RX_DATA` register.
3. Disable Slave Select.

The polled interface is useful for low-speed transactions where there are small amounts of data to receive from, or send to, the SPI slave. It is particularly useful when there are decisions to be taken depending on the contents of data returned from the SPI slave.

2.3.12.3 Interrupt Interface

An interrupt handler can be employed to speed up transmission and reception of multiple bytes of data. This method requires the code to send the first byte of data to the SPI slave and an interrupt handler routine to send subsequent bytes of data. The interrupt handler will also receive the data returned from the SPI slave.

To setup an interrupt routine:

- Clear the `tx_done_int` bit in the `SPI_MASTER_INT` register.
- Enable the interrupt in the `tx_done_ien` in the `SPI_MASTER_IEN` register.
- Initialise the buffer to transmit and buffer to receive data for use by the interrupt handler routine.
    - Enable the Slave Select method required.
    - Write the first byte of data to the `SPI_MASTER_TX_DATA` register.
    - Wait for the interrupt handler to signal that the transfer is complete.
    - Disable Slave Select.
    - Within the interrupt handler:
        - Check for `tx_done_int` bit in the `SPI_MASTER_INT` register to be set and `tx_done_ien` in the `SPI_MASTER_IEN` register to be set.
        - Read a byte of data from the slave from the `SPI_MASTER_RX_DATA` register and place it in the receive buffer.
- Clear the `tx_done_int` bit in the `SPI_MASTER_INT` register.

If there is data still to transfer, write the next byte of the transmitting buffer to the `SPI_MASTER_TX_DATA` register. If the transfer is complete, signal, transfer complete.

This method is useful for most transfers of data to and from an SPI slave. It is still under program control so decisions on data flow can be made within the interrupt handler. It is, however, recommended to have a minimum of code in any interrupt handler. An example Interrupt Service Routine is presented below.

```
//ISR

void SPIM_interrupt_handler()
{
    READ_IO_REG(0x54, SPIM_interrupt); //SPI_MASTER_STAT_1

    if(SPIM_interrupt & (0x1<<3)) //Check for RX_FULL interrupt
    {
        SPIM_num_bytes_tx--;

        READ_IO_REG(0x52, *SPIM_MISO_buf); //SPI_MASTER_DATA_RX_ADDR_1
```

```
        SPIM_MISO_buf++;

        WRITE_IO_REG(0x54, 0x1<<3); //Clear RX_FULL interrupt
    }

    if(SPIM_interrupt & 0x1) // Check for TX_HOLD interrupt

    {

        if(SPIM_num_bytes_tx > 1)

        {

            WRITE_IO_REG(0x51, *SPIM_MOSI_buf); //SPI_MASTER_DATA_TX_ADDR_1
            SPIM_MOSI_buf++;
        }

        WRITE_IO_REG(0x54, 0x1); //Clear TX_HOLD interrupt
    }

}

//It is up to the user to define the variables,
//initialise transfer and check if all bytes have been sent as follows:


//Variables

volatile uint8_t              SPIM_interrupt;

volatile uint16_t             SPIM_num_bytes_tx;

volatile uint8_t             *SPIM_MISO_buf;

volatile uint8_t             *SPIM_MOSI_buf;


//Initialise transfer

WRITE_IO_REG(0x51, *SPIM_MOSI_buf); //SPI_MASTER_DATA_TX_ADDR_1
SPIM_MOSI_buf++;

//Wait until all bytes have been sent by ISR

while(SPIM_num_bytes_tx != 0);
```

2.3.12.4 DMA Interface

Data transfers can also be controlled by DMA. Below is the sequence of steps required to control transfer via DMA. Additional details on how to configure DMAs can be found in the DMA section of this document.

1. Initialise DMA;
2. Acquire DMA PUSH (Tx) and PULL (Rx) engines;
3. Configure DMA source, destination, data size, mode and function;
4. Activate Slave Select;
5. Enable the Rx DMA (nothing will happen yet);
6. Enable the Tx DMA (this starts both transfers);
7. Wait for the Tx DMA transfer to complete;
8. Wait for the Rx DMA transfer to complete (this will happen almost immediately);
9. Deactivate Slave Select;

At the end of the transfer, the master and slave will have exchanged data.

2.4 SPI Slave

The Serial Peripheral Interface Bus is an industry standard communications interface. Devices communicate in Master or Slave modes, with the Master initiating the data transfer.

The SPI Slave module has four signals:

- Clock
- Slave Select
- MOSI (master out – slave in)
- MISO (master in – slave out).

The SPI Slave protocol by default does not support any form of handshaking and the only available mode is unmanaged. Data is clocked out of the Master and clocked in from the Slave simultaneously.

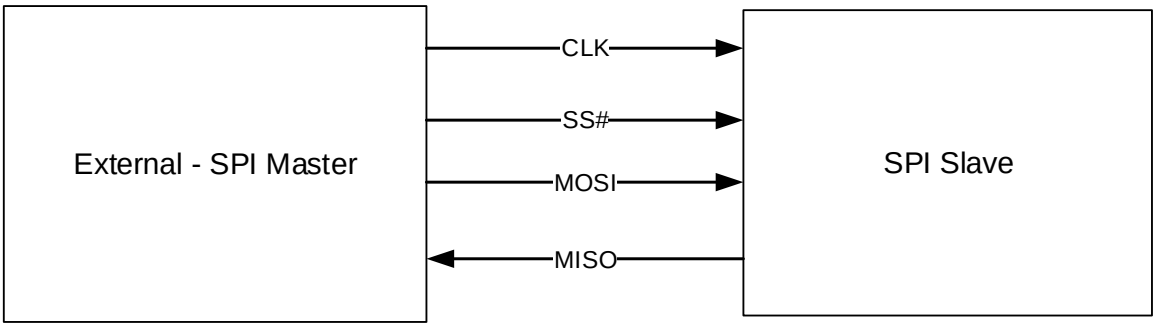


Figure 2.2 SPI Slave Schematic Diagram

The registers associated with the SPI Slave are outlined in Table 2.36.

I/O Address	Register Name	Description
0x48	SPI_SLAVE_CONTROL	SPI Slave Control Register
0x4A	SPI_SLAVE_DATA_TX	Transmit data register
0x4B	SPI_SLAVE_DATA_RX	Receive data register
0x4C	SPI_SLAVE_IEN	Interrupt Enable register
0x4D	SPI_SLAVE_INT	Interrupt Status register
0x4E	SPI_SLAVE_SETUP	Setup register

Table 2.36 SPI Slave Register Addresses

The SPI Slave module uses a four wire interface: MOSI, MISO, CLK and SS# as shown in Figure 2.2

The main purpose is to send data from main memory to the attached SPI master, and or receive data and send it to main memory. The SPI Slave is controlled by the internal CPU using internal memory-mapped I/O registers. It operates from the main system clock, although sampling of input data and transmission of output data is controlled by the SPI clock (CLK). An SPI transfer can only be initiated by the SPI Master and begins with the slave select signal being asserted. This

is followed by a data byte being clocked out with the master driving CLK. The master always supplies the first byte, which is called a command byte. After this the desired number of data bytes are transferred before the transaction is terminated by the master de-asserting slave select. An SPI Master is able to abort a transfer at any time by de-asserting its SS# output. This will cause the Slave to end its current transfer and return to an idle state.

Data transfer can be controlled by the CPU using interrupts/status register `SPI_SLAVE_INT`. Data sent to the SPI Slave block can be read out from the `SPI_SLAVE_RX` register and data to be sent out has to be written to the `SPI_SLAVE_TX` register. In case of data being sent from the block there are bits indicating when there is space to write into the Tx holding register. Also, there is a Tx overrun bit which is set when data is attempted to be written to a full Tx register, a bit indicating whether the state machine is busy processing a transfer and a Tx done interrupt when a byte has been sent. In the case of data to be received by the block there is a RX full interrupt indicating new data to be read out by the CPU and Rx overrun bit that indicates that data has been received when RX register was full.

Data transfers can also be controlled by DMA. More details on how to configure DMAs to work with the SPI Slave block can be found in the DMA section of this document.

2.4.1 SPI_SLAVE_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	`spi_slave_dev_en`	R/W	0	Enable SPI Slave
0	`spi_slave_soft_reset`	R/W	0	Reset SPI Slave

Table 2.37 SPI Slave Control Register

The SPI Slave Control register provides top-level enables and reset functions for the SPI Slave module.

The SPI Slave module is enabled by setting the `spi_slave_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `spi_slave_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.4.2 SPI_SLAVE_TX_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`Data`	R/W	0	Byte of data to be transmitted from the SPI Slave module

Table 2.38 SPI Slave Transmit Register

This register contains data to be transmitted from the Slave to the Master.

2.4.3 SPI_SLAVE_RX_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`Data`	R	0	Byte of data received by the SPI Slave module

Table 2.39 SPI Slave Receive Register

This register contains data that was sent from the SPI Master.

2.4.4 SPI_SLAVE_IEN

Bit Position	Bit Field Name	Type	Reset	Description
7..5	RFU	R	0	Reserved
4	`rx_oe_ien`	R/W	0	When set will enable `rx_oe_int`
3	`rx_full_ien`	R/W	0	When set will enable `rx_full_int`
2	`tx_oe_ien`	R/W	0	When set will enable `tx_oe_int`
1	`tx_done_ien`	R/W	0	When set will enable `tx_done_int`
0	`hold_txe_ien`	R/W	0	When set will enable `hold_txe_int`

Table 2.40 SPI Slave Interrupt Enable Register

This register will enable or disable interrupts from the SPI Slave module. When enabled, an interrupt in the `SPI_SLAVE_INT` register will lead to a top-level peripheral interrupt in the `spi_slave_irq` bit in the `PERIPHERAL_INT1` register.

2.4.5 SPI_SLAVE_INT

Bit Position	Bit Field Name	Type	Reset	Description
7..6	RFU	R	0	Reserved
5	`tx_busy`	R	0	Indicates the module is busy processing a transfer
4	`rx_oe_int`	R/W1C	0	Indicates a RX overrun error when data is received and the SPI_SLAVE_RX_DATA register is still full. If this occurs the new data is discarded.
3	`rx_full_int`	R/W1C	0	Indicates a Rx data register interrupt that SPI_SLAVE_RX_DATA has new data to be read out.
2	`tx_oe_int`	R/W1C	0	Indicates a Tx overrun error when data is written to the `SPI_SLAVE_TX_DATA` register while the register is still full. If this occurs the old data is overwritten.
1	`tx_done_int`	R/W1C	0	Indicates when a transmission has completed. Set when the data in `SPI_SLAVE_TX_DATA` has been sent.
0	`hold_tx_int`	R/W1C	0	Indicates a Tx holding register interrupt. Set when the holding register is empty.

Table 2.41 SPI Slave Interrupt Status Register

The status of each SPI Slave module interrupt is read from this register. When an interrupt is enabled and the interrupt is active then a top level peripheral interrupt in the `spi_slave_irq` bit in the `PERIPHERAL_INT1` register is set.

Clearing an interrupt bit is achieved by writing a 1 to the corresponding bit field. Writing a zero has no effect.

2.4.6 SPI_SLAVE_SETUP

Bit Position	Bit Field Name	Type	Reset	Description
7..3	RFU	R	0	Reserved
2	`lsbfirst`	R/W	0	When HIGH, data is transferred LSB first. When LOW, data is transferred MSB first.
1	`Cpol`	R/W	0	SPI Clock Polarity (CPOL) Bit - selects the polarity of the SPI clk.
0	`Cpha`	R/W	0	SPI Clock Phase (CPHA) Bit - selects the phase of the SPI clk.

Table 2.42 SPI Slave Setup Register

When transmitting data between SPI modules, both modules must be using the same CPOL and CPHA values. A change to either of these bits aborts a transmission in progress and returns the SPI system into an idle state.

Combined, the CPOL and CPHA settings make 4 modes that are listed in Table 5.

Mode 0 and 1: CPOL = 0, the base (inactive) level of SCLK is 0.

- When CPHA = 0, data is clocked in, on the rising edge of SCLK, and data is clocked out, on the falling edge of SCLK.
- When CPHA = 1, data is clocked in, on the falling edge of SCLK, and data is clocked out, on the rising edge of SCLK

Mode 2 and 3: CPOL = 1, the base (inactive) level of SCLK is 1.

- When CPHA = 0, data is clocked in, on the falling edge of SCLK, and data is clocked out, on the rising edge of SCLK
- When CPHA = 1, data is clocked in, on the rising edge of SCLK, and data is clocked out, on the falling edge of SCLK.

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Table 2.43 SPI Slave Mode Numbers

Note: The `lsbfirst` bit does not affect the order, in which data is stored in the rx and tx registers, simply the order in which data is transmitted and received.

2.5 I²C Master

The I²C is an industry standard communications interface. Devices communicate in Master or Slave mode, with the Master initiating the data transfer.

The I²C Master module has two signals:

- Clock (SCL)
- Data (SDA)

The I²C Master transmits any data by prefixing it with an I²C Slave address. The Least Significant Bit of the address specifies a Read or Write operation.

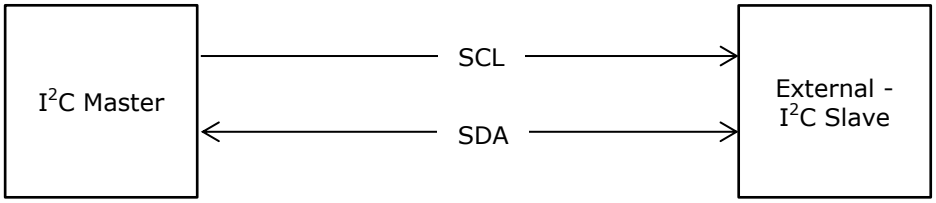


Figure 2.3 I²C Master Schematic Diagram

The registers associated with the I²C Master are outlined in Table 2.44. These are accessed using SFRs directly.

SFR Address	Register Name	Description
0xF4	I2CMSA	Slave Address register.
0xF5	I2CMCR	Control register (write operation).
0xF5	I2CMSR	Status register (read operation).
0xF6	I2CMBUF	Transmitted/received data register.
0xF7	I2CMTP	Timer period register.

Table 2.44 I²C Master Register Addresses

2.5.1 I2CMSA

Bit Position	Bit Field Name	Type	Reset	Description
7..1	Addr	R/W	0	Slave address MSB bits 7..1
0	R/S	R/W	0	Receive/Send

Table 2.45 I²C Master Slave Address Register

The Slave Address register sets the address of the I²C Slave. The most significant 7 bits of the address are set in this register. The least significant bit is '1' for a receive transaction or '0' for a send transaction.

2.5.2 I2CMCR

The I^{2}C Master Control register is accessed only during a write. If this register is read then it will return the status value of the I2CMSA register.

Bit Position	Bit Field Name	Type	Reset	Description
7	RSTB	W1T	0	Triggers a reset of the I^{2}C Master module.
6	SLRST	W1T	0	Performs a slave reset.
5	ADDR	W1T	0	Slave Address
4	HS	W1T	0	High-speed mode
3	ACK	W1T	0	Acknowledgment
2	STOP	W1T	0	When set, causes a stop after the first data cycle. When clear, will allow transfers to continue on to a burst.
1	START	W1T	0	When set, causes generation of START or REPEATED START condition.
0	RUN	W1T	0	Run condition

Table 2.46 I^{2}C Master Control Register

To reset a bus blocked by an I2C Slave device set SLRST and RUN. This will generate 9 SCKs and a STOP.

2.5.3 I2CMSR

The I^{2}C Master Status register is accessed only during a read operation.

Bit Position	Bit Field Name	Type	Reset	Description
7	Reserved	RFU	0	Reserved
6	BUS_BUSY	R	0	Indicates that the Bus is Busy, and access is not possible; set/reset by START and STOP conditions
5	IDLE	R	0	Indicates that the I^{2}C Bus controller is in the idle state
4	ARB_LOST	R	0	Indicates that due to the last operation, the I^{2}C Bus controller lost the arbitration
3	DATA_ACK	R	0	Indicates that due to the last operation the transmitted data was not acknowledged
2	ADDR_ACK	R	0	Indicates that due to the last operation the slave address was not acknowledged
1	ERROR	R	0	Indicates that due to the last operation an error occurred: slave address was not acknowledged, transmitted data was not acknowledged, or the I^{2}C Bus controller lost the arbitration
0	BUSY	R	0	Indicates that the I^{2}C Bus controller receiving, or transmitting data on the bus, and other bits of the Status register are not valid

Table 2.47 I^{2}C Master Status Register

2.5.4 I2CMBUF

Bit Position	Bit Field Name	Type	Reset	Description
7..0	data	R/W	0	Data register.

Table 2.48 I^{2}C Master Data Buffer Register

The I2CMBUF register when read contains the data received during the last read operation. When written the data in the register will be transmitted on the next send operation.

2.5.5 I2CMTP

Bit Position	Bit Field Name	Type	Reset	Description
7	HS_TIMER_SELECT	R/W	0	Set to select the timer period register for HIGH speed I²C Master. Clear for timer period register for STANDARD, FAST and FAST+.
6..0	timer	R/W	0x01	Timer Period register.

Table 2.49 I²C Master Timer Period Register

$$F_{scl} = F_{clk} / (2 * (1 + timer) * 10)$$

F_{scl} – I2C Master clock frequency.
F_{clk} – CPU system clock frequency.
timer – Timer period divider.

The I²C Master will automatically adopt the relevant I²C mode (STANDARD, FAST, FAST-PLUS, HIGH-SPEED) depending on the SCL frequency calculated. The maximum frequency is limited to the lesser of one tenth of the system clock frequency or 3,400,000Hz. This will support the standard I²C modes:

- 100 kbit/s standard
- 400 kbit/s Fast
- 1 Mbit/s Fast+
- 3.4 Mbit/s High Speed

If the system clock frequency is changed then the value in this register will need to be recalculated to ensure correct operation.

2.5.6 Use Case

The I²C Master can process single bytes or bursts of an indeterminate length from the I²C Slave.

2.5.6.1 Interface Setup

To setup the I²C Master the following steps have to be performed:

- The I²C Master must first be reset by writing a '1' to the `RSTB` bit in the `I2CMCR register`.
- Set the frequency via the `I2CMTP` register.

```
// Reset I2C Master block

I2CMCR |= 0x80;


__asm NOP __endasm;

// Set freqency
I2CMTP = 0x20;
__asm NOP __endasm;
```

2.5.6.2 Send Data

To send data to an I²C Slave the following procedure is used:

- For a single cycle:
  - Write the slave address to the `I2CMSA` register and set bit R/S to 0.
  - Write the first byte of data to the `I2CMBUF` register.
  - Write to the Control Register `I2CMCR` with `HS=0`, `STOP=1`, `START=1`, `RUN=1`.
  - Read the `I2CMSR` register until the `BUSY` bit is clear.
- For multiple cycles:
  - Write the slave address to the `I2CMSA` register and set bit R/S to 0.
  - Write the first byte of data to the `I2CMBUF` register.
  - Write to the Control Register `I2CMCR` with `HS=0`, `STOP=0`, `START=1`, `RUN=1`.
  - Read the `I2CMSR` register until the `BUSY` bit is clear.
  - For remaining bytes (except last byte):
    - Write the next byte of data to the `I2CMBUF` register.
    - Write to the Control Register `HS=0`, `STOP=0`, `START=1`, `RUN=1`.
    - Read the `I2CMSR` register until the `BUSY` bit is clear.
  - Write the last byte of data to the `I2CMBUF` register.
  - Write to the Control Register `I2CMCR` with `HS=0`, `STOP=1`, `START=1`, `RUN=1`.
  - Read the `I2CMSR` register until the `BUSY` bit is clear.

To allow a burst write, change `STOP` bit to 0.

```
uint8_t       data;

// Set I2C Slave Address
I2CMSA = 0x22<<1;
__asm NOP __endasm;

I2CMBUF = data;
I2CMCR  = 0x04 | 0x02 | 0x01; // I2C_FLAGS_STOP | I2C_FLAGS_START | I2C_FLAGS_RUN

__asm NOP __endasm;
__asm NOP __endasm;
__asm NOP __endasm;

do
{ // loop while busy
    status = I2CMCR;

    if (!(status & 0x01)) //I2C_STATUS_BUSY
    {
      if (status & 0x02) //I2C_STATUS_ERROR
        return 0;
      return 1;
    }
}while( 1 );
```

2.5.6.3 Receive Data

To receive data from an I²C Slave the following procedure is used:

- For single cycle:
  - Write the slave address to the `I2CMSA` register and set bit R/S to 1.
  - Write to the Control Register `I2CMCR` with `HS=0`, `ACK=1`, `STOP=1`, `START=1`, `RUN=1`.
  - Read the `I2CMSA` register until the `BUSY` bit is clear.
  - Read the first byte of data from the `I2CMBUF` register.
- For multiple cycles:
  - Write the slave address to the `I2CMSA` register and set bit R/S to 1.
  - Write to the Control Register `I2CMCR` with `HS=0`, `ACK=1`, `STOP=0`, `START=1`, `RUN=1`.

- Read the I2CMSA register until the BUSY bit is clear.
- Read the first byte of data from the I2CMBUF register.
- For the remaining bytes (except the last byte):
  - Write to the Control Register I2CMCR with HS=0, ACK=1, STOP=0, START=1, RUN=1.
  - Read the I2CMSA register until the BUSY bit is clear.
  - Read the next byte of data from the I2CMBUF register.
- Write to the Control Register I2CMCR with HS=0, ACK=0, STOP=1, START=1, RUN=1.
- Read the I2CMSA register until the BUSY bit is clear.
- Read the last byte of data from the I2CMBUF register.

```
uint8_t       data;
uint8_t       nFlagData;

// Set I2C Slave Address
I2CMSA = 0x22<<1 | 0x01; \\I2C_READ_NOT_WRITE
__asm NOP __endasm;

do
{ // loop while busy
    status = I2CMCR;

    if (!(status & 0x01)) //I2C_STATUS_BUSY
    {
      if (status & 0x02) //I2C_STATUS_ERROR
        return 0;
      return 1;
    }
}while( 1 );

I2CMCR  = 0x04 | 0x02 | 0x01; // I2C_FLAGS_STOP | I2C_FLAGS_START | I2C_FLAGS_RUN

do
{ // loop while busy
    status = I2CMCR;

    if (!(status & 0x01)) //I2C_STATUS_BUSY
    {
      if (status & 0x02) //I2C_STATUS_ERROR
        return 0;

      return 1;

    }

}while( 1 );


data = I2CMBUF;


nFlagData = I2CMCR;
I2CMCR = nFlagData & ~0x08; // clear ack'ing
```

2.6 I²C Slave

The I²C Slave is an industry standard communications interface. Devices communicate in Master or Slave mode, with the Master initiating the data transfer.

The I²C Slave module has two signals:

- Clock (SCL)
- Data (SDA)

The I²C Slave responds to an address transmitted by the I²C Master that prefixes any data transferred. The Least Significant Bit of the address specifies Read or Write operation.

The I²C Slave is a polled interface. It will return data and acknowledge a read to the master only when data has been written to the I²C Slave data register. Likewise, the I²C Slave will not acknowledge a write by the master until data has been read from the I²C Slave data register.

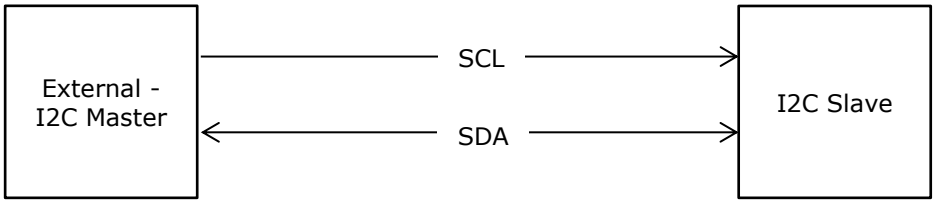


Figure 2.4 I²C Slave Schematic Diagram

The registers associated with the I²C Slave are outlined in Table 2.50. These are accessed using SFRs directly.

SFR Address	Register Name	Description
0xF1	I2CSOA	Own Address register.
0xF2	I2CSCR	Control register (write operation).
0xF2	I2CSSR	Status register (read operation).
0xF3	I2CSBUF	Transmitted/received data register.

Table 2.50 I²C Slave Register Addresses

2.6.1 I2CSOA

Bit Position	Bit Field Name	Type	Reset	Description
7	Reserved	RFU	0	Reserved
6..0	addr	R/W	0	Own address bits 7..1. Note that own address bit zero indicates direction.

Table 2.51 I²C Slave Address Register

The Own Address register sets the address that the I²C Slave will respond to. The most significant 7 bits of the address are set in this register. The least significant bit is '1' for a read by the I²C Master or '0' for a write.

2.6.2 I2CSCR

The I²C Slave Control register is accessed only during a write operation. If this register is read then it will return the value of the I2CSSR register.

Bit Position	Bit Field Name	Type	Reset	Description
7	RSTB	W	0	Reset of I²C Slave function.
6	DA	W	0	Activate I²C Slave device.
5..4	Reserved	RFU	0	Reserved
3	RECFINCLR	W	0	Clear RECFIN bit in I2CSSR register.
2	SENDFINCLR	W	0	Clear SENDFIN bit in I2CSSR register.
1..0	Reserved	RFU	0	Reserved

Table 2.52 I²C Slave Control Register

The I²C Slave Control register provides top-level enables and reset functions for the I²C Slave module. It allows the status of RECFIN and SENDFIN to be cleared in the I2CSSR (I²C Slave Status) register.

2.6.3 I2CSSR

The I²C Slave Status register is accessed only during a read operation.

Bit Position	Bit Field Name	Type	Reset	Description
7	Reserved	RFU	0	Reserved
6	DA	R	0	I²C Slave activated.
5	Reserved	RFU	0	Reserved
4	BUSACTIVE	R	0	Send, receive, or address detection in progress. This bit is cleared automatically at the end of a transfer.
3	RECFIN	R	0	The I²C Master has completed a transmit operation. Cleared by writing '1' to RECFINCLR in the I2CSCR register.
2	SENDFIN	R	0	The I²C Master has completed a receive operation. Cleared by writing '1' to SENDFINCLR in the I2CSCR register.

1	TREQ	R	0	The I²C Slave has been addressed for a read operation by the I²C Master and must send a byte of data. This bit is automatically cleared by a write to I2CSBUF.
0	RREQ	R	0	The I²C Slave has been addressed for a write operation by the I²C Master and must receive a byte of data. This bit is automatically cleared by a read from I2CSBUF.

Table 2.53 I²C Slave Status Register

When the `SENDFIN` and `RECFIN` bits are set then the I²C Slave asserts an interrupt to the EIE SFR.

2.6.4 I2CSBUF

Bit Position	Bit Field Name	Type	Reset	Description
7..0	data	R/W	0	Data register.

Table 2.54 I²C Slave Data Buffer Register

Writing to the I²C Slave Data register when `TREQ` is set in the I2CSCR register will send the byte of data to the I²C Master. Conversely, reading the I²C Slave Data register when `RREQ` is set will acknowledge a transmission from the I²C Master.

2.6.5 Use Case

The I²C Slave can process single bytes or bursts of an indeterminate length from the I²C Master.

The interrupts generated by the I²C Slave are for `RECFIN` and `SENDFIN`. These can be used for chaining burst reads and writes efficiently using an interrupt handler. However, to detect when a master has addressed the I²C Slave interface it is necessary to poll the `TREQ` and `RREQ` bits in the I2CSSR register.

2.6.5.1 Interface Setup

To setup the I²C Slave the Own Address register must be written before the interface is activated.

- Write the slave address to the `I2CSOA register`.
- Enable the I²C Slave function by setting `DA` and clearing `RSTB` in the `I2CSCR` register. The value 0x7F can be used.

2.6.5.2 Send Data

To send data to an I²C Master the following procedure is used:

- For each byte of data:
  - If `I2CSCR` register bit `TREQ` set to '1':
    - Write byte of data to `I2CSBUF`.
  - If `I2CSSR` register bit `SENDFIN` set to '1':
    - Clear `SENDFIN` bit setting `SENDFINCLR` bit in write to `I2CSCR`.
    - Finish transaction.

This will allow a burst read to occur on the I²C Master. The master controls how many bytes will be read from the I²C Slave.

2.6.5.3 Receive Data

To receive data from an I^{2}C Master the following procedure is used:

- For each byte of data:
  - If `I2CSCR` register bit `RREQ` set to '1':
    - Read byte of data from `I2CSBUF`.
  - If `I2CSCR` register bit `RECFIN` set to '1':
    - Clear `RECFIN` bit by setting `RECFINCLR` bit in write `I2CSCR`.
    - Finish transaction.

This will allow a burst write to occur on the I^{2}C Master. The master controls how many bytes will be written to the I^{2}C Slave.

2.7 UART

The UART block provides:

- Full RS232 support;
- Baud Rate Generator;
- Optional hardware flow control via RTS / CTS and DTR / DSR;
- Optional DMA support

The registers associated with the UART are outlined in Table 2.55.

I/O Address	Register Name	Description
0x60	UART_CONTROL	UART control register
0x61	UART_DMA_CTRL	UART DMA control register
0x62	UART_RX_DATA	UART Receive Data register
0x63	UART_TX_DATA	UART Transmit Data register
0x64	UART_TX_IEN	UART Tx Status Enable register
0x65	UART_TX_INT	UART Tx Status Register
0x66	UART_RX_IEN	UART Rx Status Enable Register
0x67	UART_RX_INT	UART Rx Status Register
0x68	UART_LINE_CTRL	UART Line Control Register
0x69	UART_BAUD_0	UART Baud Rate Register – lower byte
0x6A	UART_BAUD_1	UART Baud Rate Register – middle byte
0x6B	UART_BAUD_2	UART Baud Rate Register – upper byte
0x6C	UART_FLOW_CTRL	UART Flow Control Register
0x6D	UART_FLOW_STAT	UART Flow Control Status Register

Table 2.55 UART Register Addresses

2.7.1 UART_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	uart_dev_en	R/W	0	Write 1 to Enable UART
0	uart_soft_reset	R/W	0	Write 1 to Reset UART

Table 2.56 UART Control Register

The UART Control register provides top-level enables and reset functions for the UART module.

The UART module is enabled by setting the uart_dev_en bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the uart_soft_reset bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.7.2 UART_DMA_CTRL

Bit Position	Bit Field Name	Type	Reset	Description
7..1	Reserved	RFU	0	Reserved
0	uart_dma_en	R/W	0	Write 1 to enable DMA mode. Data can be transferred using DMA when enabled. See Section 0 for more details on DMA.

Table 2.57 UART DMA Control Register

2.7.3 UART_RX_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	UART_RX_DATA	R/W	8'h00	Data received by the UART

Table 2.58 UART Data Receive Register

2.7.4 UART_TX_DATA

Bit Position	Bit Field Name	Type	Reset	Description
7..0	UART_TX_DATA	R/W	8'h00	Data to be transmitted by the UART. Data is transmitted automatically by writing to this register

Table 2.59 UART Data Transmit Register

2.7.5 UART_TX_IEN

Bit Position	Bit Field Name	Type	Reset	Description
7..6	Reserved	RFU	0	Reserved
5	`dcd_ien`	R/W	0	Interrupt enable bit for `dcd_int`
4	`ri_ien`	R/W	0	Interrupt enable bit for `ri_int`
3	`dsr_ien`	R/W	0	Interrupt enable bit for `dsr_int`
2	`cts_ien`	R/W	0	Interrupt enable bit for `cts_int`
1	`tx_done_ien`	R/W	0	Interrupt enable bit for `tx_done_int`
0	`hold_txe_ien`	R/W	0	Interrupt enable bit for `hold_txe_int`

Table 2.60 UART Transmit Status Interrupt Enable Register

2.7.6 UART_TX_INT

Bit Position	Bit Field Name	Type	Reset	Description
7	Reserved	RFU	0	Reserved
6	`tx_busy`	RO	0	Set when the UART is transmitting data
5	`dcd_int`	R/W1C	0	Set on any change in the state of the Data_Carrier_Detect signal
4	`ri_int`	R/W1C	0	Set on any change in the state of the Ring_Indicator signal
3	`dsr_int`	R/W1C	0	Set on any change in the state of the Data_Set_Ready signal
2	`cts_int`	R/W1C	0	Set on any change in the state of the Clear_To_Send signal
1	`tx_done_int`	R/W1C	0	Set when the UART has completed a transfer
0	`hold_txe_int`	R/W1C	0	Set high when the Tx buffer becomes empty

Table 2.61 UART Transmit Status Interrupt Register

2.7.7 UART_RX_IEN

Bit Position	Bit Field Name	Type	Reset	Description
7..5	Reserved	RFU	0	Reserved
4	`break_rcvd_ien`	R/W	0	Enable bit for `break_rcvd_int` interrupt
3	`stop_error_ien`	R/W	0	Enable bit for `stop_error_int` interrupt
2	`parity_error_ien`	R/W	0	Enable bit for `parity_error_int` interrupt
1	`rx_overflow_ien`	R/W	0	Enable bit for `rx_overflow_int` interrupt
0	`rx_full_ien`	R/W	0	Enable bit for `rx_full_int` interrupt

Table 2.62 UART Receive Status Interrupt Enable Register

2.7.8 UART_RX_INT

Bit Position	Bit Field Name	Type	Reset	Description
7..5	Reserved	RFU	0	Reserved
4	`break_rcvd_int`	R/W	0	The interrupt bit is set when the receive data line is held at '0' for more than the time it takes to send a full word
3	`stop_error_int`	R/W	0	The interrupt bit is set to indicate that the last bit received was not a stop bit
2	`parity_error_int`	R/W	0	The interrupt bit is set to indicate there was a parity error with the data received
1	`rx_overflow_int`	R/W	0	The interrupt indicates that data has been received but the Rx Buffer had not been emptied from the previous transaction
0	`rx_full_int`	R/W	0	The interrupt indicates that the Rx Data Register contains received data

Table 2.63 UART Receive Status Interrupt Register

2.7.9 UART_LINE_CTRL

<table>
<tr><th>Bit Position</th><th>Bit Field Name</th><th>Type</th><th>Reset</th><th>Description</th></tr>
<tr><td>7..6</td><td>Reserved</td><td>RFU</td><td>0</td><td>Reserved</td></tr>
<tr><td>5</td><td>set_break</td><td>R/W</td><td>0</td><td>When set, the txd line goes into a ‘spacing’ state which causes a break in the receiving UART.
Clear this bit to disable the break.</td></tr>
<tr><td>4..2</td><td>parity_sel</td><td>R/W</td><td>0</td><td><table>
<tr><th>Parity Sel Bits</th><th>Parity</th></tr>
<tr><td>xx0</td><td>No Parity</td></tr>
<tr><td>001</td><td>Odd Parity. Parity bit will be set to a ‘1’ or ‘0’ to ensure an odd number of 1s are sent</td></tr>
<tr><td>011</td><td>Even Parity. Parity bit will be set to a ‘1’ or ‘0’ to ensure an even number of 1s are sent</td></tr>
<tr><td>101</td><td>High Parity. Parity bit is always set high</td></tr>
<tr><td>111</td><td>Low Parity. Parity bit is always set low</td></tr>
</table></td></tr>
<tr><td>1</td><td>stop_2</td><td>R/W</td><td>0</td><td>When 0, one stop bit generated
When 1, two stop bits are generated</td></tr>
<tr><td>0</td><td>size_7</td><td>R/W</td><td>0</td><td>When 0, eight bits of data are transmitted and received
When 1, seven bits of data are transmitted and received</td></tr>
</table>

Table 2.64 UART Line Control Register

2.7.10 UART_BAUD

Bit Position	Bit Field Name	Type	Reset	Description
7..0	uart_baud_0	R/W	0x88	Lower byte of the baud rate setting

Table 2.65 UART Baud Rate 0 Register

Bit Position	Bit Field Name	Type	Reset	Description
7..0	uart_baud_1	R/W	0x13	Middle byte of the baud rate setting

Table 2.66 UART Baud Rate 1 Register

Bit Position	Bit Field Name	Type	Reset	Description
7	Reserved	RFU	0	Reserved
6..4	uart_baud_frac	R/W	0	Tuning bits for the Baud Rate controller. These bits allow the baud clock to be extended by a fraction of one clock cycle Tuning Bits / Clock Extension 000 / Nothing added 001 / Add ½ clock 010 / Add ¼ clock 011 / Add 1/8 clock 100 / Add 3/8 clock 101 / Add 5/8 clock 110 / Add 6/8 clock 111 / Add 7/8 clock
3..2	Reserved	RFU	0	Reserved
1..0	uart_baud_2	R/W	0	Upper two bits of the baud rate setting

Table 2.67 UART Baud Rate 2 Register

2.7.11 UART Baud Rate Example

Baud Rate Example
The Baud Rate is defined by the value programmed into registers `UART_BAUD_0`, `UART_BAUD_1` and `UART_BAUD_2`. This value is used as a divisor of the system clock frequency. For a system clock frequency of 48MHz. The default value of the `UART_BAUD_0`, `UART_BAUD_1`, and `UART_BAUD_2` will be 0x88, 0x13 and 0x00 respectively. This will set the baud rate divisor to be 0x1388 or 5000_{dec}. The final baud rate will be 48000000/5000 = 9600 baud

Figure 2.5 UART Baud Rate Example Calculations

2.7.12 UART_FLOW_CTRL

Bit Position	Bit Field Name	Type	Reset	Description
7..6	Reserved	RFU	0	Reserved
5	`dtr_n_reg`	R/W	0	When 0, the dtr_n signal is under the control of the flow control block When 1, the dtr_n output will be held high
4	`rts_n_reg`	R/W	0	When 0, the rts_n signal is under the control of the flow control block When 1, the rts_n output will be held high
3	`Reserved`	RFU	0	Reserved
2	`Reserved`	RFU	0	Reserved
1	`dtr_dsr`	R/W	0	When set, flow control is set to DTR_DSR mode
0	`rts_cts`	R/W	0	When set, flow control is set to RTS/CTS mode

Table 2.68 UART Flow Control Register

2.7.13 UART_FLOW_STAT

Bit Position	Bit Field Name	Type	Reset	Description
7..4	Reserved	RFU	0	Reserved
3	`ri_reg`	RO	0	Status of the Ring Indicator signal
2	`dcd_reg`	RO	0	Status of the Data Carrier Detect signal
1	`dsr_n_reg`	RO	0	Status of the Data_Set_Ready signal
0	`cts_n_reg`	RO	0	Status of the Clear_to_Send signal

Table 2.69 UART Flow Control Status Register

2.8 GPIOs

GPIOs are divided into Digital and Analogue pads. Digital inputs and outputs should be mapped to the digital pads but can be mapped to analogue pads if required. Analogue inputs and outputs must be mapped to the analogue pads.

2.8.1 Digital GPIO Pads

Up to 16 digital GPIO pads are available depending on package type. The digital pads are multi-speed, multi-voltage and bidirectional I/O.

The digital GPIO is able to operate over wide voltage ranges and the GPIO voltage can be taken above that of the internal supply rails (i.e. up to 5v).

Each of the 16 digital GPIO pads has its own control register as shown in Table 2.70.

I/O Address	Register Name	Description
0x1A	DIGITAL_CONTROL_GPIO_0	Control register for DIO 0
0x1B	DIGITAL_CONTROL_GPIO_1	Control register for DIO 1
0x1C	DIGITAL_CONTROL_GPIO_2	Control register for DIO 2
0x1D	DIGITAL_CONTROL_GPIO_3	Control register for DIO 3
0x1E	DIGITAL_CONTROL_GPIO_4	Control register for DIO 4
0x1F	DIGITAL_CONTROL_GPIO_5	Control register for DIO 5
0x20	DIGITAL_CONTROL_GPIO_6	Control register for DIO 6
0x21	DIGITAL_CONTROL_GPIO_7	Control register for DIO 7
0x22	DIGITAL_CONTROL_GPIO_8	Control register for DIO 8
0x23	DIGITAL_CONTROL_GPIO_9	Control register for DIO 9
0x24	DIGITAL_CONTROL_GPIO_10	Control register for DIO 10
0x25	DIGITAL_CONTROL_GPIO_11	Control register for DIO 11
0x26	DIGITAL_CONTROL_GPIO_12	Control register for DIO 12
0x27	DIGITAL_CONTROL_GPIO_13	Control register for DIO 13
0x28	DIGITAL_CONTROL_GPIO_14	Control register for DIO 14
0x29	DIGITAL_CONTROL_GPIO_15	Control register for DIO 15

Table 2.70 GPIO DIO Digital Control Register Addresses

2.8.1.1 DIGITAL_CONTROL_GPIO_0 to DIGITAL_CONTROL_GPIO_15

Bit Position	Bit Field Name	Type	Reset	Description
7..6	RFU	R	0	Reserved
5	`Sr`	R/W	0	Slew Rate Control. When `sr=0` slew RATE is NORMAL. When `sr=1` slew RATE is SLOW
4	`Smt`	R/W	0	Schmitt Trigger Enable. When `smt=1` the Schmitt circuit is enabled, giving hysteresis on the input signal. When `smt=0` there is no hysteresis.
3	`pdena`	R/W	0	Pull down Enable. When this signal is set, a weak internal pull down is enabled to hold the pad in a "low" logic state if the pad is left unconnected or tri-state
2	`puena`	R/W	0	Pull up Enable. When this signal is set, a weak internal pull up is enabled to hold the pad in a "high" logic state if the pad is left unconnected or tri-state
1..0	`drive_strength`	R/W	0	Drive strength control. bit 1 / bit 0 / Drive 0 / 1 / Weak 0 / 0 / Low 1 / 0 / Medium 1 / 1 / High

Table 2.71 GPIO DIO Digital Control Registers

Please refer to the FT51A series datasheet for electrical information.

Note: Do NOT set `puena` or `pdena` at the same time. This can place the port in an undetermined state.

2.8.2 Analogue GPIO Pads

Up to 16 analogue I/O pads are available depending on package type. The analogue pads are multi-speed, multi-voltage and bidirectional I/O. Each pad can function in either analogue or digital mode, but not both modes at the same time.

For digital mode of operation each of the 16 AIO pads has its own control register shown in Table 2.72. The AIO_MODE register described in Section 2.10.3 is used to switch from analogue mode to digital mode.

I/O Address	Register Name	Description
0x0A	DIGITAL_CONTROL_AIO_0	Control register for AIO 0 in digital mode
0x0B	DIGITAL_CONTROL_AIO_1	Control register for AIO 1 in digital mode
0x0C	DIGITAL_CONTROL_AIO_2	Control register for AIO 2 in digital mode
0x0D	DIGITAL_CONTROL_AIO_3	Control register for AIO 3 in digital mode
0x0E	DIGITAL_CONTROL_AIO_4	Control register for AIO 4 in digital mode
0x0F	DIGITAL_CONTROL_AIO_5	Control register for AIO 5 in digital mode
0x10	DIGITAL_CONTROL_AIO_6	Control register for AIO 6 in digital mode
0x11	DIGITAL_CONTROL_AIO_7	Control register for AIO 7 in digital mode
0x12	DIGITAL_CONTROL_AIO_8	Control register for AIO 8 in digital mode
0x13	DIGITAL_CONTROL_AIO_9	Control register for AIO 9 in digital mode
0x14	DIGITAL_CONTROL_AIO_10	Control register for AIO 10 in digital mode
0x15	DIGITAL_CONTROL_AIO_11	Control register for AIO 11 in digital mode
0x16	DIGITAL_CONTROL_AIO_12	Control register for AIO 12 in digital mode
0x17	DIGITAL_CONTROL_AIO_13	Control register for AIO 13 in digital mode
0x18	DIGITAL_CONTROL_AIO_14	Control register for AIO 14 in digital mode
0x19	DIGITAL_CONTROL_AIO_15	Control register for AIO 15 in digital mode

Table 2.72 GPIO AIO Digital Control Register Addresses

2.8.2.1 DIGITAL_CONTROL_AIO_0 to DIGITAL_CONTROL_AIO_15

Bit Position	Bit Field Name	Type	Reset	Description
7..4	RFU	R	0	Reserved
3	pdena	R/W	0	Pull down Enable. When this signal is set, a weak internal pull down is enabled to hold the pad in a "low" logic state if the pad is left unconnected or tri-state
2	puena	R/W	0	Pull up Enable. When this signal is set, a weak internal pull up is enabled to hold the pad in a "high" logic state if the pad is left unconnected or tri-state
1..0	drive_strength	R/W	0	Drive strength control. bit 1 / bit 0 / Drive 0 / 1 / Weak 0 / 0 / Low 1 / 0 / Medium 1 / 1 / High

Table 2.73 GPIO AIO Digital Control Registers

Please refer to the FT51A series datasheet for electrical information.

Note: Do NOT set puena or pdena at the same time. This can place the port in an undetermined state.

2.9 IOMUX

The IOMUX allows any peripheral interface input or output signal to be assigned to any of the available IO pins. There are however some limitations:

- Digital interfaces (SPI, I2C, UART, etc.) should be assigned to one of the Digital IO Pads. However, the digital interfaces can be mapped to the analogue pads if required.
- Analogue signals (ADC) must be mapped to the Analogue IO Pads.

In order to assign a signal to a particular pin, two register writes are required to select the signal and select the pad. Pads are routed to specific IC pins depending on the IC package selected.

Pads are described in Section 2.9.6 and Input and Output signals are described in Sections 2.9.8 and 2.9.7 respectively.

The registers associated with the IOMUX are outlined in Table 2.74.

I/O Address	Register Name	Description
0x40	IOMUX_CONTROL	Used to enable and reset the IOMUX
0x41	IOMUX_OUTPUT_PAD_SEL	Used to route an output signal to a specific pad
0x42	IOMUX_OUTPUT_SIG_SEL	Used to select a pad for a specific output signal
0x43	IOMUX_INPUT_SIG_SEL	Used to route an input signal from a specific pad
0x44	IOMUX_INPUT_PAD_SEL	Used to select pad to be used with a specific input signal

Table 2.74 IOMUX Register Addresses

2.9.1 IOMUX_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	`iomux_dev_en`	R/W	0	Write 1 to Enable IOMUX
0	`iomux_soft_reset`	R/W	0	Write 1 to Reset IOMUX

Table 2.75 IOMUX Control Register

The IOMUX Control register provides top-level enables and reset functions for the IOMUX module.

The IOMUX module is enabled by setting the `iomux_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `iomux_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.9.2 IOMUX_OUTPUT_PAD_SEL

Bit Position	Bit Field Name	Type	Reset	Description
7..0	op_pad_sel	R/W	0x00	Pad to be used for an output mapping.

Table 2.76 IOMUX Output Pad Select Register

This register selects the output pad for a signal mapping. The pad is selected from Table 2.80. Note that the values in Table 2.80 are decimal values.

Writing to this register has no effect on the IOMUX module until a write to the IOMUX_OUTPUT_SIG_SEL occurs. At this time, the value in this register is sampled and used for the output pad in the mapping.

2.9.3 IOMUX_OUTPUT_SIG_SEL

Bit Position	Bit Field Name	Type	Reset	Description
7..0	op_sig_sel	R/W	0x00	Signal to be used for an output mapping.

Table 2.77 IOMUX Output Signal Select Register

This describes a signal from Table 2.81 to be mapped to an output pad.

When this register is written the pad selected in the IOMUX_OUTPUT_PAD_SEL register is mapped to the required signal.

Note: When selecting an output signal to map to a pad, write the op_pad_sel first and then the op_sig_sel second.

2.9.4 IOMUX_INPUT_SIG_SEL

Bit Position	Bit Field Name	Type	Reset	Description
7..0	ip_sig_sel	R/W	0x00	Signal to be used for an input mapping.

Table 2.78 IOMUX Input Signal Select Register

This describes a signal from Table 2.82 to be mapped to an input pad.

Writing to this register has no effect on the IOMUX module until a write to the IOMUX_INPUT_PAD_SEL occurs. At this time, the value in this register is sampled and used for the input signal in the mapping.

2.9.5 IOMUX_INPUT_PAD_SEL

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`ip_pad_sel`	R/W	0x00	Pad to be used for an input mapping.

Table 2.79 IOMUX Input Pad Select Register

This register selects the input pad for a signal mapping. The pad is selected from Table 2.80. Note that the values in Table 2.80 are decimal values.

When this register is written the signal selected in the `IOMUX_INPUT_SIG_SEL` register is mapped to the required pad.

Note: When selecting an input signal to map to a pad, write the `ip_sig_sel` first and then the `ip_pad_sel` second.

2.9.6 IOMUX Pad Values

There are 32 pads available. 16 are analogue capable and 16 are digital.

Note: The values listed in Table 2.80 are decimal values.

IOMUX Pad	Value
AIO_0	0
AIO_1	1
AIO_2	2
AIO_3	3
AIO_4	4
AIO_5	5
AIO_6	6
AIO_7	7
AIO_8	8
AIO_9	9
AIO_10	10
AIO_11	11
AIO_12	12
AIO_13	13
AIO_14	14
AIO_15	15

IOMUX Pad	Value
DIO_0	16
DIO_1	17
DIO_2	18
DIO_3	19
DIO_4	20
DIO_5	21
DIO_6	22
DIO_7	23
DIO_8	24
DIO_9	25
DIO_10	26
DIO_11	27
DIO_12	28
DIO_13	29
DIO_14	30
DIO_15	31

Table 2.80 IOMUX Pad Values

2.9.7 IOMUX Output Signal Mapping Values

Table 2.81 contains the output signal selection values to map to output pads.

Note: The signal values used in Table 2.81 are in decimal.

IOMUX Output Signals	Value
I2C_MASTER_SDA	82
I2C_MASTER_SCL	81
HUB_P2_PWREN	80
GPIO_PORT3O_7	79
GPIO_PORT3O_6	78
GPIO_PORT3O_5	77
GPIO_PORT3O_4	76
GPIO_PORT3O_3	75
GPIO_PORT3O_2	74
GPIO_PORT3O_1	73
GPIO_PORT3O_0	72
GPIO_PORT2O_7	71
GPIO_PORT2O_6	70
GPIO_PORT2O_5	69
GPIO_PORT2O_4	68
GPIO_PORT2O_3	67
GPIO_PORT2O_2	66
GPIO_PORT2O_1	65
GPIO_PORT2O_0	64
MISC_TRISTATE	63
MISC_HIGH	62
MISC_LOW	61
MISC_DEBUGGER	60
MISC_BCD	56
FIFO_245_RXF_N	55
FIFO_245_TXE_N	54
FIFO_245_DATA_READ_7	53
FIFO_245_DATA_READ_6	52
FIFO_245_DATA_READ_5	51
FIFO_245_DATA_READ_4	50
FIFO_245_DATA_READ_3	49
FIFO_245_DATA_READ_2	48
FIFO_245_DATA_READ_1	47
FIFO_245_DATA_READ_0	46
PWM_OUT_7	45
PWM_OUT_6	44

IOMUX Output Signals	Value
PWM_OUT_13	39
PWM_OUT_03	38
I2C_MASTER_SCLH	37
I2C_SDA	36
I2C_SCL	35
SPI_SLAVE_MOSI_OUT	34
SPI_SLAVE_MISO	33
SPI_MASTER_SS_N_0	32
SPI_MASTER_SS_N_1	31
SPI_MASTER_SS_N_2	30
SPI_MASTER_SS_N_3	29
SPI_MASTER_SCLK	28
SPI_MASTER_MISO_ LOOPBACK	27
SPI_MASTER_MOSI	26
CLKOUT	25
SUSPEND_OPEN_DRAIN	24
SUSPEND	23
GL_N	22
UART_TX_ACTIVE	21
UART_DTR_N	20
UART_RTS_N	19
UART_TXD	18
UART_8051_RXD0O	17
UART_8051_TXD0	16
GPIO_PORT1O_7	15
GPIO_PORT1O_6	14
GPIO_PORT1O_5	13
GPIO_PORT1O_4	12
GPIO_PORT1O_3	11
GPIO_PORT1O_2	10
GPIO_PORT1O_1	9
GPIO_PORT1O_0	8
GPIO_PORT0O_7	7
GPIO_PORT0O_6	6
GPIO_PORT0O_5	5
GPIO_PORT0O_4	4

IOMUX Output Signals	Value
PWM_OUT_5	43
PWM_OUT_44	42
PWM_OUT_34	41
PWM_OUT_24	40

IOMUX Output Signals	Value
GPIO_PORT0O_3	3
GPIO_PORT0O_2	2
GPIO_PORT0O_1	1
GPIO_PORT0O_0	0

Table 2.81 IOMUX Output Signal Mapping Values

2.9.8 IOMUX Input Signal Mapping Values

Table 2.82 contains the input signal selection values to map to input pads.

Note: The signal values used in Table 2.82 are in decimal.

IOMUX Input Signals	Value
I2C_MASTER_SDA	60
I2C_MASTER_SCL	59
HUB_P2_OVER_CURRENT	58
HUB_P1_OVER_CURRENT	57
GPIO_PORT3I_7	56
GPIO_PORT3I_6	55
GPIO_PORT3I_5	54
GPIO_PORT3I_4	53
GPIO_PORT3I_3	52
GPIO_PORT3I_2	51
GPIO_PORT3I_1	50
GPIO_PORT3I_0	49
GPIO_PORT2I_7	48
GPIO_PORT2I_6	47
GPIO_PORT2I_5	46
GPIO_PORT2I_4	45
GPIO_PORT2I_3	44
GPIO_PORT2I_2	43
GPIO_PORT2I_1	42
GPIO_PORT2I_0	41
MISC_VBUS_DETECTED	40
MISC_PWM_TRIGGER	39
MISC_WAKEUP_PIN	38
FIFO_245_RD_N	37
FIFO_245_WR_N	36
FIFO_245_DATA_WRITE_7	35

IOMUX Input Signals	Value
FIFO_245_DATA_WRITE_1	29
FIFO_245_DATA_WRITE_0	28
I2C_SDA	27
I2C_SCL	26
SPI_SLAVE_SS_N	25
SPI_SLAVE_SCLK	24
SPI_SLAVE_MOSI_IN	23
SPI_MASTER_MISO	22
UART_DCD	21
UART_RI	20
UART_DSR_N	19
UART_CTS_N	18
UART_RXD	17
UART_8051_RXD0I	16
GPIO_PORT1I_7	15
GPIO_PORT1I_6	14
GPIO_PORT1I_5	13
GPIO_PORT1I_4	12
GPIO_PORT1I_3	11
GPIO_PORT1I_2	10
GPIO_PORT1I_1	9
GPIO_PORT1I_0	8
GPIO_PORT0I_7	7
GPIO_PORT0I_6	6
GPIO_PORT0I_5	5
GPIO_PORT0I_4	4

IOMUX Input Signals	Value
FIFO_245_DATA_WRITE_6	34
FIFO_245_DATA_WRITE_5	33
FIFO_245_DATA_WRITE_4	32
FIFO_245_DATA_WRITE_3	31
FIFO_245_DATA_WRITE_2	30

IOMUX Input Signals	Value
GPIO_PORT0I_3	3
GPIO_PORT0I_2	2
GPIO_PORT0I_1	1
GPIO_PORT0I_0	0

Table 2.82 IOMUX Input Signal Mapping Values

2.9.9 Use Cases

The following use cases represent methods for mapping input and output signals to different pads. It should be noted that the IOMUX mapping MUST be performed during configuration of the device and not while the signals being mapped are active.

2.9.9.1 Setup an Input Signal

To program the IOMUX to route DIO_6 to the input signal UART_RXD.

- Select the signal:
  - Find UART_RXD in Table 2.82. Value is $17_{decimal}$.
  - Write $17_{decimal}$ to `IOMUX_INPUT_SIG_SEL`.
- Select the pad:
  - Find the pad DIO_6 in Table 2.80. Value is $22_{decimal}$.
  - Write $22_{decimal}$ to `IOMUX_INPUT_PAD_SEL`.

2.9.9.2 Setup an Output Signal

To program the IOMUX to route UART_TXD signal to the output pad DIO_8.

- Select the pad:
  - Find the pad DIO_8 in Table 2.80. Value is $24_{decimal}$.
  - Write $24_{decimal}$ to `IOMUX_OUTPUT_PAD_SEL`
- Select the signal:
  - Find UART_RXD in Table 2.81. Value is $18_{decimal}$.
  - Write $18_{decimal}$ to `IOMUX_OUTPUT_SIG_SEL`.

2.10 Analogue IO Ports

The FT51A has up to 16 Analogue IO pads available, depending on package type. The number of pads for each package type is shown in Table 2.83.

Package	Number of Available AIOs	AIO Pads Available
48 pin	16	AIO0 to AIO15
44 pin	16	AIO0 to AIO15
32 pin	8	AIO4 to AIO7, AIO10,AIO11,AIO14,AIO15
28 pin	8	AIO4 to AIO7, AIO10,AIO11,AIO14,AIO15

Table 2.83 Available AIO Ports

Each AIO pad can be configured to operate in one of the following modes:

- Digital – Where the Analogue ports behave as Digital I/Os and can be controlled in a similar fashion to Digital ports. See Section 2.8.2.
- ADC – Where the pad is an input and can be sampled to perform analogue to digital conversion.

A special Global Mode is implemented to allow multiple ADC conversions to be run simultaneously.

Address	Register Name	Description
0x100	AIO_CONTROL	Used to enable and reset the Analogue IO

Table 2.84 Analogue IO Register Addresses

2.10.1 AIO_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	aio_dev_en	R/W	0	Write 1 to Enable Analogue IO
0	aio_soft_reset	R/W	0	Write 1 to Reset Analogue IO

Table 2.85 Analogue IO Control Register

The Analogue IO Control register provides top-level enables and reset functions for the Analogue IO module.

The Analogue IO module is enabled by setting the `aio_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `aio_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.10.2 Implementation

Analogue-to-digital conversion is implemented using a DAC to successively approximate (in hardware) the output of a sample-and-hold circuit. Physically, there are four DACs, each shared by four AIO ports, as illustrated below.

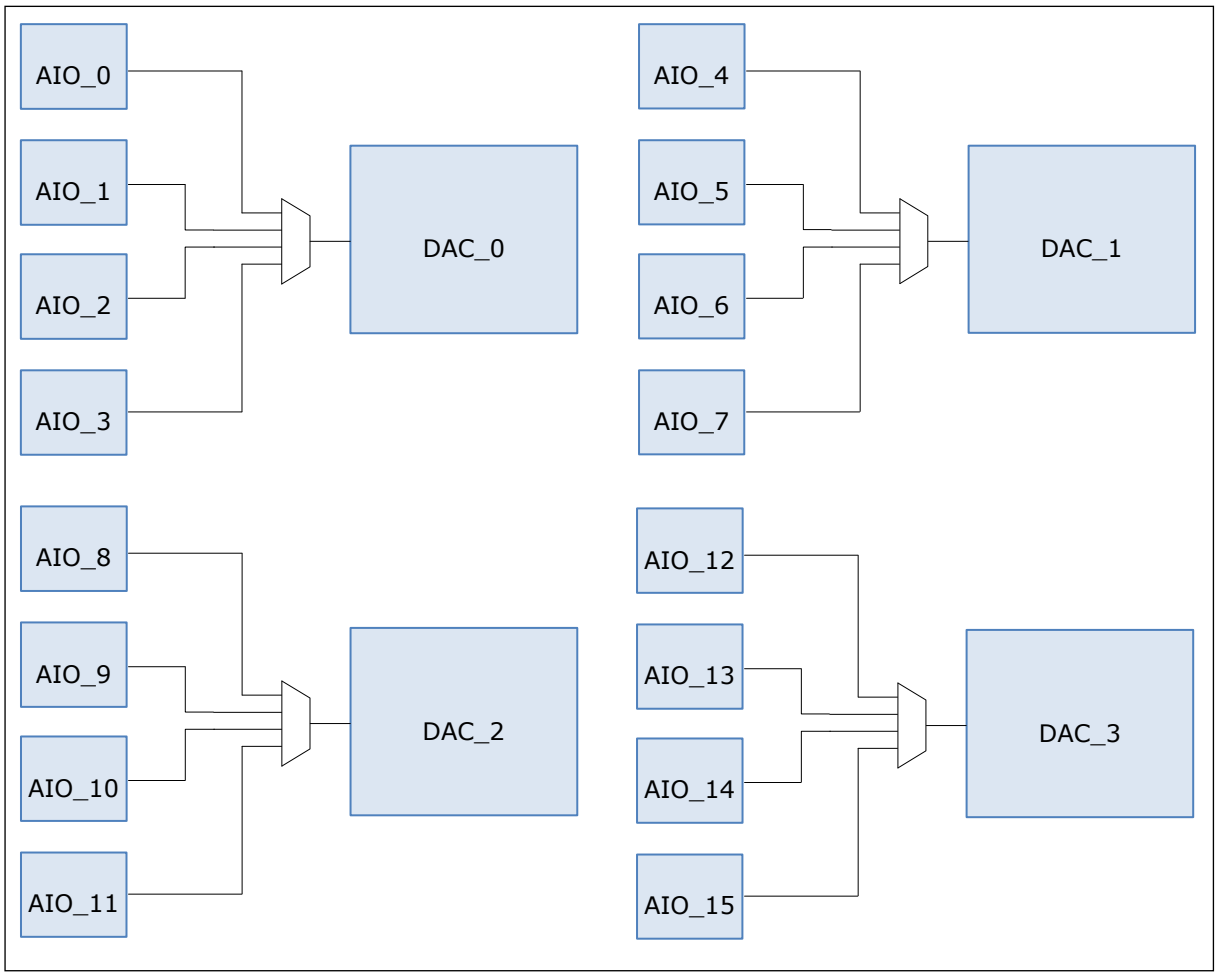


Figure 2.6 Pad Distribution

2.10.3 AIO Configuration

The mode of each AIO port can be selected by writing to the appropriate mode register. Each register sets the mode for 4 AIO ports with the bits defined in Table 2.87.

Address	Register Name	Description
0x102	AIO_MODE_0	Selects AIO ports 0 - 3
0x103	AIO_MODE_1	Selects AIO ports 4 - 7
0x104	AIO_MODE_2	Selects AIO ports 8 - 11
0x105	AIO_MODE_3	Selects AIO ports 12 - 15

Table 2.86 AIO Mode Control Register Addresses

mode1	mode0	Configuration
0	0	Analogue off. Pad configured for Digital Mode.
0	1	Reserved
1	0	ADC Mode
1	1	Reserved

Table 2.87 AIO Mode Control Bits

2.10.3.1 AIO_MODE_0

Bit Position	Bit Field Name	Type	Reset	Description
7..6	AIO_3_MODE	R/W	0	Analogue mode of operation for AIO_3 port. See Table 2.87
5..4	AIO_2_MODE	R/W	0	Analogue mode of operation for AIO_2 port. See Table 2.87
3..2	AIO_1_MODE	R/W	0	Analogue mode of operation for AIO_1 port. See Table 2.87
1..0	AIO_0_MODE	R/W	0	Analogue mode of operation for AIO_0 port. See Table 2.87

Table 2.88 AIO Mode Control 0 Register

2.10.3.2 AIO_MODE_1

Bit Position	Bit Field Name	Type	Reset	Description
7..6	AIO_7_MODE	R/W	0	Analogue mode of operation for AIO_7 port. See Table 2.87
5..4	AIO_6_MODE	R/W	0	Analogue mode of operation for AIO_6 port. See Table 2.87
3..2	AIO_5_MODE	R/W	0	Analogue mode of operation for AIO_5 port. See Table 2.87
1..0	AIO_4_MODE	R/W	0	Analogue mode of operation for AIO_4 port. See Table 2.87

Table 2.89 AIO Mode Control 1 Register

2.10.3.3 AIO_MODE_2

Bit Position	Bit Field Name	Type	Reset	Description
7..6	AIO_11_MODE	R/W	0	Analogue mode of operation for AIO_11 port. See Table 2.87
5..4	AIO_10_MODE	R/W	0	Analogue mode of operation for AIO_10 port. See Table 2.87
3..2	AIO_9_MODE	R/W	0	Analogue mode of operation for AIO_9 port. See Table 2.87
1..0	AIO_8_MODE	R/W	0	Analogue mode of operation for AIO_8 port. See Table 2.87

Table 2.90 AIO Mode Control 2 Register

2.10.3.4 AIO_MODE_3

Bit Position	Bit Field Name	Type	Reset	Description
7..6	AIO_15_MODE	R/W	0	Analogue mode of operation for AIO_15 port. See Table 2.87
5..4	AIO_14_MODE	R/W	0	Analogue mode of operation for AIO_14 port. See Table 2.87
3..2	AIO_13_MODE	R/W	0	Analogue mode of operation for AIO_13 port. See Table 2.87
1..0	AIO_12_MODE	R/W	0	Analogue mode of operation for AIO_12 port. See Table 2.87

Table 2.91 AIO Mode Control 3 Register

2.10.4 AIO ADC Mode

During ADC the incoming analogue signal to the AIO pad is sampled to provide a digital representation of the wave.

The AIO_SAMPLE_0 and AIO_SAMPLE_1 registers determine which AIO ports shall be sampled by the analogue to digital convertor.

Once the conversion is completed the interrupt bit for each AIO port is set in the AIO_INTERRUPTS_0_7 or AIO_INTERRUPTS_8_15 register. This signals that the digital representation of the analogue value can be read from the ADC data registers, AIO_x_ADC_DATA_L and AIO_x_ADC_DATA_U. The reading is encoded into bits 0..9. The 2 lowest bits (0..1) should be discarded.

Address	Register Name	Description
0x108	AIO_SAMPLE_0	Sample AIO ports 0 – 7
0x109	AIO_SAMPLE_1	Sample AIO ports 8 – 15
0x13E 0x13F	AIO_0_ADC_DATA_L AIO_0_ADC_DATA_U	Lower byte of AIO_0 ADC data Upper 2 bits of AIO_0 ADC data
0x140 0x141	AIO_1_ADC_DATA_L AIO_1_ADC_DATA_U	Lower byte of AIO_1 ADC data Upper 2 bits of AIO_1 ADC data
0x142 0x143	AIO_2_ADC_DATA_L AIO_2_ADC_DATA_U	Lower byte of AIO_2 ADC data Upper 2 bits of AIO_2 ADC data
0x144 0x145	AIO_3_ADC_DATA_L AIO_3_ADC_DATA_U	Lower byte of AIO_3 ADC data Upper 2 bits of AIO_3 ADC data
0x146 0x147	AIO_4_ADC_DATA_L AIO_4_ADC_DATA_U	Lower byte of AIO_4 ADC data Upper 2 bits of AIO_4 ADC data
0x148 0x149	AIO_5_ADC_DATA_L AIO_5_ADC_DATA_U	Lower byte of AIO_5 ADC data Upper 2 bits of AIO_5 ADC data
0x14A 0x14B	AIO_6_ADC_DATA_L AIO_6_ADC_DATA_U	Lower byte of AIO_6 ADC data Upper 2 bits of AIO_6 ADC data
0x14C 0x14D	AIO_7_ADC_DATA_L AIO_7_ADC_DATA_U	Lower byte of AIO_7 ADC data Upper 2 bits of AIO_7 ADC data
0x14E 0x14F	AIO_8_ADC_DATA_L AIO_8_ADC_DATA_U	Lower byte of AIO_8 ADC data Upper 2 bits of AIO_8 ADC data
0x150 0x151	AIO_9_ADC_DATA_L AIO_9_ADC_DATA_U	Lower byte of AIO_9 ADC data Upper 2 bits of AIO_9 ADC data
0x152 0x153	AIO_10_ADC_DATA_L AIO_10_ADC_DATA_U	Lower byte of AIO_10 ADC data Upper 2 bits of AIO_10 ADC data
0x154 0x155	AIO_11_ADC_DATA_L AIO_11_ADC_DATA_U	Lower byte of AIO_11 ADC data Upper 2 bits of AIO_11 ADC data
0x156 0x157	AIO_12_ADC_DATA_L AIO_12_ADC_DATA_U	Lower byte of AIO_12 ADC data Upper 2 bits of AIO_12 ADC data
0x158 0x159	AIO_13_ADC_DATA_L AIO_13_ADC_DATA_U	Lower byte of AIO_13 ADC data Upper 2 bits of AIO_13 ADC data
0x15A	AIO_14_ADC_DATA_L	Lower byte of AIO_14 ADC data

0x15B	AIO_14_ADC_DATA_U	Upper 2 bits of AIO_14 ADC data
0x15C 0x15D	AIO_15_ADC_DATA_L AIO_15_ADC_DATA_U	Lower byte of AIO_15 ADC data Upper 2 bits of AIO_15 ADC data

Table 2.92 AIO ADC Register Addresses

2.10.4.1 AIO_SAMPLE_0

Bit Position	Bit Field Name	Type	Reset	Description
7	AIO_7_SAMPLE	W	0	Sample AIO_7 port
6	AIO_6_SAMPLE	W	0	Sample AIO_6 port
5	AIO_5_SAMPLE	W	0	Sample AIO_5 port
4	AIO_4_SAMPLE	W	0	Sample AIO_4 port
3	AIO_3_SAMPLE	W	0	Sample AIO_3 port
2	AIO_2_SAMPLE	W	0	Sample AIO_2 port
1	AIO_1_SAMPLE	W	0	Sample AIO_1 port
0	AIO_0_SAMPLE	W	0	Sample AIO_0 port

Table 2.93 AIO ADC Sample Select 0 Register

2.10.4.2 AIO_SAMPLE_1

Bit Position	Bit Field Name	Type	Reset	Description
7	AIO_15_SAMPLE	W	0	Sample AIO_15 port
6	AIO_14_SAMPLE	W	0	Sample AIO_14 port
5	AIO_13_SAMPLE	W	0	Sample AIO_13 port
4	AIO_12_SAMPLE	W	0	Sample AIO_12 port
3	AIO_11_SAMPLE	W	0	Sample AIO_11 port
2	AIO_10_SAMPLE	W	0	Sample AIO_10 port
1	AIO_9_SAMPLE	W	0	Sample AIO_9 port
0	AIO_8_SAMPLE	W	0	Sample AIO_8 port

Table 2.94 AIO ADC Sample Select 1 Register

2.10.4.3 AIO_x_ADC_DATA_L

Bit Position	Bit Field Name	Type	Reset	Description
7..0	LOWER DATA	R	0	Digital representation of the analogue value (lower). Discard bits 0..1

Table 2.95 AIO ADC Sample Result (Lower) Registers

2.10.4.4 AIO_x_ADC_DATA_U

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Always reads as zero
1:0	UPPER DATA	R	0	Digital representation of the analogue value (upper).

Table 2.96 AIO ADC Sample Result (Upper) Registers

2.10.5 AIO Interrupts

Address	Register Name	Description
0x16E	AIO_INTERRUPTS_0_7	Interrupts for Ports 0-7
0x170	AIO_INTERRUPTS_8_15	Interrupts for Ports 8-15
0x16F	AIO_INTERRUPT_ENABLES_0_7	Interrupt Enables for Ports 0-7
0x171	AIO_INTERRUPT_ENABLES_8_15	Interrupt Enables for Ports 8-15

Table 2.97 AIO Interrupt Register Addresses

2.10.5.1 AIO_INTERRUPTS_0_7

Bit Position	Bit Field Name	Type	Reset	Description
7	AIO_7_INT	RO	0	Set when an A-D conversion is complete for that particular Analogue Cell.
6	AIO_6_INT	RO	0	
5	AIO_5_INT	RO	0	
4	AIO_4_INT	RO	0	
3	AIO_3_INT	RO	0	
2	AIO_2_INT	RO	0	
1	AIO_1_INT	RO	0	
0	AIO_0_INT	RO	0	

Table 2.98 AIO Interrupts 0-7 Register

2.10.5.2 AIO_INTERRUPTS_8_15

Bit Position	Bit Field Name	Type	Reset	Description
7	AIO_15_INT	RO	0	Set when an A-D conversion is complete for that particular Analogue Cell.
6	AIO_14_INT	RO	0	
5	AIO_13_INT	RO	0	
4	AIO_12_INT	RO	0	
3	AIO_11_INT	RO	0	
2	AIO_10_INT	RO	0	
1	AIO_9_INT	RO	0	
0	AIO_8_INT	RO	0	

Table 2.99 AIO Interrupts 8-15 Register

2.10.5.3 AIO_INTERRUPT_ENABLES_0_7

Bit Position	Bit Field Name	Type	Reset	Description
7	AIO_7_IEN	RW	0	Interrupt Enable bit. Write a 1 to enable the corresponding interrupt bit in register AIO_INTERRUPTS_0_7
6	AIO_6_IEN	RW	0	
5	AIO_5_IEN	RW	0	
4	AIO_4_IEN	RW	0	
3	AIO_3_IEN	RW	0	
2	AIO_2_IEN	RW	0	
1	AIO_1_IEN	RW	0	
0	AIO_0_IEN	RW	0	

Table 2.100 AIO Interrupt Enables 0-7 Register

2.10.5.4 AIO_INTERRUPT_ENABLES_8_15

Bit Position	Bit Field Name	Type	Reset	Description
7	AIO_15_IEN	RW	0	Interrupt Enable bit. Write a 1 to enable the corresponding interrupt bit in register AIO_INTERRUPTS_8_15
6	AIO_14_IEN	RW	0	
5	AIO_13_IEN	RW	0	
4	AIO_12_IEN	RW	0	
3	AIO_11_IEN	RW	0	
2	AIO_10_IEN	RW	0	
1	AIO_9_IEN	RW	0	
0	AIO_8_IEN	RW	0	

Table 2.101 AIO Interrupt Enables 8-15 Register

2.10.6 Global Mode

Global mode allows multiple ADC ports to be sampled simultaneously, i.e. there is no need to assert each individual sample bit. Any ADCs that are selected to be in Global mode will be sampled when the Global Sample bit is asserted.

For ports that are in ADC mode, Global Update must be asserted in order to transfer the results to the ADC Data registers once the sample is complete.

The following registers are used in Global Mode.

Address	Register Name	Description
0x101	AIO_GLOBAL_CTRL	Interrupt enables and status bits
0x10B	AIO_GLOBAL_PORT_SELECT_0_7	Used to include ports 0-7 in the global list
0x10C	AIO_GLOBAL_PORT_SELECT_8_15	Used to include ports 8-15 in the global list

Table 2.102 AIO Global Mode Register Addresses

2.10.6.1 AIO_GLOBAL_CTRL

Bit Position	Bit Field Name	Type	Reset	Description
7..6	Reserved	RFU	0	Reserved
5	`global_update_ien`	R/W	0	Write 1 to enable the Global Update interrupt bit, `global_update_int`
4	`global_update_int`	R/W1C	0	Global Update Interrupt bit. Set when ADC data has been transferred from holding buffers to `AIO_ADC_DATA_L` and `AIO_ADC_DATA_U` registers
3	`global_sample_ien`	R/W	0	Write 1 to enable the Global Sample Interrupt bit, `global_sample_int`
2	`global_sample_int`	R/W1C	0	Set when a global sample has completed. This means that any ADC conversions that were initiated by a `global_sample` are now complete
1	`global_update`	R/W	0	Write a 1 to transfer the resulting data ADC conversions from holding registers to the `AIO_ADC_DATA_L` and `AIO_ADC_DATA_U` registers
0	`global_sample`	R/W	0	Write 1 to start a global sample. This will initiate an ADC sample for all ports that are in the global list

2.10.6.2 AIO_GLOBAL_PORT_SELECT_0_7

Bit Position	Bit Field Name	Type	Reset	Description
7	aio_port_7_active	R/W	0	Write 1 to include Port in Global List
6	aio_port_6_active	R/W	0	Write 1 to include Port in Global List
5	aio_port_5_active	R/W	0	Write 1 to include Port in Global List
4	aio_port_4_active	R/W	0	Write 1 to include Port in Global List
3	aio_port_3_active	R/W	0	Write 1 to include Port in Global List
2	aio_port_2_active	R/W	0	Write 1 to include Port in Global List
1	aio_port_1_active	R/W	0	Write 1 to include Port in Global List
0	aio_port_0_active	R/W	0	Write 1 to include Port in Global List

Table 2.103 AIO Global Mode Select 0-7 Register

2.10.6.3 AIO_GLOBAL_PORT_SELECT_8_15

Bit Position	Bit Field Name	Type	Reset	Description
7	aio_port_15_active	R/W	0	Write 1 to include Port in Global List
6	aio_port_14_active	R/W	0	Write 1 to include Port in Global List
5	aio_port_13_active	R/W	0	Write 1 to include Port in Global List
4	aio_port_12_active	R/W	0	Write 1 to include Port in Global List
3	aio_port_11_active	R/W	0	Write 1 to include Port in Global List
2	aio_port_10_active	R/W	0	Write 1 to include Port in Global List
1	aio_port_9_active	R/W	0	Write 1 to include Port in Global List
0	aio_port_8_active	R/W	0	Write 1 to include Port in Global List

Table 2.104 AIO Global Mode Select 8-15 Register

Global mode limitations

Physically there are only four DACs on the device, shared amongst all available AIO pads (see section 2.10.2 for more detail).

This creates a limitation in Global mode and there is a potential for drift to be seen on the output if more than two ports sharing a DAC are used in Global mode.

Therefore, we recommend that Global mode ports are spread over multiple DACs. Table 2.105 describes the recommended Global port usage.

AIO Port	DAC Used	Recommendation
AIO_0	DAC_0	For this shared DAC, use no more than two ports in Global mode
AIO_1	DAC_0	
AIO_2	DAC_0	
AIO_3	DAC_0	
AIO_4	DAC_1	For this shared DAC, use no more than two ports in Global mode
AIO_5	DAC_1	
AIO_6	DAC_1	
AIO_7	DAC_1	
AIO_8	DAC_2	For this shared DAC, use no more than two ports in Global mode
AIO_9	DAC_2	
AIO_10	DAC_2	
AIO_11	DAC_2	
AIO_12	DAC_3	For this shared DAC, use no more than two ports in Global mode
AIO_13	DAC_3	
AIO_14	DAC_3	
AIO_15	DAC_3	

Table 2.105. Recommended Global Port Selection

2.10.7 Differential Mode

In differential mode, two AIO pads are used together to form a pair of differential pads.

The voltage difference between the two pads will be the input to the ADC circuit.

⚠ *Both inputs must be positive voltages*

Address	Register Name	Description
0x2A	AIO_DIFFERENTIAL_ENABLE	This register is used to configure pairs of pads in differential mode

Table 2.106 AIO Differential Register Addresses

2.10.7.1 AIO_DIFFERENTIAL_ENABLE

Bit Position	Bit Field Name	Type	Reset	Description
7	`diff_14_15`	R/W	0	Write 1 to configure AIO Pads 14 and 15 as a differential pair
6	`diff_12_13`	R/W	0	Write 1 to configure AIO Pads 12 and 13 as a differential pair
5	`diff_10_11`	R/W	0	Write 1 to configure AIO Pads 10 and 11 as a differential pair
4	`diff_8_9`	R/W	0	Write 1 to configure AIO Pads 8 and 9 as a differential pair
3	`diff_6_7`	R/W	0	Write 1 to configure AIO Pads 6 and 7 as a differential pair
2	`diff_4_5`	R/W	0	Write 1 to configure AIO Pads 4 and 5 as a differential pair
1	`diff_2_3`	R/W	0	Write 1 to configure AIO Pads 2 and 3 as a differential pair
0	`diff_0_1`	R/W	0	Write 1 to configure AIO Pads 0 and 1 as a differential pair

Table 2.107 AIO Differential Enable Register

2.10.8 Settling Times

It is possible to vary the amount of time that the AIO module will wait for certain parts of the ADC to complete their particular function.

Delays can be varied for the *Sample & Hold circuit settling time* and for the AIO clock divider.

The following registers are used to increase or decrease delays:

Address	Register Name	Description
0x176 0x177	AIO_SH_COUNTER_L AIO_SH_COUNTER_U	This register allows the user to change the amount of time that the AIO block will wait for the Sample and Hold circuit to complete its function. The value written to these registers will dictate the number of clock cycles that the AIO module will wait for the S&H to complete. This is based on the clock after the CLOCK_DIVIDER ratio has been applied.
0x17A	AIO_CLOCK_DIVIDER	Write to this register to divide the system clock supplied to the AIO Module The divided clock is used to determine the delays applied based on the above registers.

Table 2.108 AIO Settling Times Register Addresses

2.10.8.1 AIO_SH_COUNTER

2.10.8.1.1 AIO_SH_COUNTER_L

Bit Position	Bit Field Name	Type	Reset	Description
7:0	sh_settling_time_l	R/W	0x39	Lower byte of the value used to calculate the number of clock cycles to wait for the Sample & Hold circuit to complete its function

Table 2.109 AIO Cell Sample and Hold Counter Lower Register

2.10.8.1.2 AIO_SH_COUNTER_U

Bit Position	Bit Field Name	Type	Reset	Description
7:2	Reserved	RFU	0	Reserved
1:0	sh_settling_time_u	R/W	0x01	Upper two bits of the value used to calculate the number of clock cycles to wait for the Sample & Hold circuit to complete its function

Table 2.110 AIO Cell Sample and Hold Counter Upper Register

2.10.8.2 Settling Time Examples (for optimal performance)

The ADC function has the following characteristics:

DAC max settling time	0.245µs
Sample & Hold, max settling time	11.8µs

Sample & Hold Settling time

Given the following settings:

System clock frequency	*48MHz*
Clk_div_sel	*0*
SH Settling Time Value	*0x237 (567)*

The divided clock is 48 MHz / (0+1) = 48 MHz, giving a period of 20.83 ns.

Therefore, the allowed Sample & Hold settling time will be 567×20.83 =11.8 us.

For optimal ADC performance, it is recommended that the AIO_SH_COUNTER default value of 0x139 is overwritten with 0x237, i.e. overwrite the AIO_SH_COUNTER_L register with the value 0x37 and the AIO_SH_COUNTER_H register with the value of 0x02.

Assuming a system clock of 48MHz is used and an AIO Clock Division ratio of 0 (default value).

2.10.8.3 AIO_CLOCK_DIVIDER

<table>
<tr><th>Bit Position</th><th>Bit Field Name</th><th>Type</th><th>Reset</th><th>Description</th></tr>
<tr><td>7:0</td><td>clk_div_sel</td><td>R/W</td><td>0</td><td>This value will be used to select the division ratio used to divide the system clock going to the AIO module. The divided clock frequency will be:
sys_clk/clk_div_sel+1
<table><tr><th>clk_div_sel bits</th><th>Clock Divider</th></tr><tr><td>000</td><td>1</td></tr><tr><td>001</td><td>2</td></tr><tr><td>010</td><td>3</td></tr><tr><td>011</td><td>4</td></tr><tr><td>100</td><td>5</td></tr><tr><td>…</td><td>…</td></tr><tr><td>111</td><td>256</td></tr></table></td></tr>
</table>

Table 2.111 Clock Divider Register

2.10.9 ADC Programming Flow

Single Sample

- Write to the appropriate mode register (AIO_MODE_0, AIO_MODE_1, AIO_MODE_2, AIO_MODE_3) to select ADC mode (see AIO_MODE Control AIO_0_MODE to AIO_15_MODE for more details)
- Enable the corresponding interrupt bit in the AIO_INTERRUPT_ENABLES_0_7 or AIO_INTERRUPT_ENABLES_8_15 registersAssert the corresponding Sample bit in the (AIO_SAMPLE_0, AIO_SAMPLE_1) registers
- Wait for the corresponding interrupt bit to be set to indicate that the conversion is complete (AIO_INTERRUPTS_0_7 or AIO_INTERRUPTS_8_15 registers)
- Read digital data (10 bits) from the appropriate AIO_ADC_DATA registers and remove the lower 2 bits.

Global Sample (see Global Mode for more details)

- Write to the AIO_GLOBAL_CTRL registers to select which cells are to be included in global sample
- Write to the appropriate mode register (AIO_MODE_0, AIO_MODE_1, AIO_MODE_2, AIO_MODE_3) to select ADC mode (see AIO_MODE Control AIO_0_MODE to AIO_15_MODE for more details)
- Enable the global_sample_int interrupt with global_sample_int_en in register AIO_GLOBAL_CTRL
- Enable the global_update_int interrupt with global_update_int_en in register AIO_GLOBAL_CTRL
- To initiate a global sample, set the Global_Sample bit in the AIO_GLOBAL_CTRL register
- Once all ports included in the global list have completed their conversions, the global_sample_int bit will be set in the AIO_GLOBAL_CTRL register.
- The result for each ADC port will be stored in a holding register until the Global_Update bit is asserted in the AIO_GLOBAL_CTRL register. At which point the contents of all ADC port holding registers will be transferred into the ADC Data Register
- When the global_update_int bit is set in the AIO_GLOBAL_CTRL register, the ADC result can be read from the corresponding AIO_ADC_DATA registers.

2.11 USB Full Speed Device Controller

The FT51A contains USB Full Speed device controller based on the FT122 technology. The registers to access the device are shown in Table 2.112.

SFR Address	Register Name	Description
0xFC	FT122_CMD	Command register.
0xFD	FT122_DATA	Data register.

Table 2.112 USB Full Speed device controller Register Addresses

The device is addressed by writing the command code from Table 2.117 or Table 2.118 into the FT122_CMD register and then optionally reading or writing data to the FT122_DATA register.

2.11.1 Endpoint Buffer Management

The USB Full Speed device controller has 2 modes of operation for command and memory management: the default mode (FT120 compatible mode) and the enhanced mode. The buffer management schemes are different in these two modes. Upon reset the default mode is functional. The enhanced mode is activated when any of the Set Endpoint Configuration commands (0xB0 to 0xBF) is received.

2.11.1.1 Endpoint Buffer Management in Default Mode

In default mode, the USB Full Speed device controller has 3 bi-directional endpoints (EP0, EP1 and EP2). EP0 is the control endpoint, with 16 bytes maximum packet size for both control OUT and control IN transfers. EP1 can be used as either a bulk endpoint or an interrupt endpoint, with 16 bytes maximum packet size for both OUT and IN transfers. Table 2.113 shows the endpoint type and maximum packet size for EP0 and EP1.

Endpoint Number (EP)	Endpoint Index (EPI)	Endpoint Direction	Transfer Type	Max Packet Size
0	0 1	OUT IN	Control Control	16 16
1	2 3	OUT IN	Bulk/Interrupt Bulk/Interrupt	16 16

Table 2.113 Endpoint Configuration for EP0 and EP1

EP2 is the primary endpoint. It can be configured for either bulk/interrupt or isochronous transfers. The maximum packet size allowed for EP2 is configured using the Set Mode command. Table 2.114 shows all four endpoint configuration modes for EP2.

EP2 Endpoint Configuration Mode	Endpoint Index (EPI)	Endpoint Direction	Transfer Type	Max Packet Size
0 (default)	4 5	OUT IN	Bulk/Interrupt Bulk/Interrupt	64 64
1	4	OUT	Isochronous	128
2	5	IN	Isochronous	128
3	4 5	OUT IN	Isochronous Isochronous	64 64

Table 2.114 Endpoint Configuration for EP2

As the primary endpoint, EP2 is suitable for transmitting or receiving relatively large data. To improve the data throughput, EP2 is implemented with double buffering. This allows concurrent access by the USB bus and the FT51A core. For example, for EP2 IN endpoint (EPI5), the USB host can read data from Buffer 0 while the FT51A core is writing to Buffer 1. The USB host can subsequently read from Buffer 1 without waiting for it to be filled. Buffer switching is handled automatically by the USB Full Speed device controller.

2.11.1.2 Endpoint Buffer Management in Enhanced Mode

In enhanced mode, the USB Full Speed device controller supports a dedicated 1 kB buffer for IN packets and a dedicated 1 kB buffer for OUT packets. The OUT/IN buffer can be allocated to any endpoint with the same direction, up to a maximum of 504 bytes double buffered (1008 bytes in total) to one endpoint. 504 is the maximum byte count as there are 1024 bytes in total per OUT/IN Buffer and 8 bytes for IN and OUT packets on control endpoint 0 must always be reserved. Control, interrupt and bulk endpoints can have a maximum packet size of 64 bytes and only isochronous endpoints can be allocated more than 64 bytes.

Isochronous modes can have larger buffer sizes as USB packets can be larger than 64 bytes for isochronous transfers. The isochronous buffer is managed in the same way as bulk, interrupt and control buffers – i.e. a buffer is for one USB packet only and will not span more than one USB packet.

An example of buffer configurations follows, where Configuration 1 and 2 have larger isochronous buffers. In this example, each row indicates a buffer of 64 bytes - control endpoints are therefore 64 bytes.

<table>
<tr><th colspan="2">Configuration 0</th><th colspan="2">Configuration 1</th><th colspan="2">Configuration 2</th></tr>
<tr><td>EP</td><td>Buffer</td><td>EP</td><td>Buffer</td><td>EP</td><td>Buffer</td></tr>
<tr><td>7</td><td>1</td><td rowspan="2">7
(ISO)</td><td rowspan="2">1
(128 bytes)</td><td rowspan="6">5
(ISO)</td><td rowspan="6">1
(448 bytes)</td></tr>
<tr><td>7</td><td>0</td></tr>
<tr><td>6</td><td>1</td><td rowspan="2">7
(ISO)</td><td rowspan="2">0
(128 bytes)</td></tr>
<tr><td>6</td><td>0</td></tr>
<tr><td>5</td><td>1</td><td>6</td><td>1</td></tr>
<tr><td>5</td><td>0</td><td>6</td><td>0</td></tr>
<tr><td>4</td><td>1</td><td>2</td><td>1</td></tr>
<tr><td>4</td><td>0</td><td>2</td><td>0</td><td rowspan="6">5
(ISO)</td><td rowspan="6">0
(448 bytes)</td></tr>
<tr><td>3</td><td>1</td><td rowspan="3">1
(ISO)</td><td rowspan="3">1
(192 bytes)</td></tr>
<tr><td>3</td><td>0</td></tr>
<tr><td>2</td><td>1</td></tr>
<tr><td>2</td><td>0</td><td rowspan="3">1
(ISO)</td><td rowspan="3">0
(192 bytes)</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Table 2.115 Example Buffer Configuration

The endpoint buffer configurations, settable using the Set Endpoint Configuration command, are as follows:

Endpoint buffer size register setting (binary)	Non-isochronous endpoint	Isochronous endpoint
0000	8 bytes	16 bytes
0001	16 bytes	32 bytes
0010	32 bytes	48 bytes
0011	64 bytes	64 bytes
0100	-	96 bytes
0101	-	128 bytes
0110	-	160 bytes
0111	-	192 bytes
1000	-	256 bytes
1001	-	320 bytes
1010	-	384 bytes
1011	-	504 bytes
1100-1111	-	-

Table 2.116 Endpoint Maximum Packet Size

Note: 504 is the maximum byte count as there are 1024 bytes in total and 8 bytes IN and OUT packets for control endpoint 0 must always be reserved.

2.11.2 Command Summary

Command Name	Target	Code (hex)	Data phase
Initialization Commands			
Set Address Enable	Device	0xD0	Write 1 byte
Set Endpoint Enable	Device	0xD8	Write 1 byte
Set Mode	Device	0xF3	Write 2 bytes
Reserved	Device	0xFB	Write/Read 1 byte
Data Flow Commands			
Read Interrupt Register	Device	0xF4	Read 2 bytes
Select Endpoint	Endpoint 0 OUT	0x00	Read 1 byte (optional)
	Endpoint 0 IN	0x01	Read 1 byte (optional)
	Endpoint 1 OUT	0x02	Read 1 byte (optional)
	Endpoint 1 IN	0x03	Read 1 byte (optional)
	Endpoint 2 OUT	0x04	Read 1 byte (optional)
	Endpoint 2 IN	0x05	Read 1 byte (optional)
Read Last Transaction Status	Endpoint 0 OUT	0x40	Read 1 byte
	Endpoint 0 IN	0x41	Read 1 byte
	Endpoint 1 OUT	0x42	Read 1 byte
	Endpoint 1 IN	0x43	Read 1 byte
	Endpoint 2 OUT	0x44	Read 1 byte
	Endpoint 2 IN	0x45	Read 1 byte
Read Endpoint Status	Endpoint 0 OUT	0x80	Read 1 byte
	Endpoint 0 IN	0x81	Read 1 byte
	Endpoint 1 OUT	0x82	Read 1 byte
	Endpoint 1 IN	0x83	Read 1 byte
	Endpoint 2 OUT	0x84	Read 1 byte
	Endpoint 2 IN	0x85	Read 1 byte
Read Buffer	Selected Endpoint	0xF0	Read multiple bytes
Write Buffer	Selected Endpoint	0xF0	Write multiple bytes
Set Endpoint Status	Endpoint 0 OUT	0x40	Write 1 byte
	Endpoint 0 IN	0x41	Write 1 byte
	Endpoint 1 OUT	0x42	Write 1 byte
	Endpoint 1 IN	0x43	Write 1 byte
	Endpoint 2 OUT	0x44	Write 1 byte
	Endpoint 2 IN	0x45	Write 1 byte
Acknowledge Setup	Selected Endpoint	0xF1	None
Clear Buffer	Selected Endpoint	0xF2	None

Command Name	Target	Code (hex)	Data phase
Validate Buffer	Selected Endpoint	0xFA	None
General Commands			
Read Current Frame Number	Device	0xF5	Read 1 or 2 bytes
Send Resume	Device	0xF6	None

Table 2.117 Default Command Set

Command Name	Target	Code (hex)	Data phase
Initialization Commands			
Set Address Enable	Device	0xD0	Write 1 byte
Set Endpoint Enable	Device	0xD8	Write 1 byte
Set Mode	Device	0xF3	Write 2 bytes
Reserved	Device	0xFB	Write/Read 2 bytes
Set Endpoint Configuration	Endpoint 0 OUT	0xB0	Write 1 byte
	Endpoint 0 IN	0xB1	Write 1 byte
	Endpoint 1 OUT	0xB2	Write 1 byte
	Endpoint 1 IN	0xB3	Write 1 byte
	Endpoint 2 OUT	0xB4	Write 1 byte
	Endpoint 2 IN	0xB5	Write 1 byte
	Endpoint 3 OUT	0xB6	Write 1 byte
	Endpoint 3 IN	0xB7	Write 1 byte
	Endpoint 4 OUT	0xB8	Write 1 byte
	Endpoint 4 IN	0xB9	Write 1 byte
	Endpoint 5 OUT	0xBA	Write 1 byte
	Endpoint 5 IN	0xBB	Write 1 byte
	Endpoint 6 OUT	0xBC	Write 1 byte
	Endpoint 6 IN	0xBD	Write 1 byte
	Endpoint 7 OUT	0xBE	Write 1 byte
	Endpoint 7 IN	0xBF	Write 1 byte
Data Flow Commands			
Read Interrupt Register	Device	0xF4	Read 1 to 4 bytes
Select Endpoint	Endpoint 0 OUT	0x00	Read 1 byte (optional)
	Endpoint 0 IN	0x01	Read 1 byte (optional)
	Endpoint 1 OUT	0x02	Read 1 byte (optional)
	Endpoint 1 IN	0x03	Read 1 byte (optional)
	Endpoint 2 OUT	0x04	Read 1 byte (optional)
	Endpoint 2 IN	0x05	Read 1 byte (optional)
	Endpoint 3 OUT	0x06	Read 1 byte (optional)
	Endpoint 3 IN	0x07	Read 1 byte (optional)
	Endpoint 4 OUT	0x08	Read 1 byte (optional)
	Endpoint 4 IN	0x09	Read 1 byte (optional)
	Endpoint 5 OUT	0x0A	Read 1 byte (optional)
	Endpoint 5 IN	0x0B	Read 1 byte (optional)
	Endpoint 6 OUT	0x0C	Read 1 byte (optional)

Command Name	Target	Code (hex)	Data phase
	Endpoint 6 IN	0x0D	Read 1 byte (optional)
	Endpoint 7 OUT	0x0E	Read 1 byte (optional)
	Endpoint 7 IN	0x0F	Read 1 byte (optional)
Read Last Transaction Status	Endpoint 0 OUT	0x40	Read 1 byte
	Endpoint 0 IN	0x41	Read 1 byte
	Endpoint 1 OUT	0x42	Read 1 byte
	Endpoint 1 IN	0x43	Read 1 byte
	Endpoint 2 OUT	0x44	Read 1 byte
	Endpoint 2 IN	0x45	Read 1 byte
	Endpoint 3 OUT	0x46	Read 1 byte
	Endpoint 3 IN	0x47	Read 1 byte
	Endpoint 4 OUT	0x48	Read 1 byte
	Endpoint 4 IN	0x49	Read 1 byte
	Endpoint 5 OUT	0x4A	Read 1 byte
	Endpoint 5 IN	0x4B	Read 1 byte
	Endpoint 6 OUT	0x4C	Read 1 byte
	Endpoint 6 IN	0x4D	Read 1 byte
	Endpoint 7 OUT	0x4E	Read 1 byte
	Endpoint 7 IN	0x4F	Read 1 byte
Read Endpoint Status	Endpoint 0 OUT	0x80	Read 1 byte
	Endpoint 0 IN	0x81	Read 1 byte
	Endpoint 1 OUT	0x82	Read 1 byte
	Endpoint 1 IN	0x83	Read 1 byte
	Endpoint 2 OUT	0x84	Read 1 byte
	Endpoint 2 IN	0x85	Read 1 byte
	Endpoint 3 OUT	0x86	Read 1 byte
	Endpoint 3 IN	0x87	Read 1 byte
	Endpoint 4 OUT	0x88	Read 1 byte
	Endpoint 4 IN	0x89	Read 1 byte
	Endpoint 5 OUT	0x8A	Read 1 byte
	Endpoint 5 IN	0x8B	Read 1 byte
	Endpoint 6 OUT	0x8C	Read 1 byte
	Endpoint 6 IN	0x8D	Read 1 byte
	Endpoint 7 OUT	0x8E	Read 1 byte
	Endpoint 7 IN	0x8F	Read 1 byte
Read Buffer	Selected Endpoint	0xF0	Read n bytes

Command Name	Target	Code (hex)	Data phase
Write Buffer	Selected Endpoint	0xF0	Write n bytes
Set Endpoint Status	Endpoint 0 OUT	0x40	Write 1 byte
	Endpoint 0 IN	0x41	Write 1 byte
	Endpoint 1 OUT	0x42	Write 1 byte
	Endpoint 1 IN	0x43	Write 1 byte
	Endpoint 2 OUT	0x44	Write 1 byte
	Endpoint 2 IN	0x45	Write 1 byte
	Endpoint 3 OUT	0x46	Write 1 byte
	Endpoint 3 IN	0x47	Write 1 byte
	Endpoint 4 OUT	0x48	Write 1 byte
	Endpoint 4 IN	0x49	Write 1 byte
	Endpoint 5 OUT	0x4A	Write 1 byte
	Endpoint 5 IN	0x4B	Write 1 byte
	Endpoint 6 OUT	0x4C	Write 1 byte
	Endpoint 6 IN	0x4D	Write 1 byte
	Endpoint 7 OUT	0x4E	Write 1 byte
	Endpoint 7 IN	0x4F	Write 1 byte
Acknowledge Setup	Selected Endpoint	0xF1	None
Clear Buffer	Selected Endpoint	0xF2	None
Validate Buffer	Selected Endpoint	0xFA	None
General Commands			
Send Resume	Device	0xF6	None
Read Current Frame Number	Device	0xF5	Read 1 or 2 bytes
Set Buffer Interrupt Mode	Device	0xEC	None

Table 2.118 Enhanced Command Set

2.11.3 Initialization Commands

2.11.3.1 Set Address Enable

Command : 0xD0

Data : Write 1 byte

Bit	Symbol	Reset	Description
6..0	Address	0	USB assigned device address. A bus reset will reset all address bits to 0.
7	Enable	0	Function enable. A bus reset will automatically enable the function at default address 0.

Table 2.119 Address Enable Register

2.11.3.2 Set Endpoint Enable

Command : 0xD8

Data : Write 1 byte

Bit	Symbol	Reset	Description
0	EP_Enable	0	Enable all endpoints (Note EP0 is always enabled regardless the setting of EP_Enable bit). Endpoints can only be enabled when the function is enabled.
7..1	Reserved	0	Reserved, write to 0

Table 2.120 Endpoint Enable Register

2.11.3.3 Set Mode

Command : 0xF3

Data : Write 2 bytes

Bit	Symbol	Reset	Description
0	Reserved	0	Reserved, write to 0
1	No Suspend Clock	1	0: CLKOUT switches to 30 kHz during USB suspend 1: CLKOUT remains unchanged during USB suspend Note: The programmed value is not changed by a bus reset.
2	Clock Running	1	0: internal clocks stop during USB suspend 1: internal clocks continue running during USB suspend This bit must be set to '0' for bus powered applications in order to meet the USB suspend current requirement. Note: The programmed value is not changed by a bus reset.
3	Interrupt Mode	1	0: an interrupt will not be generated on NAK or Error transactions 1: an interrupt will be generated on NAK and Error transactions Note: The programmed value is not changed by a bus reset.
4	DP_Pullup	0	0: Pullup resistor on DP pin disabled 1: Pullup resistor on DP pin enabled when Vbus is present Note: The programmed value is not changed by a bus reset.
5	Reserved	0	Reserved, write to 0
7-6	Endpoint Configuration Mode	0	Set the endpoint configuration mode for EP2. 00: Mode 0 (Non-ISO Mode) 01: Mode 1 (ISO-OUT Mode) 10: Mode 2 (ISO-IN Mode) 11: Mode 3 (ISO-IO Mode) In Enhanced Mode, these 2 bits are reserved. The Endpoint Configuration will be done through separate commands. See "Set Endpoint Configuration" commands.

Table 2.121 Configuration Register (Byte 1)

Bit	Symbol	Reset	Description
3-0	Clock Division Factor	0xB	The Clock Division Factor value (CDF) determines the output clock frequency on the CLKOUT pin. Frequency = 48 MHz / (CDF +1), where CDF ranges 1-12 or the allowed CLKOUT frequency is 4-24 MHz. Default CLKOUT is 4 MHz. When the CDF is programmed to 0xF, the CLKOUT will be turned off. It is recommended to turn off the CLKOUT if it is not used, for power saving. Note: The programmed value is not be changed by a bus reset.
5-4	Reserved	0	Reserved, write to 0
6	SET_TO_ONE	0	This bit must be set to 1
7	SOF-only Interrupt Mode	0	0: normal operation 1: interrupt will generate on receiving SOF packet only.

Table 2.122 Clock Division Factor Register (Byte 2)

2.11.3.4 Set Endpoint Configuration (for Enhanced Mode)

Command : 0xB0-0xBF

Data : Write 1 byte

Bit	Symbol	Reset	Description
0	Endpoint Enabled	0	Enable or disable the endpoint index associated with the command
2-1	Endpoint Type	0	Endpoint type 00: control 01: bulk or interrupt 10: isochronous 11: reserved
6-3	Max Packet Size	0	Maximum USB packet size for this endpoint. Defined by the IN buffer or OUT buffer size for the endpoint. Refer to Section 2.11.1 for full details on the buffer configuration.
7	Reserved	0	Reserved, write to 0

Table 2.123 Endpoint Configuration Register

2.11.4 Data Flow Commands

2.11.4.1 Read Interrupt Register

Command : 0xF4

Data : Read 1 or 2 bytes (Default Mode); Read 1-4 bytes (Enhanced Mode)

Bit	Symbol	Reset	Description
0	Endpoint 0 Out	0	Interrupt for endpoint 0 OUT buffer. Cleared by Read Last Transaction Status command.
1	Endpoint 0 In	0	Interrupt for endpoint 0 IN buffer. Cleared by Read Last Transaction Status command.
2	Endpoint 1 Out	0	Interrupt for endpoint 1 OUT buffer. Cleared by Read Last Transaction Status command.
3	Endpoint 1 In	0	Interrupt for endpoint 1 IN buffer. Cleared by Read Last Transaction Status command.
4	Endpoint 2 Out	0	Interrupt for endpoint 2 OUT buffer. Cleared by Read Last Transaction Status command.
5	Endpoint 2 In	0	Interrupt for endpoint 2 IN buffer. Cleared by Read Last Transaction Status command.
6	Bus Reset	0	Interrupt for bus reset. This bit will be cleared after reading.
7	Suspend Change	0	Interrupt for USB bus suspend status change. This bit will be set to '1' when the USB Full Speed device controller goes to suspend (missing 3 continuous SOFs) or resumes from suspend. This bit will be cleared after reading.

Table 2.124 Interrupt Register Byte 1

Bit	Symbol	Reset	Description
7..0	Reserved	0	Reserved

Table 2.125 Interrupt Register Byte 2

Bit	Symbol	Reset	Description
0	Endpoint 3 Out	0	Interrupt for endpoint 3 OUT buffer. Cleared by Read Last Transaction Status command.
1	Endpoint 3 In	0	Interrupt for endpoint 3 IN buffer. Cleared by Read Last Transaction Status command.
2	Endpoint 4 Out	0	Interrupt for endpoint 4 OUT buffer. Cleared by Read Last Transaction Status command.
3	Endpoint 4 In	0	Interrupt for endpoint 4 IN buffer. Cleared by Read Last Transaction Status command.
4	Endpoint 5 Out	0	Interrupt for endpoint 5 OUT buffer. Cleared by Read Last Transaction Status command.
5	Endpoint 5 In	0	Interrupt for endpoint 5 IN buffer. Cleared by Read Last Transaction Status command.
6	Endpoint 6 Out	0	Interrupt for endpoint 6OUT buffer. Cleared by Read Last Transaction Status command.
7	Endpoint 6 In	0	Interrupt for endpoint 6 IN buffer. Cleared by Read Last Transaction Status command.

Table 2.126 Interrupt Register Byte 3 (for Enhanced Mode)

Bit	Symbol	Reset	Description
0	Endpoint 7 Out	0	Interrupt for endpoint 7 OUT buffer. Cleared by Read Last Transaction Status command.
1	Endpoint 7 In	0	Interrupt for endpoint 7 IN buffer. Cleared by Read Last Transaction Status command.
7..2	Reserved	0	Reserved

Table 2.127 Interrupt Register Byte 4 (for Enhanced Mode)

2.11.4.2 *Select Endpoint*

Command : 0x00-0x05 (0x00-0x0F for Enhanced Mode)

Data : Optional Read 1 byte

Bit	Symbol	Reset	Description
0	Full/Empty	0	0: selected endpoint buffer is empty 1: selected endpoint buffer is full
1	Stall	0	0: selected endpoint is not stalled 1: selected endpoint is stalled
7..2	Reserved	0	Reserved

Table 2.128 Endpoint Status Register

2.11.4.3 *Read Last Transaction Status*

Command : 0x40-0x45 (0x40-0x4F for Enhanced Mode)

Data : Read 1 byte

Bit	Symbol	Reset	Description
0	Data Receive/Transmit Success	0	0: indicate USB data receive or transmit not OK 1: indicate USB data receive or transmit OK
4..1	Error Code	0	Refer to the error code Table 2.130
5	Setup Packet	0	0: indicates the packet is not a setup packet 1: indicates the last received packet has a SETUP token
6	Data 0/1 Packet	0	0: packet has a DATA0 token 1: packet has a DATA1 token
7	Previous Status not Read	0	0: previous transaction status was read 1: previous transaction status was not read

Table 2.129 Endpoint Last Transaction Status Register

Error Code	Result
0000	No error
0001	PID encoding error
0010	PID unknown
0011	Unexpected packet
0100	Token CRC error
0101	Data CRC error
0110	Time out error
0111	Reserved
1000	Unexpected EOP
1001	Packet NAKed
1010	Sent stall
1011	Buffer overflow
1101	Bit stuff error
1111	Wrong DATA PID

Table 2.130 Transaction error code

2.11.4.4 Read Endpoint Status

Command : 0x80-0x85 (0x80-0x8F for Enhanced Mode)

Data : Read 1 byte

Bit	Symbol	Reset	Description
1..0	Reserved	0	Reserved
2	Setup packet	0	0: indicates packet is not a setup packet 1: indicates last received packet has a SETUP token
4..3	Reserved	0	Reserved
5	Buffer 0 Full	0	0: buffer 0 is not filled up 1: buffer 0 is filled up
6	Buffer 1 Full	0	0: buffer 1 is not filled up 1: buffer 1 is filled up
7	Endpoint Stalled	0	0: endpoint is not stalled 1: endpoint is stalled

Table 2.131 Endpoint Buffer Status Register

2.11.4.5 Read Buffer

Command : 0xF0

Data : Read multiple bytes

The Read Buffer command is used to read the received packet from the selected endpoint OUT buffer.

The data in the endpoint buffer is organized as follows:

byte 0: length of payload packet, MSB(for default mode this byte is ignored)

byte 1: length of payload packet, LSB

byte 2: Payload packet byte 1

byte 3: Payload packet byte 2

…

byte n+1: Payload packet byte n (n = packet length)

2.11.4.6 Write Buffer

Command : 0xF0

Data : Write multiple bytes

The Write Buffer command is used to write a payload packet to the selected endpoint IN buffer.

The data must be organized in the same way as described in the Read Buffer command. For the default mode, byte 0 should always be set to 0x00.

2.11.4.7 Clear Buffer

Command : 0xF2

Data : None

Following a Read Buffer command, the Clear Buffer command should be issued after all data has been read out from the endpoint buffer. This is to free the buffer to receive next packet from the USB host.

2.11.4.8 Validate Buffer

Command : 0xFA

Data : None

Following a Write Buffer command, the Validate Buffer command should be issued after all data has been written to the endpoint buffer. This is to set the buffer full flag so that the packet can be sent to the USB host when the IN token arrives.

2.11.4.9 Set Endpoint Status

Command : 0x40-0x45 (0x40-0x4F for Enhanced Mode)

Data : Write 1 byte

Bit	Symbol	Reset	Description
0	Stall	0	0: Disable the endpoint STALL state. 1: Enable the endpoint STALL state. For EP0 OUT (control OUT endpoint) the STALL state will automatically be cleared by a received SETUP packet. When this bit is written to '0', the endpoint will reinitialise. Any data in the endpoint buffer will be flushed away, and the PID for the next packet will carry a DATA0 flag.
7..1	Reserved	0	Reserved

Table 2.132 Endpoint Control Register

2.11.4.10 Acknowledge Setup

Command : 0xF1

Data : None

When receiving a SETUP packet the USB Full Speed device controller will flush the IN buffer and disable the Validate Buffer and Clear Buffer commands for both Control IN and Control OUT endpoints. The MCU shall read and process the SETUP packet, and then issue the Acknowledge Setup command to re-enable the Validate Buffer and Clear Buffer commands. The Acknowledge Setup command must be sent to both Control IN and Control OUT endpoints.

2.11.5 General Commands

2.11.5.1 Read Current Frame Number

Command : 0xF5

Data : Read One or Two Bytes

Bit	Symbol	Reset	Description
7..0	Frame Number LSB	0	Frame number for last received SOF, byte 1 (least significant byte)

Table 2.133 Frame Number LSB Register

Bit	Symbol	Reset	Description
2..0	Frame Number MSB	0	Frame number for last received SOF, byte 2 (Most significant byte)
7..3	Reserved	0	Reserved

Table 2.134 Frame Number MSB Register

2.11.5.2 Send Resume

Command : 0xF6

Data : None

To perform remote-wakeup when suspended, the CPU needs to issue a Send Resume command. The USB Full Speed device controller will send an upstream resume signal for a period of 10 ms. If the clock is not running during suspend, the MCU needs to wakeup FT122USB Full Speed device controller by drive SUSPEND pin to LOW, followed by Send Resume command.

2.11.5.3 Set Buffer Interrupt Mode

Command : 0xEC (for Enhanced Mode)

Data : none

The read or write buffer commands can be interrupted, typically by a read interrupt register or read last transaction status command, and can be resumed without having to re-issue a read or write buffer command. When the default command set is in use, a read or write buffer command can be resumed after 2 bytes have been read with a read interrupt command. In this case the USB Full Speed device controller design is primed to resume a read or write buffer command if another command is not issued and a read or write occurs.

For the enhanced command set the read interrupt register command has been extended to read 4 bytes. The USB Full Speed device controller therefore needs to know whether to prime at 2 or 4 bytes. The Set Buffer Interrupt Mode command notifies the USB Full Speed device controller to prime after 4 bytes.

2.12 Pulse Width Modulation

The Pulse Width Modulation (PWM) module provides a number of PWM outputs individually controlled by the FT51A CPU.

The main purpose is to generate PWM signals which can be used to control motors, DC/DC converters, AC/DC supplies, etc.

The PWM block can generate pulse width modulated signals where parameters such as period and duty cycle are controlled by the CPU writing to memory mapped registers. All PWM outputs use a common pre-scaler block and 16-bit counter. The PWM module has eight 16-bit comparators to control up to eight toggles (four pulses where each period and duty cycle is individually set) of the generated signals. The additional clr signal sets output to the initial state. The module also consists of an output control block which enables PWM outputs (once, twice .... 255 times, forever) and generates an interrupt.

The registers associated with the Pulse Width Modulation (PWM) are outlined below.

I/O Address	Register Name	Description
0x80	PWM_CONTROL	PWM top level control register
0x81	PWM_INT_CTRL	PWM control register
0x82	PWM_PRESCALER	System clock pre-scaler register
0x83	PWM_CNT16_LSB	PWM counter LSB register
0x84	PWM_CNT16_MSB	PWM counter MSB register
0x85	PWM_CMP16_0_LSB	PWM comparator LSB register 0
0x86	PWM_CMP16_0_MSB	PWM comparator MSB register 0
0x87	PWM_CMP16_1_LSB	PWM comparator LSB register 1
0x88	PWM_CMP16_1_MSB	PWM comparator MSB register 1
0x89	PWM_CMP16_2_LSB	PWM comparator LSB register 2
0x8A	PWM_CMP16_2_MSB	PWM comparator MSB register 2
0x8B	PWM_CMP16_3_LSB	PWM comparator LSB register 3
0x8C	PWM_CMP16_3_MSB	PWM comparator MSB register 3
0x8D	PWM_CMP16_4_LSB	PWM comparator LSB register 4
0x8E	PWM_CMP16_4_MSB	PWM comparator MSB register 4
0x8F	PWM_CMP16_5_LSB	PWM comparator LSB register 5
0x90	PWM_CMP16_5_MSB	PWM comparator MSB register 5
0x91	PWM_CMP16_6_LSB	PWM comparator LSB register 6

0x92	PWM_CMP16_6_MSB	PWM comparator MSB register 6
0x93	PWM_CMP16_7_LSB	PWM comparator LSB register 7
0x94	PWM_CMP16_7_MSB	PWM comparator MSB register 7
0x95	PWM_OUT_TOGGLE_EN_0	PWM out toggle enable register 0
0x96	PWM_OUT_TOGGLE_EN_1	PWM out toggle enable register 1
0x97	PWM_OUT_TOGGLE_EN_2	PWM out toggle enable register 2
0x98	PWM_OUT_TOGGLE_EN_3	PWM out toggle enable register 3
0x99	PWM_OUT_TOGGLE_EN_4	PWM out toggle enable register 4
0x9A	PWM_OUT_TOGGLE_EN_5	PWM out toggle enable register 5
0x9B	PWM_OUT_TOGGLE_EN_6	PWM out toggle enable register 6
0x9C	PWM_OUT_TOGGLE_EN_7	PWM out toggle enable register 7
0x9D	PWM_OUT_CLR_EN	PWM out clear enable register
0x9E	PWM_CTRL_BL_CMP8	PWM control block register
0x9F	PWM_INIT	PWM initialization register

Table 2.135 PWM Register Addresses

Pulse Width Modulation (PWM) is the simplest form of duty cycle control. Consider the simple square wave in the figure below.

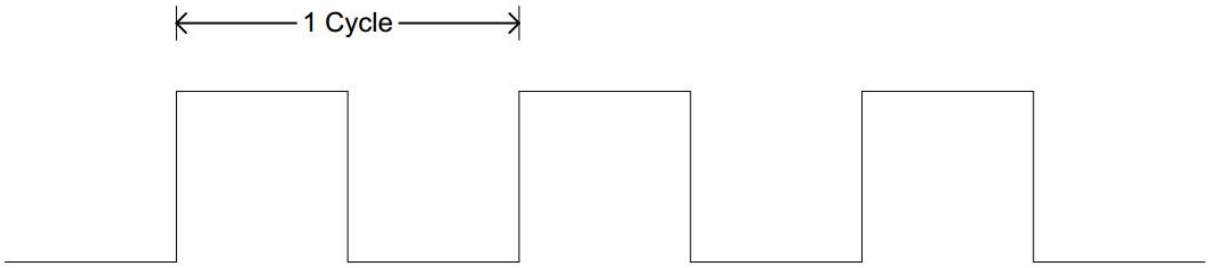


Figure 2.7 Square wave with 50 % duty cycle

In this example, the width of the high pulse is equal to the width of the low pulse. This waveform has a 50 % duty cycle. If the amplitude of this square wave is 5V, the RMS voltage can be calculated as:

VRMS = VPEAK * SQRT (Duty Cycle)

VRMS = 5V * SQRT (.50)

VRMS = 3.54V

If the duty cycle is reduced, then the RMS voltage reduces. This is illustrated in the following example which shows a waveform with a 20% duty cycle:

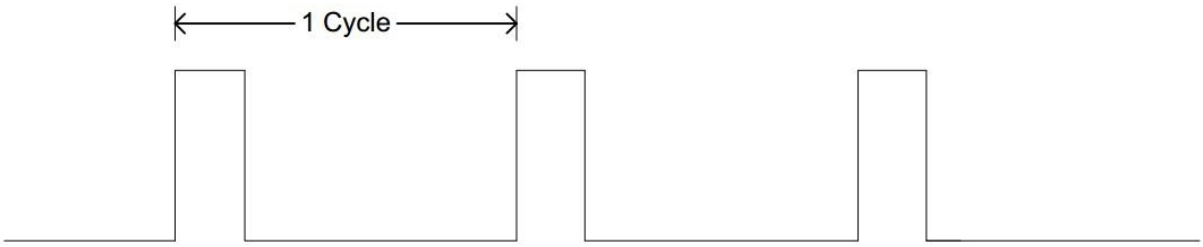


Figure 2.8 Square wave with 20 % duty cycle

With the same 5V peak amplitude as the 50% duty cycle waveform, the 20% duty cycle waveform has the following RMS voltage:

VRMS = 5V * SQRT (.20)

VRMS = 2.24V

By changing the duty cycle, the effective RMS is modified without changing the signal amplitude.

Why is this important?

By changing the amplitude and duty cycle of the signal, it is essentially generating an analogue signal from a digital source. PWM is a method that can be used to interface to analogue hardware using a digital source such as a microcontroller.

Real world applications of PWM include lamp brightness, electric motor control and servo control.

The FT51A has 8 independent PWM channels. The following describes all registers used to control these PWM channels.

2.12.1 PWM_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	`pwm_dev_en`	R/W	0	Enable PWM
0	`pwm_soft_reset`	W1T	0	Reset PWM

Table 2.136 PWM Control Register

The PWM Control register provides top-level enable and reset functions for the PWM module.

The PWM module is enabled by setting the `pwm_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `pwm_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.12.2 PWM_INT_CTRL

Bit Position	Bit Field Name	Type	Reset	Description
7..6	Reserved	RFU	0	Reserved
5	pwm_int	R/W	0	PWM interrupt
4	pwm_int_ien	R/W	0	PWM interrupt enable
3	pwm_busy	R	0	PWM busy
2	pwm_trigger_en_1	R/W	0	PWM trigger enable: 00 disabled, 01 positive edge, 01 negative edge 11 any edge
1	pwm_trigger_en_0	R/W	0	
0	pwm_en	R/W	0	PWM enable connected to Control block.

Table 2.137 PWM Ctrl 1 Register

This register allows enabling and detecting PWM interrupt, PWM busy, and setting up the trigger edge.

2.12.3 PWM_PRESCALER

Bit Position	Bit Field Name	Type	Reset	Description
7..0	prescaler	R/W	0	8-bit prescaler value

Table 2.138 PWM Prescaler Register

This is a programmable counter that reduces the frequency of the system clock to the desired frequency. The pre-scaler is shared by all 8 PWM channels.

2.12.4 PWM_CNT16_LSB

Bit Position	Bit Field Name	Type	Reset	Description
7..0	cnt16_lsb	R/W	0	LSB of a 16-bit counter value

Table 2.139 PWM Counter LSB Register

This is a LSB part of a programmable counter that determines the period of the PWM signal.
The input clock is from the pre-scaler block. The 16 bit counter is shared by all 8 PWM channels.

2.12.5 PWM_CNT16_MSB

Bit Position	Bit Field Name	Type	Reset	Description
7..0	cnt16_msb	R/W	0	MSB of a 16-bit counter value

Table 2.140 PWM Counter MSB Register

This is a MSB part of a programmable counter that determines the period of the PWM signal. The input clock is from the pre-scaler block. The 16 bit counter is shared by all 8 PWM channels.

2.12.6 PWM_CMP16_0_LSB - PWM_CMP16_7_LSB

Bit Position	Bit Field Name	Type	Reset	Description
7..0	cmp16_lsb	R/W	0	LSB of a 16-bit comparator value

Table 2.141 PWM Comparator LSB Register

Up to 8 comparators can be used per PWM channel. The number of comparators assigned to a PWM channel determines the toggle events (up to 8), which give up to 4 data pulses.
Simple duty cycle based pulse width modulation can be programmed with only two comparators. There are a total of 8 comparators in the PWM module. Here only the first comparator is shown, as the remaining seven are exact copies of it.

2.12.7 PWM_CMP16_0_MSB - PWM_CMP16_7_MSB

Bit Position	Bit Field Name	Type	Reset	Description
7..0	cmp16_msb	R/W	0	MSB of a 16-bit comparator value

Table 2.142 PWM Comparator MSB Register

This is a MSB part of a programmable comparator that determines the toggle events. The 16-bit counter is shared across all 8 PWM channels. Here only the first comparator is shown, as the remaining seven are exact copies of it.

2.12.8 PWM_OUT_TOGGLE_EN_0 - PWM_OUT_TOGGLE_EN_7

Bit Position	Bit Field Name	Type	Reset	Description
7..0	toggle_en	R/W	0	PWM Out toggle enable bits for each of the 8 PWM channels.

Table 2.143 PWM Toggle Enable Register

There are eight Toggle Enable registers. Each is used to select which combination of PWM comparators to use.

2.12.9 PWM_OUT_CLR_EN

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`clr_en`	R/W	0	PWM Out Clear Enable bits for each of the 8 PWM channels.

Table 2.144 PWM Out Clear Enable Register

2.12.10 PWM_CTRL_BL_CMP8

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`ctrl_bl_cmp8`	R/W	0	Control block CMP8 value. 0 continuous 1 one shot 2-255 repeat specified number of times

Table 2.145 PWM Control Block Register

This controls the number of times to repeat the PWM waveform. The control block is shared across all 8 PWM channels.

2.12.11 PWM_INIT

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`init`	R/W	0	PWM Initialization register bits for each of the 8 PWM channels.

Table 2.146 PWM Initialisation Register

2.12.12 Use Cases

The following section describes an example of how to generate a 4-pulse PWM waveform using the FT51A, which toggles at the following counter states: 7, 8, 12, 14, 15, 16, 19, and 22.

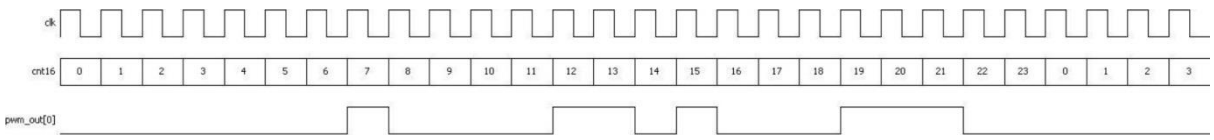


Figure 2.9 Pulse Waveform generated by 8 comparators

In this example there are eight toggle events and all eight comparators are used. In this example, the 16 bit counter is programmed to count 24 states and then restart, and four pulses are generated.

Comparator#	Programmed Toggle Value	Pulse Width (clock cycles)
0	7	
1	8	1
2	12	
3	14	2
4	15	
5	16	1
6	19	
7	22	3

Table 2.147 Programming 8 FT51A comparators to generate above waveform

For a simple duty cycle PWM, where only a single pulse is required, only 2 comparators would be necessary and only a single pulse is generated. For example, to generate a 50% duty cycle waveform for the clock/counter combination, comparators 0 & 1 could be programmed as follows:

Comparator#	Programmed Toggle Value	Pulse Width (clock cycles)
0	2	
1	14	12

Table 2.148 Programming 2 FT51A comparators for 50 % duty cycle

In this example, there are 24 clocks per cycle and the PWM output changes state (toggles) every 12 clocks (12/24). This produces a 50 % duty cycle. By programming different toggling values into the FT51A comparators, a wide range of duty cycles can be generated.

First parameter to decide on is the frequency of the PWM Clock. The highest possible PWM frequency is 48MHz (no pre-scaling).

To derive the PWM Clock, set up the FT51A System Clock pre-scaler.
Pre-scale by 1 to get the PWM Clock equalling to a half of the FT51A System Clock.
Pre-scale by 255 to get the PWM Clock equalling to the FT51A System Clock divided by 256, etc.
Therefore to get the required PWM Clock frequency use the following formula and code:

PWM Clock = FT51A System Clock / (prescaler+1)

```
//Set PWM prescaler to divide the FT51A System Clock by 2
WRITE_IO_REG(0x82,  0x01); //PWM_PRESCALER
```

To set the PWM sequence time, specify the number of PWM pulses to count from 0 to 65535:

*PWM sequence period = (total PWM pulses) * (prescaler+1)/ FT51A System Clock*,
where 0<= total PWM pulses <=65535.

And set the PWM counter value with the MSB and LSB registers as follows:

```
//Set PWM counter
WRITE_IO_REG(0x84,  0x12); //PWM_CNT16_MSB
WRITE_IO_REG(0x83,  0x34); //PWM_CNT16_LSB
```

For clarification, follow this example.

Specify the PWM Pre-scaler to divide the FT51A System Clock to get the 1MHz PWM clock.

Next, set the PWM counter, for example, to the maximum value of 0xFFFF. Hence, the time it takes to reach value 0xFFFF is equal to 1us * 0xFFFF = 65535us. In brief, the duration (width) of the PWM pulse is influenced by two parameters: PWM Clock and PWM Counter, giving two extreme situations (min and max):

1 - (assuming the 48MHz FT51A System Clock) Set the PWM Clock to its maximum value of 48MHz and increment PWM counter only by one to get the PWM pulse of roughly 20ns.

PWM Clock = 48MHz;

PWM Counter = 1;

PWM Pulse Duration = 20ns

2a - (assuming the 48MHz FT51A Clock) Set the PWM Clock to its minimum value of 187.5kHz (assuming the 48MHz FT51A Clock) and increment PWM counter by 65535 to get the PWM pulse of roughly 349.52ms.

PWM Clock = 187.5kHz;

PWM Counter = 65535;

PWM Pulse Duration = 349.52ms

2b - (assuming the 6MHz FT51A Clock) Set the PWM Clock to its minimum value of 23.437kHz and increment PWM counter by 65535 to get the PWM pulse of roughly 2.8s.

PWM Clock = 23.437kHz;

PWM Counter = 65535;

PWM Pulse Duration = 2.8s

Hence the ranges become:

System Clock [MHz]	PWM Clock		PWM Pulse Duration	
	Min [Hz]	Max [MHz]	Min [ns]	Max [s]
48	187500	48	20.8ns	0.34952s
24	93750	24	41.6ns	0.69904s
12	46875	12	83.3ns	1.39808s
6	23437	6	166.6ns	2.79622s

Table 2.149. PWM Ranges

To specify the width of a specific PWM pulse, set the comparators at times required and enable them by the toggle register. The following sets the width from 0 to 0xAAFF.

Note: The toggle register number relates to the PWM number assigned via IOMux.

```
// Assign PWM output pad
WRITE_IO_REG (0x41, 12); // IOMUX_OUTPUT_PAD_SEL, pad number AIO_12
WRITE_IO_REG (0x42, 38);// IOMUX_OUTPUT_SIG_SEL, PWM_OUT_03
WRITE_IO_REG(0x16, 1); // DIGITAL_CONTROL_AIO_12

//Set PWM Comparator 0
WRITE_IO_REG(0x86,  0x00); //PWM_CMP16_0_MSB
WRITE_IO_REG(0x85,  0x00); //PWM_CMP16_0_LSB

//Set PWM Comparator 1
WRITE_IO_REG(0x88, 0xAA); //PWM_CMP16_1_MSB
WRITE_IO_REG(0x87, 0xFF); //PWM_CMP16_1_LSB

// Set toggle enables for the two comparators above
WRITE_IO_REG(0x95, 0x03); //PWM_OUT_TOGGLE_EN_0
```

PWM initialization is performed as follows:

```
// Top level PWM soft reset
WRITE_IO_REG (0x80, 0x01); // PWM_CONTROL, PWM_SOFT_RESET

// Top level PWM enable
WRITE_IO_REG (0x80, 0x02); // PWM_CONTROL, PWM_DEV_EN

// Set initial output state
WRITE_IO_REG (0x9F, 0x00); // PWM_INIT

// Number of repetitions
WRITE_IO_REG (0x9E, 0x00); // PWM_CTRL_BL_CMP8

// Enable output
WRITE_IO_REG (0x81, 0x01); // PWM_INT_CTRL, PWM_EN
```

2.13 Timers

The FT51A is equipped with two types of timers: standard 8051 timers and FTDI timers. Standard timers are Timer 0, 1 and 2 which are accessed through SFRs, while the FTDI timers comprise 16-bit Timers A, B, C and D and are controlled by means of I/O registers. .

The four FTDI timers can be clocked off main clock or a common 16-bit pre-scaler. This can be selected for each timer individually. These timers can be started, stopped and cleared or initialised separately. Current values of all four "user timers" can be read from registers (one at a time - common register, multiplexed access). All timers can count up or down and signal an interrupt when they roll over. Each of them can be configured to be in one-shot or in continuous mode. They are initialised from a common register set so only one may be initialised at a time (multiplexed access).

The FTDI watchdog timer is clocked off the main clock. The watchdog is initialised with a 5-bit register. The value of this register points to a single bit of a 32-bit counter that will be set. A timer then decrements and signals an interrupt when it rolls over. Once started and initialised the watchdog cannot be stopped. It can be cleared by writing into a register.

The pre-scaler block is a 16-bit timer/counter. It can be cleared / initialised by writing into a register in the same way as other timers. However if one of the FTDI timers has already started using the pre-scaler it cannot be cleared and the command is ignored. The pre-scaler automatically stops after it is cleared. It also starts automatically when any of the FTDI timers using it starts.

All timers (the 4 FTDI timers, pre-scaler and watchdog) are initialised with a start value if counting up or 0 if counting down.

Below is a list of registers associated with the FTDI Timers:

I/O Address	Register Name	Description
0x70	TIMER_CONTROL	Timers top level control register
0x71	TIMER_CONTROL_1	Timer start/stop control register
0x72	TIMER_CONTROL_2	Timer pre-scaler and watchdog
0x73	TIMER_CONTROL_3	Timer settings
0x74	TIMER_CONTROL_4	Clear timer and pre-scaler
0x75	TIMER_INT	Timer interrupts
0x76	TIMER_SELECT	Timer select/read
0x77	TIMER_WDG	Watchdog start value
0x78	TIMER_WRITE_LS	Timer start value
0x79	TIMER_WRITE_MS	Timer start value
0x7A	TIMER_PRESC_LS	Timer pre-scale value
0x7B	TIMER_PRESC_MS	Timer pre-scale value
0x7C	TIMER_READ_LS	Timer read value
0x7D	TIMER_READ_MS	Timer read value

Table 2.150 Timer Register Addresses

2.13.1 TIMER_CONTROL

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	timer_dev_en	R/W	0	Write a 1 to enable TIMER
0	timer_soft_reset	W1T	0	Write a 1 to reset TIMER

Table 2.151 Timer Control Register

The Timer Control register provides top-level enable and reset functions for the timer module.

The timer module is enabled by setting the `timer_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `timer_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.13.2 TIMER_CONTROL_1

Bit Position	Bit Field Name	Type	Reset	Description
7	stop_D	W1T	0	Stop the timer D
6	stop_C	W1T	0	Stop the timer C
5	stop_B	W1T	0	Stop the timer B
4	stop_A	W1T	0	Stop the timer A
3	start_D	W1T	0	Start the timer D
2	start_C	W1T	0	Start the timer C
1	start_B	W1T	0	Start the timer B
0	start_A	W1T	0	Start the timer A

Table 2.152 Timer Control 1 Register

2.13.3 TIMER_CONTROL_2

Bit Position	Bit Field Name	Type	Reset	Description
7..4	prescaler_en	R/W	0	Enable pre-scaler bits for timers A..D.
3	wdg_int_ien	R/W	0	Watchdog interrupt enable.
2	wdg_int	R/W	0	Watchdog interrupt.
1	clear_wdg	W1T	0	Clear watchdog.
0	start_wdg	W1T	0	Start watchdog.

Table 2.153 Timer Control 2 Register

2.13.4 TIMER_CONTROL_3

Bit Position	Bit Field Name	Type	Reset	Description
7..4	direction	R/W	0	Write '1' for up counter for timer A (bit 4) to D (bit 7). Default: Down counter.
3..0	mode	R/W	0	Write '1' to enable one-shot from timer A (bit 0) to D (bit 3). Default: Continuous mode.

Table 2.154 Timer Control 3 Register

2.13.5 TIMER_CONTROL_4

Bit Position	Bit Field Name	Type	Reset	Description
4	presc_clear	W1T	0	Pre-scaler clear.
3	clear_D	W1T	0	Clear Timer D.
2	clear_C	W1T	0	Clear Timer C.
1	clear_B	W1T	0	Clear Timer B.
0	clear_A	W1T	0	Clear Timer A.

Table 2.155 Timer Control 3 Register

2.13.6 TIMER_INT

Bit Position	Bit Field Name	Type	Reset	Description
7	timer_int_D_ien	R/W	0	Timer D interrupt enable.
6	timer_int_D	R/W	0	The timer D interrupt.
5	timer_int_C_ien	R/W	0	Timer C interrupt enable.
4	timer_int_C	R/W	0	The timer C interrupt.
3	timer_int_B_ien	R/W	0	Timer B interrupt enable.
2	timer_int_B	R/W	0	The timer B interrupt.
1	timer_int_A_ien	R/W	0	Timer A interrupt enable.
0	timer_int_A	R/W	0	The timer A interrupt.

Table 2.156 Timer Control 3 Register

2.13.7 TIMER_SELECT

Bit Position	Bit Field Name	Type	Reset	Description
3..2	timer_read_sel	R/W	0	Timer read select bits. - 00 for timer A - 01 for timer B - 10 for timer C - 11 for timer D
1..0	timer_sel	R/W	0	Timer select bits. - 00 for timer A - 01 for timer B - 10 for timer C - 11 for timer D

Table 2.157 Timer Control 3 Register

2.13.8 TIMER_WDG

Bit Position	Bit Field Name	Type	Reset	Description
4..0	timer_wdg_write	R/W	0	Watchdog bit position to initialise

Table 2.158 Timer Watchdog Register

2.13.9 TIMER_WRITE_LS

Bit Position	Bit Field Name	Type	Reset	Description
7..0	timer_write_7_0	R/W	0	Bits 7 to 0 of the timer start value.

Table 2.159 Timer Write LSB Register

2.13.10 TIMER_WRITE_MS

Bit Position	Bit Field Name	Type	Reset	Description
7..0	timer_write_15_8	R/W	0	Bits 15 to 8 of the timer start value.

Table 2.160 Timer Write MSB Register

2.13.11 TIMER_PRESC_LS

Bit Position	Bit Field Name	Type	Reset	Description
7..0	timer_presc_7_0	R/W	0	Bits 7 to 0 of the pre-scale value.

Table 2.161 Timer Prescaler MSB Register

2.13.12 TIMER_PRESC_MS

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`timer_presc_15_8`	R/W	0	Bits 15 to 8 of the pre-scale value.

Table 2.162 Timer Prescaler MSB Register

2.13.13 TIMER_READ_LS

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`timer_read_7_0`	R/W	0	Read bits 7 to 0 of the timer.

Table 2.163 Timer Read MSB Register

2.13.14 TIMER_READ_MS

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`timer_read_15_8`	R/W	0	Read bits 15 to 8 of the timer.

Table 2.164 Timer Read MSB Register

2.13.15 Use Cases

The table below presents normal operation of FTDI timers along with pre-scaler and watchdog.

FTDI timers	Prescaler	Watchdog
Select the timer to initialise by writing `timer_sel` field in the `TIMER_SELECT` register. Write initial/final value into the `TIMER_WRITE_MS` and `TIMER_WRITE_LS` registers.	Write an initial value into the `TIMER_PRESC_LS` and `TIMER_PRESC_MS` registers.	Write an initial value into the `TIMER_WDG` register to initialise one of the 32 bits of the timer.
Write into `TIMER_CTRL_3` register `direction` field to select up/down counting.	N/A	N/A
Write into `TIMER_CTRL_3` register `mode` field to select mode.	N/A	N/A
Write bit for selected timer to `TIMER_CTRL_4` register to initialise the timer.	Write `clear` in `TIMER_CTRL_4` register to initialise the prescaler (if possible)	Write `clear_wdg` in `TIMER_CTRL_2` register to clear watchdog.
Write the start bit for selected timer in `TIMER_CTRL_1` register to start the timer.	Write `prescaler_en` bit in `TIMER_CTRL_2` register to enable prescaler and it will automatically start when the timer/timers using it starts.	Write `start_wdg` bit in `TIMER_CTRL_2` register to start watchdog.
Select the timer to read from by writing the `timer_read_sel` field in the `TIMER_SELECT` register. Current value can be read from the `TIMER_READ_LS` and `TIMER_READ_MS` registers.	N/A	N/A
Write the stop bit for the selected timer in the `TIMER_CTRL_1` register to stop the timer.	N/A	N/A

Table 2.165 Timers Normal Operation

2.13.15.1 Timers

Below is a comparison of steps necessary to setup a standard 8051 Timer 0 versus the FTDI Timer A.

Standard 8051 Timer 0

```
//Set the Timer 0 prescaler to 0 (0 - divide-by-12, 1 - divide-by-4)
//Note: CKCON bit 3 relates to Timer 0, bit 4 Timer 1
CKCON &= 0xF7;

// Set timer control mode, either timer mode, counter mode or UART mode.
TMOD &= 0xF0;
TMOD |= 0x01; //select timer mode 1, 16-bit timer THx and TLx are cascaded.

//Preload high and low timer value
TH0 = 0xAA;
TL0 = 0xAA;

//Enable interrupts
IE &= 0x7D;

//Timer control
TCON &= 0xCF; //reset
TCON |= 0x10; //run
while((TCON & 0x20) == 0); //loop until timer overflows
TCON &= 0xCF; // stop
```

FTDI Timer A

```
//Precautionary Top Level Soft Reset
WRITE_IO_REG(0x70, 0x01); //TIMER_CONTROL, TIMER_SOFT_RESET

//Initialize timer by clearing it and the prescaler
WRITE_IO_REG(0x74, 0x11); //TIMER_CONTROL_4, CLEAR_A | PRESC_CLEAR

//Top Level Enable
WRITE_IO_REG (0x70, 0x02); //TIMER_CONTROL, TIMER_DEV_EN

//Select timer
WRITE_IO_REG (0x76, 0x00); //TIMER_SELECT, TIMER A

//Write initial value
WRITE_IO_REG (0x79, 0xAA); //TIMER_WRITE_MS
WRITE_IO_REG (0x78, 0xAA); //TIMER_WRITE_LS

//Enable top-level interrupts
WRITE_IO_REG (0x05, 0x04); //TOP_INT1_0, TIMER_IRQ

//Enable timer interrupt
WRITE_IO_REG (0x75, 0x02); //TIMER_INT, TIMER_INT_A_IEN

//Set the prescaler
WRITE_IO_REG (0x72, 0x10); //TIMER_CONTROL_2, PRESCALER_EN_0

//Start timer A
WRITE_IO_REG (0x71, 0x01); //TIMER_CONTROL_1, START_A
```

FTDI Timers A, B, C and D require 1 cycle per instruction, whereas standard timers 0, 1 or 2 require either 12 (default) or 4 cycles per instruction. Additionally, the clock for Timers A, B, C and D can also be pre-scaled, which changes the available time range. The maximum pre-scale value is 0xFFFF.

The delay range (in seconds) achievable with one 16-bit timer is shown in the table below. (Due to the aforementioned 16-bit constraint, Timers 0 and 1 can only be in mode 1; timers A, B, C and D can be in any mode.)

CLK	**NUMBER OF CYCLES PER INSTRUCTION**		
	12	**4**	**1**
48000000	0.00000025 - 0.01638375	8.33333E-08 - 0.00546125	2.08333E-08 - 0.001365313
24000000	0.0000005 - 0.0327675	1.66667E-07 - 0.0109225	4.16667E-08 - 0.002730625
12000000	0.000001 - 0.065535	3.33333E-07 - 0.021845	8.33333E-08 - 0.00546125
6000000	0.000002 - 0.13107	6.66667E-07 - 0.04369	1.66667E-07 - 0.0109225

Table 2.166 Available timer ranges (in seconds)

Yielding

MAX = 0.13107 s
MIN = 2.08333E-08 s
MAX-MIN = 0.131069979 s

The most appropriate unit to accurately represent the delay is the nanosecond. The type has to be able to contain value 131070000 (MAX-MIN), which requires four bytes, as number 131070000 = 0x07CFF830. Therefore, the delay parameter in an example function should be of uint32_t type.

With the uint32_t type, the timer range becomes:

CLK [Hz]	Timer range	
	Timer 0 - 2	**Timer A – D**
48000000	84 ns - 16383750 ns	21 ns - 4294967295 ns (= 4.3 s) (90 s if not limited by the uint32_t type) calculated by inverse of maximally pre-scaled clock times max number of timer increments = 1/(48MHz/65536))*65535
24000000	167 ns - 32767500 ns	42 ns - 4.3 s (179 s if not limited by the uint32_t type)
12000000	334 ns - 65535000 ns	84 ns - 4.3 s (358 s if not limited by the uint32_t type)
6000000	667 ns - 131070000 ns	167 ns - 4.3 s (720 s if not limited by the uint32_t type)

Figure 2.10 Timer range for uint32_t timer

2.13.15.2 Watchdog

Software must feed the watchdog within a set period (as determined by the `timer_wdg_write` bit of the `TIMER_WDG` register) to verify proper software execution. If a hardware or software failure prevents feeding the watchdog within the minimum timeout period, the FT51A reset is forced. Once the system has undergone the reset, it is possible to check with the `wdg_int` bit of the `TIMER_CONTROL_2` register whether the reset was caused by a watchdog overflow or normal reset. The watchdog timer is disabled only by power-on reset and during power-down. Once enabled, it cannot be disabled. It is enabled by the `start_wdg` bit of the `TIMER_CONTROL_2` register.

Below are example functions that could be incorporated into user's code.

```
/**
    @brief      Watchdog Enable
    @details    A function to start the watchdog.
    @param[in]  start_value  Watchdog start value (only bits from 0 to 4 used).
    @return     void
**/
void watchdog_enable(uint8_t  start_value)
{

    // Initialise watchdog timer with the set Start Value.
        WRITE_IO_REG(TIMER_WDG,
                     start_value & 0x1F);

    // Watchdog interrupt enable.
        WRITE_IO_REG(TIMER_CONTROL_2,
                     WDG_INT_IEN);

    // Enable Timer.
        WRITE_IO_REG(TIMER_CONTROL,
                     TIMER_DEV_EN);

    // Clear the watchdog before the start is issued.
        WRITE_IO_REG(TIMER_CONTROL_2,
```

```
                        CLEAR_WDG);

    // Start the watchdog.
        WRITE_IO_REG(TIMER_CONTROL_2,
                        MSTART_WDG);
}

/**
    @brief      Watchdog Feed
    @details    The function feeds the watchdog in order to prevent it from
                resetting the system.
    @return     void
**/

void watchdog_feed()
{
    // Clear the watchdog. **/
        WRITE_IO_REG(TIMER_CONTROL_2,
                        CLEAR_WDG);
}


/**
    @enum RESET_REASON
    @brief Explains the cause of the most recent reset.
**/
typedef enum
{
    // Power supply was removed or interrupted.
    NORMAL_RESET,
    // Watchdog was not fed in time.
    WATCHDOG_RESET
}
RESET_REASON;

/**
    @brief      Reset cause.
    @details    A function to check what caused the reset:
                i.e. whether the system started after a normal reset
                or a watchdog reset.
    @param[in]  reason  Explains most-recent reset.
    @return     0 - success, 1 - failure
**/
int reset_cause(RESET_REASON *reason)
{
    uint8_t data;

    if (reason == NULL)
        return 1;

    *reason = NORMAL_RESET; // Assume normal reset until found otherwise.

    // Check if a watchdog overflow occurred.
    READ_IO_REG(TIMER_CONTROL_2,
                1,
                &data);

    // Ensure that only the watchdog interrupt bit gets checked.
    data &= WDG_INT;

    if (data == WDG_INT)
    {
        *reason = WATCHDOG_RESET;
    }
```

```
    return 0;
}

/**
    @brief      Main
    @details    Main program entry point.
    @return     No! Returning from main() is undefined!
**/
int main()
{
    RESET_REASON    reset_reason = WATCHDOG_RESET;
    int             i;

    /**
        Once the cause of the reset is known, the behaviour of the app can
        be changed accordingly.
    **/
    reset_cause(&reset_reason);

    if (reset_reason != WATCHDOG_RESET)
    {
        // Write code to do something if watchdog had not caused the reset
       watchdog_enable(0x1A); // Enable watchdog

       // Feed the watchdog for some time and go to an infinite loop
       for(i=0; i<13; i++)
        {
             watchdog_feed();
        }
    }
    else
    {
        // Write code to do something in the event watchdog had caused the reset.
    }

    //This infinite loop will cause the watchdog to time out and reset the system.
    while(1);

}
```

2.14 DMA

The DMA relieves the CPU from performing a data/memory transfer. Below is a list of registers associated with FTDI DMA feature. There are 4 DMA engines available.

I/O Address	Register Name	Description
0xB0	DMA_CONTROL_1	Enable/reset for DMA 1
0xB1	DMA_ENABLE_1	IO Peripheral DMA enable/reset register
0xB2	DMA_IRQ_ENA_1	IO Peripheral DMA interrupts enable register
0xB3	DMA_IRQ_1	IO Peripheral DMA Interrupts register
0xB4	DMA_SRC_MEM_ADDR_L_1	IO Peripheral DMA source memory address LSB register
0xB5	DMA_SRC_MEM_ADDR_U_1	IO Peripheral DMA source memory address MSB register
0xB6	DMA_DEST_MEM_ADDR_L_1	IO Peripheral DMA destination memory address LSB register
0xB7	DMA_DEST_MEM_ADDR_U_1	IO Peripheral DMA destination memory address MSB register
0xB8	DMA_IO_ADDR_L_1	IO Peripheral DMA IO Address LSB Register
0xB9	DMA_IO_ADDR_U_1	IO Peripheral DMA IO Address MSB Register
0xBA	DMA_TRANS_CNT_L_1	IO Peripheral DMA Transfer Byte Count LSB Register
0xBB	DMA_TRANS_CNT_U_1	IO Peripheral DMA Transfer Byte Count MSB Register
0xBC	DMA_CURR_CNT_L_1	IO Peripheral DMA Current Transfer Byte Count LSB Register
0xBD	DMA_CURR_CNT_U_1	IO Peripheral DMA Current Transfer Byte Count MSB Register
0xBE	DMA_FIFO_DATA_1	IO Peripheral DMA FIFO DATA
0xBF	DMA_AFULL_TRIGGER_1	IO Peripheral DMA Almost Full Trigger Value
0xC0	DMA_CONTROL_2	Enable/reset DMA 2

0xC1	DMA_ENABLE_2	IO Peripheral DMA enable/reset register
0xC2	DMA_IRQ_ENA_2	IO Peripheral DMA interrupts enable register
0xC3	DMA_IRQ_2	IO Peripheral DMA Interrupts register
0xC4	DMA_SRC_MEM_ADDR_L_2	IO Peripheral DMA source memory address LSB register
0xC5	DMA_SRC_MEM_ADDR_U_2	IO Peripheral DMA source memory address MSB register
0xC6	DMA_DEST_MEM_ADDR_L_2	IO Peripheral DMA destination memory address LSB register
0xC7	DMA_DEST_MEM_ADDR_U_2	IO Peripheral DMA destination memory address MSB register
0xC8	DMA_IO_ADDR_L_2	IO Peripheral DMA IO Address LSB Register
0xC9	DMA_IO_ADDR_U_2	IO Peripheral DMA IO Address MSB Register
0xCA	DMA_TRANS_CNT_L_2	IO Peripheral DMA Transfer Byte Count LSB Register
0xCB	DMA_TRANS_CNT_U_2	IO Peripheral DMA Transfer Byte Count MSB Register
0xCC	DMA_CURR_CNT_L_2	IO Peripheral DMA Current Transfer Byte Count LSB Register
0xCD	DMA_CURR_CNT_U_2	IO Peripheral DMA Current Transfer Byte Count MSB Register
0xCE	DMA_FIFO_DATA_2	IO Peripheral DMA FIFO DATA
0xCF	DMA_AFULL_TRIGGER_2	IO Peripheral DMA Almost Full Trigger Value
0xD0	DMA_CONTROL_3	Enable/reset DMA 3
0xD1	DMA_ENABLE_3	IO Peripheral DMA enable/reset register
0xD2	DMA_IRQ_ENA_3	IO Peripheral DMA interrupts enable register
0xD3	DMA_IRQ_3	IO Peripheral DMA Interrupts register
0xD4	DMA_SRC_MEM_ADDR_L_3	IO Peripheral DMA source memory

		address LSB register
0xD5	DMA_SRC_MEM_ADDR_U_3	IO Peripheral DMA source memory address MSB register
0xD6	DMA_DEST_MEM_ADDR_L_3	IO Peripheral DMA destination memory address LSB register
0xD7	DMA_DEST_MEM_ADDR_U_3	IO Peripheral DMA destination memory address MSB register
0xD8	DMA_IO_ADDR_L_3	IO Peripheral DMA IO Address LSB Register
0xD9	DMA_IO_ADDR_U_3	IO Peripheral DMA IO Address MSB Register
0xDA	DMA_TRANS_CNT_L_3	IO Peripheral DMA Transfer Byte Count LSB Register
0xDB	DMA_TRANS_CNT_U_3	IO Peripheral DMA Transfer Byte Count MSB Register
0xDC	DMA_CURR_CNT_L_3	IO Peripheral DMA Current Transfer Byte Count LSB Register
0xDD	DMA_CURR_CNT_U_3	IO Peripheral DMA Current Transfer Byte Count MSB Register
0xDE	DMA_FIFO_DATA_3	IO Peripheral DMA FIFO DATA
0xDF	DMA_AFULL_TRIGGER_3	IO Peripheral DMA Almost Full Trigger Value
0xE0	DMA_CONTROL_4	Enable/reset DMA 4
0xE1	DMA_ENABLE_4	IO Peripheral DMA enable/reset register
0xE2	DMA_IRQ_ENA_4	IO Peripheral DMA interrupts enable register
0xE3	DMA_IRQ_4	IO Peripheral DMA Interrupts register
0xE4	DMA_SRC_MEM_ADDR_L_4	IO Peripheral DMA source memory address LSB register
0xE5	DMA_SRC_MEM_ADDR_U_4	IO Peripheral DMA source memory address MSB register
0xE6	DMA_DEST_MEM_ADDR_L_4	IO Peripheral DMA destination memory address LSB register
0xE7	DMA_DEST_MEM_ADDR_U_4	IO Peripheral DMA destination memory address MSB register

0xE8	DMA_IO_ADDR_L_4	IO Peripheral DMA IO Address LSB Register
0xE9	DMA_IO_ADDR_U_4	IO Peripheral DMA IO Address MSB Register
0xEA	DMA_TRANS_CNT_L_4	IO Peripheral DMA Transfer Byte Count LSB Register
0xEB	DMA_TRANS_CNT_U_4	IO Peripheral DMA Transfer Byte Count MSB Register
0xEC	DMA_CURR_CNT_L_4	IO Peripheral DMA Current Transfer Byte Count LSB Register
0xED	DMA_CURR_CNT_U_4	IO Peripheral DMA Current Transfer Byte Count MSB Register
0xEE	DMA_FIFO_DATA_4	IO Peripheral DMA FIFO DATA
0xEF	DMA_AFULL_TRIGGER_4	IO Peripheral DMA Almost Full Trigger Value

Table 2.167 DMA Register Addresses

2.14.1 DMA_CONTROL_x

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	`dma_control_x_dev_en`	R/W	0	Write a 1 to enable DMA engine x
0	`dma_control_x_soft_reset`	W1T	0	Write a 1 to reset DMA engine x

Table 2.168 DMA Control Register

The DMA Control register provides top-level enable and reset functions for the given DMA engine.

The DMA engine is enabled by setting the `dma_control_x_dev_en` bit to 1. Clearing this bit will disable the module.

To reset the module, a 1 is written to the `dma_control_x_soft_reset` bit. This is cleared when the reset is performed and will therefore always read as '0'.

2.14.2 DMA_ENABLE_x

Bit Position	Bit Field Name	Type	Reset	Description
7..2	Reserved	RFU	0	Reserved
1	dma_stop	R/W	0	IO Peripheral DMA Stop
0	dma_start	W1T	0	IO Peripheral DMA Start

Table 2.169 DMA Enable/Reset Register

2.14.3 DMA_IRQ_ENA_x

Bit Position	Bit Field Name	Type	Reset	Description
7	Reserved	RFU	0	Reserved
6..5	dma_fifo_size	R/W	0	IO Peripheral DMA Set FIFO Size: 00 == 64, 01 == 128, 10 == 256, 11 == 512 Bytes
4..3	dma_mode	R/W	0	IO Peripheral DMA Mode: 00 == PUSH (push data from memory to IO), 01 == PULL (pull data from IO to memory), 10 == MEM_COPY (copy data from one memory location to another memory location), 11 == FIFO
2	dma_fifo_davail_ien	R/W	0	IO Peripheral DMA FIFO Data Available IRQ Enable
1	dma_fifo_full_ien	R/W	0	IO Peripheral DMA FIFO Full IRQ Enable
0	dma_done_ien	W1T	0	IO Peripheral DMA Done IRQ Enable

Table 2.170 DMA Interrupts Enable Register

2.14.4 DMA_IRQ_x

Bit Position	Bit Field Name	Type	Reset	Description
7..3	Reserved	RFU	0	Reserved
2	`dma_fifo_davail_int`	R/W	0	IO Peripheral DMA FIFO Data Available IRQ
1	`dma_fifo_full_int`	R/W	0	IO Peripheral DMA FIFO Full IRQ
0	`dma_done_int`	W1T	0	IO Peripheral DMA Done IRQ indicating an interrupt for DMA in either Push, Pull or FIFO mode.

Table 2.171 DMA Interrupts Register

2.14.5 DMA_SRC_MEM_ADDR_L_x

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`dma_src_mem_addr_l`	R/W	0	IO Peripheral DMA Source Memory Address Register [7..0]

Table 2.172 IO Peripheral DMA Source Memory Address LSB Register

2.14.6 DMA_SRC_MEM_ADDR_U_x

Bit Position	Bit Field Name	Type	Reset	Description
7	`dma_src_mem_addr_id`	R/W	0	IO Peripheral DMA Source Memory Address Increment/decrement: 0 == inc mem addr, 1 == dec mem addr
6..0	`dma_src_mem_addr_u`	R/W	0	IO Peripheral DMA Source Memory Address Register [14..8]

Table 2.173 IO Peripheral DMA Source Memory Address MSB Register

2.14.7 DMA_DEST_MEM_ADDR_L_x

Bit Position	Bit Field Name	Type	Reset	Description
7..0	`dma_dest_mem_addr_l`	R/W	0	IO Peripheral DMA Destination Memory Address Register [7..0]

Table 2.174 IO Peripheral DMA Destination Memory Address LSB Register

2.14.8 DMA_DEST_MEM_ADDR_U_x

Bit Position	Bit Field Name	Type	Reset	Description
7	dma_dest_mem_addr_id	R/W	0	IO Peripheral DMA Destination Memory Address Increment/decrement: 0 == inc mem addr, 1 == dec mem addr
6..0	dma_dest_mem_addr_u	R/W	0	IO Peripheral DMA Destination Memory Address Register [14..8]

Table 2.175 IO Peripheral DMA Destination Memory Address MSB Register

2.14.9 DMA_IO_ADDR_L_x

Bit Position	Bit Field Name	Type	Reset	Description
7..0	dmaio_io_addr_l	R/W	0	IO Peripheral DMA IO Address Register [7..0]

Table 2.176 IO Peripheral DMA IO Address LSB Register

2.14.10 DMA_IO_ADDR_U_x

Bit Position	Bit Field Name	Type	Reset	Description
7..4	dma_fctrl	R/W	0	Bits 4 to 7 determine which flow control signals to use: 0 UART, 1 SPI Master, 2 SPI Slave, 3 Parallel245, 4 DMA_FIFO, 5 unused, 6 unused, 7 unused
3..1	Reserved	RFU	0	Reserved
0	dmaio_io_addr_u	R/W	0	IO Peripheral DMA IO Address Register [8]

Table 2.177 IO Peripheral DMA IO Address MSB Register

2.14.11 DMA_TRANS_CNT_L_x

Bit Position	Bit Field Name	Type	Reset	Description
7..0	dma_trans_cnt_l	R/W	0	IO Peripheral DMA Transfer Byte Count Register [7..0]

Table 2.178 IO Peripheral DMA Transfer Byte Count LSB Register

2.14.12 DMA_TRANS_CNT_U_x

Bit Position	Bit Field Name	Type	Reset	Description
5..0	dma_trans_cnt_u	R/W	0	IO Peripheral DMA Transfer Byte Count Register [13..8]

Table 2.179 IO Peripheral DMA Transfer Byte Count MSB Register

2.14.13 DMA_CURR_CNT_L_x

Bit Position	Bit Field Name	Type	Reset	Description
7..0	dma_curr_cnt_l	R	0	IO Peripheral DMA Current Transfer Byte Count Register [7..0]

Table 2.180 IO Peripheral DMA Current Transfer Byte Count LSB Register

2.14.14 DMA_CURR_CNT_U_x

Bit Position	Bit Field Name	Type	Reset	Description
5..0	dma_curr_cnt_u	R	0	IO Peripheral DMA Current Transfer Byte Count Register [13..8]

Table 2.181 IO Peripheral DMA Current Transfer Byte Count MSB Register

2.14.15 DMA_FIFO_DATA_x

Bit Position	Bit Field Name	Type	Reset	Description
7..0	dma_fifo_data	R/W	0	IO Peripheral DMA FIFO DATA

Table 2.182 IO Peripheral DMA FIFO DATA Register

2.14.16 DMA_AFULL_TRIGGER_x

Bit Position	Bit Field Name	Type	Reset	Description
5..0	dma_afull_trigger	R/W	0	IO Peripheral DMA Almost Full Trigger Value

Table 2.183 IO Peripheral DMA Almost Full Trigger Value

2.14.17 Use Cases

DMA can be configured in three modes: Push, Pull and FIFO. The first two perform a one-shot transfer, while the last one is for continuous reception of data.

Push is used to transfer data from memory to I/O register, while Pull and FIFO are used to transfer data in the opposite direction. Below is a list of steps necessary to setup DMA engine 1 in the Push mode.

1. Enable global and 'External 0' interrupts via SFR IE register.

```
IE |= (IE_GLOBAL | IE_EXTERNAL_0);
```

2. Enable interrupts on the DMA engine.

```
WRITE_IO_REG(DMA_IRQ_ENA_1, DMA0_IRQ_IEN);
```

3. Enable individual DMA engine.

```
WRITE_IO_REG(DMA_CONTROL_1, DMA_CONTROL_0_DEV_EN);
```

4. Enable interrupts for the Push mode.

```
WRITE_IO_REG(DMA_IRQ_ENA_1, (DMA_IRQ_DONE_IEN | (0 << MASK_DMA_MODE)));
```

5. Specify transfer size.

```
WRITE_IO_REG(DMA_TRANS_CNT_U_1, transfer_size >>8 );
WRITE_IO_REG(DMA_TRANS_CNT_L_1, transfer_size);
```

6. Define source.

```
WRITE_IO_REG(DMA_SRC_MEM_ADDR_U_1, source >>8 );
WRITE_IO_REG(DMA_SRC_MEM_ADDR_L_1, source);
```

7. Define destination along with specifying flow control.

```
destination |= ((uint16_t)(flow_control) << 8);
WRITE_IO_REG(DMA_IO_ADDR_U_1, destination >> 8);
WRITE_IO_REG(DMA_IO_ADDR_L_1, destination);
```

8. Start the DMA engine.

```
WRITE_IO_REG(DMA_ENABLE_1, DMA_START)
```

The DMA relieves the CPU from performing the transfer. In the case of Push mode, the only time when the CPU gets interrupted is when the transfer has completed. In the case of Pull/FIFO mode, the DMA may also interrupt when its buffer has been filled to a specified level, as set by the DMA_AFULL_TRIGGER_1 register.

Preferably, one would want to react on any received interrupt, for example, by calling some call-back function. The implementation of such scenario is presented below.

```
// Check what engine interrupted

void INT0_ISR(void) __interrupt (0)
{
    uint8_t              events = 0;

    READ_IO_REG(TOP_INT0, events);

    events &= (DMA0_IRQ | DMA1_IRQ | DMA2_IRQ | DMA3_IRQ);

    if (events && DMA0_IRQ != NULL)
    {
        user_callback();
    }

    // Clear the event flags
    WRITE_IO_REG(TOP_INT0, events);
}
```

It is important to note that while called back, the function must clear the DMA engine-specific interrupts that triggered the interrupt: `dma_fifo_davail_int`, `dma_fifo_full_int` or `dma_done_int`.

3 Application Guide

FTDI provide sample code and source code libraries as part of the FT51A Software Development Kit to enable fast project development. The FT51A SDK Installer (see AN_352_FT51A_Installation_Guide) will install the source code for the libraries described in this section. It will also install sample code which will demonstrate the use of these source code libraries.

The source code libraries provide functions that can be used to facilitate various features reducing the amount of additional code being required. They are intended to implement hardware features only. The algorithms and methods used are recommended and tested by FTDI.

Several of the source code libraries implement a framework that can be used or adapted in an application.

3.1 Libraries

The source code libraries are provided to show how to implement and manage features of the FT51A. They can be modified as required.

Source code is normally provided with one or two functions in each file. The filename is the name of the function prepended with the name of the library. The reason for this is that the SDCC linker will optimize-out entire files which are not called but will not normally optimize-out unused functions within files. This reduces code size in an application without requiring manual examination of calling graphs.

The intention is that only functions (and hence files) that are required for an application are included in the project.

The USB device library, Interrupts library and DMA library provide frameworks for an application to use to exploit the features in hardware.

The function descriptions in this section are only brief overviews of the functions. Please also refer to the library header files which contain Dioxygen compatible comments for each exposed function. The Doxygen comments contain descriptions, parameters and return values of functions. Hence, there are no comments describing functions in the source files.

The filename containing the function is noted along with any other files that the function requires.

3.1.1 Configuration Library

This library contains functions or macros to properly initialise the FT51A, query or set the system clock frequency (6, 12, 24 or 48 MHz) and determine the IC package.

Source Folder: general_configuration

3.1.1.1 device_initialise()

File: ft51_general_config_device_initialise.c

This is used to initialise the registers in the FT51's 8051 core registers and enable FT51A peripheral register access.

This should be called once at the very start of an application.

3.1.1.2 get_ic_package()

File: ft51_general_config_get_ic_package.c

This will return an enumerated value indicating the package type and hence pin count of the device.

3.1.1.3 get_system_clock()

File: ft51_general_config_get_system_clock.c

The function returns an enumerated value allowing the current frequency of the FT51A clock to be determined.

3.1.1.4 set_system_clock()

File: ft51_general_config_set_system_clock.c

The system clock frequency can be adjusted by changing the SYSTEM_CLOCK_DIVIDER register. This function abstracts the process using one of the enumerated clock frequency values.

3.1.2 USB Library

The USB library is designed to manage the interface from the FT51A device to the host. The library invokes call-back functions in the application code when certain events happen, such as the host issuing a standard request, or resetting the bus.

There are 2 control endpoints, IN and OUT; in default mode there are 2 other pairs of endpoints; in extended mode there are 6 other pairs of endpoints. The control endpoints are always enabled; however, other endpoints must be enabled by code when required.

Call-backs are nominated using the USB_initialise() function call. This function will also setup an interrupt handler for USB interrupts and create the control endpoints that are always required in a USB device.

The USB_process() function is called during the application's main loop to perform required USB operations via the call-back functions. This allows the call-backs to be run by the application and not at interrupt level.

Non-control endpoints must be created by a call to USB_create_endpoint() before being used. An endpoint is described in this library by both the endpoint number and its direction. Once created, an endpoint can be removed with USB_free_endpoint().

The application will use the USB_transfer() function to send information to the host via IN endpoints. The same function called on an OUT endpoint will receive any data pending from the host.

There is a memory structure associated with an endpoint and code to process interrupts. Therefore the number of endpoints to be used can be defined in the application as a build definition. When building the USB library files add the following macro to the SDCC compiler command line (where x is the number of endpoints):

```
-D USB_MAX_ENDPOINT_COUNT=x
```

Additional parameters checking for endpoint functions can be enabled by setting the following macro:

```
-D USB_ENDPOINT_CHECKS=1
```

This is normally disabled to save code space.

Source Folder: usb

3.1.2.1 USB_initialise

File: ft51_usb_initialise.c

Requires: ft51_usb_remote_wakeup_feature.c, ft51_usb_endpoint.c, ft51_interrupts.c

This is called to initialise the USB library with call-back functions, configure the hub device and set the initial size for the control endpoints. It will also enable and configure the USB peripheral device to a state where it can receive and respond to SETUP packets from the host.

The call-back functions passed to this function will allow the USB device to be enumerated by the host and respond to all required SETUP requests from the host device driver.

The function will initialise memory structures for control IN and OUT endpoints and register an interrupt handler for peripheral interrupts from the USB device. The hub will be configured as requested and the USB Full Speed device controller device will be enabled ready to be discovered by the host.

This should only be called once in the application.

3.1.2.2 USB_req_set_address

File: ft51_usb_address.c

Requires: ft51_usb_transfer.c, ft51_usb_endpoint_features.c

Sets the USB address of the device during enumeration. This is called from the call-back function which deals with standard requests when a SET_ADDRESS request is received.

3.1.2.3 USB_req_set_configuration

File: ft51_usb_configuration.c

Requires: ft51_usb_transfer.c

The last step of enumeration is the SET_CONFIGURATION request. This is a standard request and this function is called from the standard request call-back function.

3.1.2.4 USB_req_get_configuration

File: ft51_usb_configuration.c

This function will return the current configuration for the GET_CONFIGURATION request to the host. This is a standard request and this function is called from the standard request call-back function.

3.1.2.5 USB_clear_endpoint

File: ft51_usb_endpoint_features.c

This is called by the standard CLEAR_FEATURE request when an endpoint stall clear is required. When this request is received by the application then it should call this handler to clear the selected endpoint. The application standard request handler must then return a zero length acknowledge packet to the host.

This may also be called by an application to manage an endpoint.

3.1.2.6 USB_stall_endpoint

File: ft51_usb_endpoint_features.c

This can be called by the standard SET_FEATURE request when an endpoint stall is required. When this request is received by the application then it should call this handler to stall the selected endpoint. The application standard request handler must then return a zero length acknowledge packet to the host.

This may also be called by an application to manage an endpoint when a stall is required. For instance when request for an unimplemented SETUP transaction is received.

3.1.2.7 USB_free_endpoint

File: ft51_usb_endpoint.c

This function can be used to disable an endpoint.

3.1.2.8 USB_create_endpoint

File: ft51_usb_endpoint.c

This function can be used to enable an endpoint. It will return a status indicating if the endpoint was created.

The USB_initialise function will call this function for the control endpoints. It is not necessary to call this function from elsewhere to create control endpoints.

3.1.2.9 USB_ep_buffer_full

File: ft51_usb_ep_buffer_full.c

OUT endpoints can be polled to see if there is data requiring action. This function is used to check an OUT endpoint for data received from the host. A non-zero value indicates that there is data received.

For IN endpoints, the return value indicates that data is waiting to be transmitted to the host at the next IN request.

3.1.2.10 USB_get_ep_stalled

File: ft51_usb_endpoint_features.c

This will return a non-zero value if the endpoint is stalled.

3.1.2.11 USB_get_state

File: ft51_usb_state.c

This function will return the current state of the USB device. This is defined in Section 9.1 of the USB Specifications.

3.1.2.12 USB_set_state

File: ft51_usb_state.c

Calling this function will set the current state of the USB device. This is defined in Section 9.1 of the USB Specifications.

3.1.2.13 USB_transfer

File: ft51_usb_transfer.c

Performs a data transfer from the USB device to or from the USB host. This will use an endpoint handle to select the endpoint used for the transfer. A zero length packet for a control endpoint acknowledge can be sent via this function.

3.1.2.14 USB_process

File: ft51_usb_process.c

Requires: ft51_usb_endpoint_features.c, ft51_usb_remote_wakeup_feature.c

When the application is running it should call this function to process events received by the USB device. Processing will result in the activation of call-back functions if required.

3.1.2.15 USB_isr

File: ft51_usb_isr.c

This is the routine that handles interrupts from the USB Full Speed device controller device on the FT51A. It will check and clear interrupt status bits and call handler routines accordingly.

When USB_initialise is called, it will register this function as the handler for USB interrupts.

3.1.2.16 USB_finalise

File: ft51_usb_finalise.c

Will release all resources held by the USB device and disable all endpoints.

3.1.3 DMA Library

The UART, 245 FIFO and SPI interfaces can send/receive data directly to/from RAM without the intervention of the 8051 core using the DMA controller. It can also send data around RAM without using a peripheral interface.

This library provides functions to configure these fixed-size transfers and also continuous reads, where each byte arriving at an interface is copied into a First-In-First-Out buffer in data memory.

After initialisation, one of the 4 DMA engines can be acquired with DMA_acquire(). The function returns one of the 4 engines. If there are no engines free then it will return an error.

The engine must then be configured with DMA_configure() to be push, pull or FIFO. The configure function also specifies either a memory or I/O address to use as source or sink locations and the length of the transfer.

Once configured, the DMA_enable() command will set a DMA engine to run to complete the transfer. As this does not block until completion the DMA_wait_on_complete() call is used to wait until the transfer is finished.

When the DMA engine is no longer required then DMA_release() will allow the DMA engine to be used again.

Source Folder: dma

3.1.3.1 DMA_initialise

File: ft51_dma_initialise.c

Requires: ft51_interrupts.c

Initialise the DMA controller. This must be called before any other DMA operation.

3.1.3.2 DMA_acquire

File: ft51_dma_acquire.c

Request a DMA engine to use. This will call calloc() to allocate memory from the heap.

3.1.3.3 DMA_configure

File: ft51_dma_configure.c

Configure a DMA engine source and destination addresses, transfer length, mode and flow control method. The source and destination addresses may be registers or memory addresses.

The mode is one of push or pull. FIFO mode is configured with the DMA_configure_fifo() function.

3.1.3.4 DMA_enable

File: ft51_dma_enable.c

Start a DMA engine to run until the transfer is complete. FIFO DMAs will run indefinitely.

3.1.3.5 DMA_disable

File: ft51_dma_disable.c

Stop a DMA engine when a transfer is incomplete.

3.1.3.6 DMA_wait_on_complete

File: ft51_dma_wait_on_complete.c

Wait until a DMA engine has completed its transfer.

3.1.3.7 DMA_configure_fifo

File: ft51_dma_configure_fifo.c

The FIFO mode is always paired with a pull DMA engine. Flow control can be enabled for certain peripherals such as the UART to activate and deactivate flow control depending on the amount of data stored in a FIFO buffer.

3.1.3.8 DMA_purge_fifo

File: ft51_dma_purge_fifo.c

Remove all data in the FIFO buffer. This is currently not implemented.

3.1.3.9 DMA_get_fifo_count

File: ft51_dma_get_fifo_count.c

Get the number of bytes in the FIFO buffer.

3.1.3.10 DMA_get_fifo_data

File: ft51_dma_get_fifo_data.c

Get a number of bytes from the FIFO buffer.

3.1.3.11 DMA_switch_fifo

File: ft51_dma_switch _fifo.c

Change the destination buffer of a FIFO.

3.1.3.12 DMA_is_complete

File: ft51_dma_is_complete.c

Checks whether a DMA is complete.

3.1.3.13 DMA_release

File: ft51_dma_release.c

Requires: ft51_dma_reset.c

There are no more uses for this DMA engine and it can be disabled and released. This function calls free() to remove allocated memory from the heap.

3.1.3.14 DMA_reset

File: ft51_dma_reset.c

Reset one of the DMA engines.

3.1.4 UART Library

The peripheral UART is capable of sending and receiving data at up to 3 Mbaud. The library contains functions to claim the UART, configure it (number of data bits, stop bits, baud rate etc.) and send or receive individual messages.

The default configuration for the UART at power on is 9600 baud, 8 data bits, 1 stop bit, no parity, and no flow control.

Source Folder: uart

3.1.4.1 UART_initialise

File: ft51_uart_initialise.c

Initialise the UART. This must be called before any other UART operation.

3.1.4.2 UART_set_baud_rate

File: ft51_uart_set_baud_rate.c

Requires: ft51_general_config_get_system_clock.c

This function will calculate the baud rate divisor values for the UART_BAUD_0, UART_BAUD_1 and UART_BAUD_2, taking into account the speed of the system clock.

It requires that the Configuration library is included for the get_system_clock() function.

The calculation utilises 32 bit arithmetic and will therefore include SDCC system libraries for 32 bit calculations. These can add a significant amount of code to a program. If space is an issue then the baud rate divisor can be pre-calculated and programmed directly into the divisor registers without making the calculation.

3.1.4.3 UART_set_data_bits

File: ft51_uart_set_data_bits.c

Set the number of data bits for the UART to 7 or 8.

3.1.4.4 UART_set_flow_control

File: ft51_uart_set_flow_control.c

Configure the UART to enable or disable flow control. If it is enabled then it can be set to CTS/RTS or DTR/DSR modes.

3.1.4.5 UART_set_parity

File: ft51_uart_set_parity.c

Enable or disable parity bit generation and checking on the UART.

3.1.4.6 UART_set_stop_bits

File: ft51_uart_set_stop_bits.c

Set the number of stop bits in a UART character to 1 or 2.

3.1.4.7 UART_read

File: ft51_uart_read.c

Read a number of bytes from the UART interface. This will use interrupts to manage the transaction and will block until the required amount of data has been received.

3.1.4.8 UART_write

File: ft51_uart_write.c

Write a number of bytes from a buffer to the UART interface. This uses interrupts to manage the transaction. It will block until the data has been transmitted.

3.1.5 SPI Master Library

This library has functions to send and receive messages using the SPI master interface. It includes variants which are interrupt-driven (per byte) or DMA-driven (per block). Interrupt and DMA driven options cannot be used at the same time within a project.

To use the SPI Master in interrupt mode, call the function SPIM_initialise_ints() to enable the hardware interface before configuring it to match the communication method of a SPI Slave with SPIM_configure_slave(). Then calling SPIM_transceive_ints() is used to exchange data with the slave device.

To use the SPI Master in DMA mode, call the function SPIM_initialise_DMA() to enable the hardware interface before configuring it to match the communication method of a SPI Slave with SPIM_configure_slave(). Then calling SPIM_transceive_DMA() is used to exchange data with the slave device.

Source Folder: spi_master

3.1.5.1 SPIM_configure_slave

File: ft51_spim_configure_slave.c

This function is used to configure the SPI Master to connect to a slave device with the required communication properties. The master's bit order, clock speed, clock mode and slave select idle and polarity matched to the slave device.

3.1.5.2 SPIM_initialise_ints

File: ft51_spim_transceive_ints.c

This is called first to initialise the SPI Master to use interrupts.

3.1.5.3 SPIM_transceive_ints

File: ft51_spim_transceive_ints.c

A transaction sending data to and receiving data from the SPI Slave will be started. The call will block until all data is sent (and received). Interrupts are used to wait for completion of the transfers.

3.1.5.4 SPIM_initialise_DMA

File: ft51_spim_transceive_dma.c

Requires: ft51_dma_initialise.c, ft51_dma_acquire.c, ft51_dma_release.c

This is called first to initialise the SPI Master to use DMA.

It requires that the DMA library is included.

3.1.5.5 SPIM_transceive_DMA

File: ft51_spim_transceive_dma.c

Requires: ft51_dma_configure.c, ft51_dma_enable.c, ft51_dma_wait_on_complete.c

A transaction sending data to and receiving data from the SPI Slave will be started. The call will block until all data is sent (and received). DMAs are used to wait for completion of the transfers.

It is necessary to include the DMA library.

3.1.6 I^2C Master Library

The I^2C Master library provides a simple read and write interface to the I^2C Master peripheral. There is a function to abstract setting the timer period register.

It is possible to use interrupt methods to perform both reads and write. This is not implemented in this library.

Source Folder: i2c

3.1.6.1 I2C_master_initialise

File: ft51_i2c_master.c

This function will reset and initialise the I^2C Master peripheral. By default a value of 0x40 is programmed into the `I2CMTP` register resulting in a frequency of 36.9 kHz. A dummy slave address (0x22) is programmed into the `I2CMSA` register.

3.1.6.2 I2C_master_set_timer

File: ft51_i2c_master.c

This function will change the setting of the `I2CMTP` register to change the frequency and the mode of the I^2C Master. High-speed mode can be selected along with the timer counter value.

3.1.6.3 I2C_master_get_status

File: ft51_i2c_master.c

This function returns the value of the `I2CMSR` register.

3.1.6.4 I2C_master_write

File: ft51_i2c_master.c

This will perform an I^2C write to a specified slave address. The first byte sent is designated to be a command byte and is passed separately from the optional data packet in the function parameters.

3.1.6.5 I2C_master_read

File: ft51_i2c_master.c

This will perform an I^2C read from a specified slave address. The first byte sent is designated to be a command byte and is passed separately from the optional data packet in the function parameters. The command byte is written before the I^2C bus direction is changed to read from the slave address.

3.1.7 I^2C Slave Library

The I^2C Slave library provides functions to receive data from and transmit data to an I^2C Master.

It is possible to use interrupt methods to perform both reads and write. This is not implemented in this library.

Source Folder: i2c

3.1.7.1 I2C_slave_initialise

File: ft51_i2c_slave.c

This function will reset and initialise the I^2C Slave peripheral. The requested slave address is passed as a parameter and programmed into the `I2CSOA` register.

3.1.7.2 I2C_slave_get_status

File: ft51_i2c_ slave.c

This function returns the value of the `I2CSSR` register.

3.1.7.3 I2C_slave_write

File: ft51_i2c_ slave.c

This will respond to read operations from an I^2C Master.

3.1.7.4 I2C_slave_read

File: ft51_i2c_ slave.c

This will perform an I^2C read from a specified slave address. The first byte sent is designated to be a command byte and is passed separately from the optional data packet in the function parameters. The command byte is written before the I^2C bus direction is changed to read from the slave address.

3.1.8 AIO Library

The analogue library can sample an analogue voltage. It is a simple interface to the ADC peripheral on the device.

There are 16 analogue pads which can be used for this, AIO_0 to AIO_15. These are at fixed pins on the device and cannot be routed with the IOMUX Library.

Source Folder: aio

3.1.8.1 AIO_intialise

File: ft51_aio_initialise.c

Set up the AIO Cell Controller.

3.1.8.2 AIO_read

File: ft51_aio_read.c

Perform an ADC operation on one analogue pad and wait for the result.

The ADC function can capture a signal in the range of 0 to 3.3 V and express it as an 8 bit integer (0 to 255).

3.1.9 IOMUX Library

Each pin on the device can be configured to act as the input or output pins from almost any peripheral. There are exclusions due to differences between analogue and digital signals.

Source Folder: iomux

3.1.9.1 IOMUX_initialise

File: ft51_iomux_initialise.c

Enable the IOMUX hardware to allow mappings of external pins to signals on internal peripherals. This must be called before any other IOMUX accesses.

3.1.9.2 IOMUX_INPUT

This is a macro to connect a pad to a signal to receive data for a peripheral.

3.1.9.3 IOMUX_OUTPUT

This is a macro to connect a signal to a pad to transmit data from a peripheral.

3.1.9.4 IOMUX_set_pad_config

File: ft51_iomux_set_pad_config.c

Permits the drive current, trigger levels, slew rates and pull parameters of a pad to be configured.

3.1.9.5 IOMUX_get_pad_config

File: ft51_iomux_get_pad_config.c

Retrieves the settings for the drive current, trigger levels, slew rates and pull parameters of a pad.

3.1.10 Watchdog Library

The watchdog timer library will allow a watchdog timer to be started with a configurable period. When the watchdog expires it will reset the device. To prevent the watchdog from expiring it needs regular feeding.

When the device is powered up it is possible to see if the watchdog was the cause of the reset.

Source Folder: watchdog

3.1.10.1 watchdog_enable

File: ft51_watchdog_enable.c

Setup the watchdog timer with a start value of 0 to 31. The start value is used to set a single bit in a 32 bit timer register. The timer counts down from this value and will reset the device when it reaches zero.

3.1.10.2 watchdog_feed

File: ft51_watchdog_feed.c

Feed the watchdog and therefore restore the start value to the watchdog timer.

3.1.10.3 reset_cause

File: ft51_watchdog_reset_cause.c

Get the reason for the last reset of the device. This can be a normal reset or a watchdog reset.

3.1.11 DFU Library

The DFU library provides functions for handling the Device Firmware Update class. The class is described in the following document:

http://www.usb.org/developers/docs/devclass_docs/DFU_1.1.pdf

It will manage a detach timer for transitioning to the DFU mode from Run Time mode and control the state machine when updating the firmware via download requests.

The firmware may still be required to reset the device after the firmware is downloaded when a USB reset is seen on the bus. The device cannot do a bus detach-attach sequence so the *bitWillDetach* in the bmAttributes will need to be set to 0 (for no).

It supports downloading of firmware only (*bitCanDnload*) but does not support manifest checking (*bitManifestationTolerant)*.

This library is in a single file

Source Folder: dfu

File: ft51_usb_dfu.c

Requires: ft51_usb_transfer.c, ft51_usb_endpoint_features.c

3.1.11.1 dfu_is_runtime

Returns non-zero if the current state of the DFU machine is runtime, i.e. DFU are not active.

3.1.11.2 dfu_class_req_detach

Handler for the DFU_DETACH class request. This will change the state from appIDLE to dfuDETACH and allow entering DFU mode on a USB reset. Starts a detach timer which will return to normal runtime mode (appIDLE) if it expires before a USB reset.

3.1.11.3 dfu_timer

Decrements the detach timer. If it expires then return to the appIDLE state.

3.1.11.4 dfu_class_req_download

Handles DFU_DNLOAD requests and programs the MTP memory with data received.

3.1.11.5 dfu_class_req_getstatus

Returns the current DFU state machine enumeration, and increments the state from dfuMANIFEST-SYNC to dfuMANIFEST-WAIT-RESET or dfuDNLOAD-SYNC to dfuDNLOAD-IDLE.

Returns current error status if set.

3.1.11.6 dfu_class_req_clrstatus

Clears a dfuERROR state back to dfuIDLE.

3.1.11.7 dfu_class_req_abort

Cancels the current DFU operation and returns back to dfuIDLE.

3.1.11.8 dfu_class_req_getstate

Return the current DFU state machine enumeration.

3.1.11.9 dfu_reset

Resets the DFU state machines for dfuDETACH to appIDLE for a cancelled detach, dfuMANIFEST-WAIT-RESET to appIDLE for a successful firmware download. If it is called at dfuDNLOAD-IDLE then this will flag an error as there is an incomplete firmware download.

3.1.12 LCD Library

The LCD library provides functions and methods for using an ST7036 LCD display and compatibles. It requires the I^2C Master library to provide the communication between the FT51A and the LCD display.

This library is in a single file

Source Folder: lcd

File: ft51_lcd_st7036.c

Requires: ft51_i2c_master.c

3.1.12.1 lcd_initialise

Send commands to the LCD to setup the display preferences.

It will enable a 2 line display with single height characters; set the bias and contrast; cursor off and blink off; clear the display and set the entry mode.

3.1.12.2 lcd_clear

Send the clear command to remove text from the display.

3.1.12.3 lcd_home

Move the cursor to the top left of the display.

3.1.12.4 lcd_position

Set the position of the cursor to the desired location on the display.

3.1.12.5 lcd_display

Writes a string to the display from the current cursor position.

3.1.13 TMC Library

The TMC library provides functions for handling the Test and Measurement class. The class is described in the following document (Test & Measurement Class section):

http://www.usb.org/developers/docs/devclass_docs/

The library will receive two types of packets from the host: a Message Out, which is a command; or a Request Message In, which is a request to return data to the host. After the latter packet, the host will do a Message In operation to receive a packet from the TMC device.

Each packet type from the host will execute a callback function. The callback functions are in the application and handle decoding the command and formatting the response.

The Message In is always immediately preceded by a Request Message In.

This library is in a single file.

Source Folder: tmc

File: ft51_usb_tmc.c

Requires: ft51_usb_transfer.c, ft51_usb_endpoint_features.c

3.1.13.1 tmc_initialise

Initialise the library and setup callback functions.

3.1.13.2 tmc_class_req_capabilities

Returns a response to a USB class request for the TMC capabilities (GET_CAPABILITIES).

3.1.13.3 tmc_class_init_clear

Performs an initiate clear to clear any pending or unprocessed TMC messages or responses (INITIATE_CLEAR).

3.1.13.4 tmc_class_check_clear

Returns the status of a previously sent INITIATE_CLEAR (CHECK_CLEAR_STATUS).

3.1.13.5 tmc_class_init_abort_bulk_out

Aborts a Bulk OUT transfer. This may be a Message Out or a Request Message In (INITIATE_ABORT_BULK_OUT).

3.1.13.6 tmc_class_check_abort_bulk_out

Returns the status of a previously sent INITIATE_ABORT_BULK_OUT (CHECK_ABORT_BULK_OUT_STATUS).

3.1.13.7 tmc_class_init_abort_bulk_in

Aborts a Bulk IN transfer. This may be a Message In transfer (INITIATE_ABORT_BULK_IN).

3.1.13.8 tmc_class_check_abort_bulk_in

Returns the status of a previously sent INITIATE_ABORT_BULK_IN (CHECK_ABORT_BULK_IN_STATUS).

3.1.13.9 tmc_process

Process Message Out, Request Message In and Message In packets from the host.

Callbacks in the application are used to start processing commands in a Message Out, and to format responses in a Request Message In.

The formatted response is sent after a Request Message In and must complete before returning.

This should be called periodically to handle the packets sent over the BULK endpoints of the TMC device.

3.2 USB Applications

An application which implements a USB device with the FTDI USB Library on the FT51A must include the following parts of the FT51A libraries:

- Include the ft51_interrupt.c file from the top-level of the library. This is required to implement the interrupt handler for the USB device.
- Add the general_config_device_initialise.c file from the Configuration Library.
- Add the following files from the USB Library:
  - usb_address.c
  - usb_configuration.c
  - usb_endpoint_features.c
  - usb_endpoint.c
  - usb_initialise.c
  - usb_isr.c
  - usb_process.c
  - usb_state.c
  - usb_transfer.c

The application must perform the following operations:

- Call device_initialise() from the Configuration Library to initialise the FT51A device.
- Initialise the device, configuration and string descriptors.
- Call USB_initialise() function to start the USB device function
  - Provide callback functions for standard SETUP requests from the host.
- Call the USB_process() function periodically.

If vendor or class SETUP requests are required by the host driver then additional handlers must be implemented for these.

The examples in this section are given in such a way that the descriptors defined in section 3.2.2 can be used in the call-back code in section 3.2.2.

3.2.1 Initialising USB Device

The USB Library provides a USB_initialise() function which will setup the USB device for use by an application. The call-backs are nominated, interrupt handler setup and control endpoints created.

A setup routine for a typical USB device may look like this:

```
void usb_setup(void)
{
    USB_ctx usb_ctx;

    memset(&usb_ctx, 0, sizeof(usb_ctx));

    usb_ctx.standard_req = standard_req_cb;        // required
    usb_ctx.class_req = class_req_cb;              // optional
    usb_ctx.vendor_req = vendor_req_cb;            // optional
    usb_ctx.suspend = suspend_cb;                  // optional
    usb_ctx.resume = resume_cb;                    // optional
    usb_ctx.reset = reset_cb;                      // optional
    usb_ctx.lpm = NULL;

    // Hub disabled
    usb_ctx.hub_en.mask.hub_enable = 0;
    usb_ctx.hub_en.mask.hub_CompdDev = 1;
    usb_ctx.hub_en.mask.hub_RmtWkUpEn = 0;

    // Initialise the USB device with a control endpoint size
    // of 8 bytes. This must match the size for bMaxPacketSize0
    // defined in the device descriptor.
    usb_ctx.ep_size = USB_EP_SIZE_8;
```

```
    // Perform USB Device initialisation.
    USB_initialise(&usb_ctx);
}
```

The USB_ctx structure passes the required and optional values to the initialization routine. The only call-back required is the standard_req callback. This handles standard requests.

The ep_size must also be specifically set to match the control endpoint bMaxPacketSize0 in the device descriptor returned to the host. The default value is 8 bytes that correlate with a value 0 in the USB_ENDPOINT_SIZE enumeration in ft51_usb.h.

When using the hub functionality, the hub must be configured to use the "compound device" unless the USB device application can be disabled.

3.2.2 Descriptors

The descriptors for the USB device are used in response to standard requests. There are typedefs in ft51_usb.h for the common standard descriptors. The typedefs can be used to initialise descriptors or byte arrays can be used instead.

It is useful to store descriptors in __code space as having them as automatic variables will mean that they are copied to the __data or __xdata areas. This will use memory resources. Typically the descriptors will not be changed by an application.

3.2.2.1 Device Descriptors

The device descriptor uses USB_device_descriptor typedef from ft51_usb.h. It provides a structure that can be initialised with values for the device descriptor.

```
__code USB_device_descriptor device_descriptor =
{
        .bLength = 0x12,
        .bDescriptorType = 0x01,
        .bcdUSB = USB_BCD_VERSION_2_0,
        .bDeviceClass = USB_CLASS_DEVICE,
        .bDeviceSubClass = USB_SUBCLASS_DEVICE,
        .bDeviceProtocol = USB_PROTOCOL_DEVICE,
        .bMaxPacketSize0 = 8,       // MUST match the control endpoint size
        .idVendor = USB_VID_FTDI,   // idVendor: 0x0403 (FTDI)
        .idProduct = USB_PID_DEVICE,// idProduct: User defined.
        .bcdDevice = 0x0101,
        .iManufacturer = 0x01,         // Manufacturer String
        .iProduct = 0x02,              // Product Stirng
        .iSerialNumber = 0x03,         // Serial Number String
        .bNumConfigurations = 0x01,
};
```

The idVendor and idProduct values (the USB VID and PID) **must** be unique to the application. FTDI applications for the FT51A use the FTDI VID and an example PID.

FTDI can assign a valid PID on request. FT51A applications assign PIDs in the range 0x0FE0 to 0x0FEF.

Do not use these values in a final product.

3.2.2.2 Configuration Descriptors

The ft51_usb.h file has typedefs for several configuration descriptor types. These can be combined in a struct to make a memory structure which can be initialised in code.

A sample configuration descriptor for a keyboard would be:

```
__code struct config_descriptor
{
    USB_configuration_descriptor configuration;
    USB_interface_descriptor interface;
    USB_hid_descriptor hid;
    USB_endpoint_descriptor endpoint;
};
struct config_descriptor config_descriptor =
{
        .configuration.bLength = 0x09,
        .configuration.bDescriptorType = USB_DESCRIPTOR_TYPE_CONFIGURATION,
        .configuration.wTotalLength = sizeof(struct config_descriptor_keyboard),
        .configuration.bNumInterfaces = 0x01,
        .configuration.bConfigurationValue = 0x01,
        .configuration.iConfiguration = 0x00,
        .configuration.bmAttributes =  USB_CONFIG_BMATTRIBUTES_SELF_POWERED |
            USB_CONFIG_BMATTRIBUTES_RESERVED_SET_TO_1,
        .configuration.bMaxPower = 0xFA,           // 500mA
        // ---- INTERFACE DESCRIPTOR for Keyboard ----
        .interface.bLength = 0x09,
        .interface.bDescriptorType = USB_DESCRIPTOR_TYPE_INTERFACE,
        .interface.bInterfaceNumber = 0,
        .interface.bAlternateSetting = 0x00,
        .interface.bNumEndpoints = 0x01,
        .interface.bInterfaceClass = USB_CLASS_HID,
        .interface.bInterfaceSubClass = 1,
        .interface.bInterfaceProtocol = 1,
        .interface.iInterface = 0x05,
        // ---- HID DESCRIPTOR for Keyboard ----
        .hid.bLength = 0x09,
        .hid.bDescriptorType = 0x21,
        .hid.bcdHID = USB_BCD_VERSION_HID_1_1,
        .hid.bCountryCode = USB_HID_LANG_NOT_SUPPORTED,
        .hid.bNumDescriptors = 0x01,
        .hid.bDescriptorType_0 = USB_DESCRIPTOR_TYPE_REPORT,
        .hid.wDescriptorLength_0 = 0x0041,
        // ---- ENDPOINT DESCRIPTOR for Keyboard ----
        .endpoint.bLength = 0x07,
        .endpoint.bDescriptorType = USB_DESCRIPTOR_TYPE_ENDPOINT,
        .endpoint.bEndpointAddress = 0x81,
        .endpoint.bmAttributes = USB_ENDPOINT_DESCRIPTOR_ATTR_INTERRUPT,
        .endpoint.wMaxPacketSize = 0x0008,
        .endpoint.bInterval = 0x0a, // 10mS
};
```

3.2.2.3 String Descriptors

The string descriptor is an array of bytes. The first byte of each descriptor is the length of the descriptor. This makes it possible for the array to be parsed to find a specified string in the array.

The standard_req_get_descriptor() function above will read through this byte array to find the requested string.

```
__code uint8_t string_descriptor[] =
{
        // String 0 is actually a list of language IDs supported
        // by the real strings.
        0x04, USB_DESCRIPTOR_TYPE_STRING,
        0x09, 0x04, // 0409 = English (US)
        // String 1 (Manufacturer): "FTDI"
        0x0a, USB_DESCRIPTOR_TYPE_STRING,
```

```
        'F', 0x00, 'T', 0x00, 'D', 0x00, 'I', 0x00,
        // String 2 (Product): "FT51A"
        0x0C, USB_DESCRIPTOR_TYPE_STRING,
        'F', 0x00, 'T', 0x00, '5', 0x00, '1', 0x00,
        'A', 0x00,
        // String 3 (Serial Number): "FT424242"
        0x12, USB_DESCRIPTOR_TYPE_STRING,
        'F', 0x00, 'T', 0x00, '4', 0x00, '2', 0x00,
        '4', 0x00, '2', 0x00, '4', 0x00, '2', 0x00,
        // END OF STRINGS
        0x00
};
```

FTDI applications will place the string descriptors in a block of 256 bytes starting at offset 0x80 in the program code. This is achieved using the following code.

```
// String descriptors - allow a maximum of 256 bytes for this
#define STRING_DESCRIPTOR_LOCATION 0x80
#define STRING_DESCRIPTOR_ALLOCATION 0x100

__code __at(STRING_DESCRIPTOR_LOCATION)uint8_t
string_descriptor[STRING_DESCRIPTOR_ALLOCATION] = {
```

The reason for this is to allow tools to edit the string descriptors. The ft51str.exe tool can edit string descriptors during the process of programming the device allowing unique serial numbers (or other strings) to be coded into the device.

3.2.3 Standard Requests

The standard request handler must return FT51_OK if the request was handled by the application or FT51_FAILED if not.

The USB Specification requires that the Set Address, Set Configuration, Get Configuration, Get Status, Get Descriptor for device and configuration descriptors be implemented. The sample code in this section illustrates the basic requests required to implement a USB device.

```
FT51_STATUS standard_req_cb(USB_device_request *req)
{
    FT51_STATUS status = FT51_FAILED;

    switch (req->bRequest)
    {
    case USB_REQUEST_CODE_GET_STATUS:
        status = standard_req_get_status(req);
        break;

    case USB_REQUEST_CODE_CLEAR_FEATURE:
    case USB_REQUEST_CODE_SET_FEATURE:
        status = standard_req_get_set_feature(req);
        break;

    case USB_REQUEST_CODE_GET_DESCRIPTOR:
        status = standard_req_get_descriptor(req);
        break;

    case USB_REQUEST_CODE_SET_CONFIGURATION:
```

```
        status = USB_set_configuration(req);
        break;

    case USB_REQUEST_CODE_GET_CONFIGURATION:
        status = USB_get_configuration();
        break;

    case USB_REQUEST_CODE_SET_ADDRESS:
        status = USB_set_address(req);
        break;

    default:
    case USB_REQUEST_CODE_GET_INTERFACE:
    case USB_REQUEST_CODE_SET_INTERFACE:
        // Unknown or unsupported request.
        status = FT51_FAILED;
        break;
    }
    return status;
}
```

This call-back is only activated when the USB_process() function is called after a SETUP packet has been received from the host.

If FT51_FAILED is returned then the USB_process() function will cause a stall on the control endpoint to signal to the host that the SETUP request failed.

3.2.3.1 Get Status

The minimum requirement for a response to the standard request GET_STATUS is for endpoint and device statuses.

```
FT51_STATUS standard_req_get_status(USB_device_request *req)
{
    USB_STATE state = USB_get_state();
    USB_ENDPOINT_NUMBER ep_number = LSB(req->wIndex) & 0x0F;
    USB_ENDPOINT_DIR ep_dir = (LSB(req->wIndex) >> 7);
    uint8_t buf[2];

    buf[0] = buf[1] = 0;

    // Get Status for endpoints only
    if (req->bmRequestType == (USB_BMREQUESTTYPE_DIR_DEV_TO_HOST |
            USB_BMREQUESTTYPE_RECIPIENT_ENDPOINT))
    {
        if (((state < CONFIGURED) && (ep_number == 0))
                || (state >= CONFIGURED))
        {
            if (USB_ep_stalled(ep_number, ep_dir))
            {
                buf[0] = USB_GET_STATUS_ENDPOINT_HALT;
            }
            USB_transfer(USB_EP_0, USB_DIR_IN, buf, 2);

            return FT51_OK;
        }
    }
    else if (req->bmRequestType == (USB_BMREQUESTTYPE_DIR_DEV_TO_HOST |
            USB_BMREQUESTTYPE_RECIPIENT_DEVICE))
    {
        // This must match the configuration descriptor's bmAttributes
        buf[0] = USB_GET_STATUS_DEVICE_SELF_POWERED;
```

```
        USB_transfer(USB_EP_0, USB_DIR_IN, buf, 2);

        return FT51_OK;
    }
    return FT51_FAILED;

}
```

The endpoint status may only return a valid status for endpoint zero when the device is not configured. When the device is in the configured state then it can return statuses for all endpoints.

3.2.3.2 Set Features

The control endpoints must respond to SET_FEATURE and CLEAR_FEATURE requests. This example code will provide a response for endpoint stall SET and CLEAR operations.

```
FT51_STATUS standard_req_get_set_feature(USB_device_request *req)
{
    USB_ENDPOINT_NUMBER ep_number = LSB(req->wIndex) & 0x0F;
    USB_ENDPOINT_DIR ep_dir = (LSB(req->wIndex) >> 7);
    USB_STATE state = USB_get_state();

    if (req->bmRequestType == (USB_BMREQUESTTYPE_DIR_DEV_TO_HOST |
            USB_BMREQUESTTYPE_RECIPIENT_ENDPOINT))
    {
        // Only support the endpoint halt feature in this device
        if (req->wValue == USB_FEATURE_ENDPOINT_HALT)
        {
            // Only allow this if the device is configured or the endpoint
            // is zero if the device is not configured.
            if (((state < CONFIGURED) && (ep_number == 0))
                    || (state >= CONFIGURED))
            {
                // Perform a stall or a clear
                if (req->bRequest == USB_REQUEST_CODE_CLEAR_FEATURE)
                {
                    // Feature selector CLEAR
                    USB_clear_endpoint(ep_number, ep_dir);
                }
                else
                {
                    // Feature selector SET
                    USB_stall_endpoint(ep_number, ep_dir);
                }
                USB_transfer(USB_EP_0, USB_DIR_IN, NULL, 0);
                return FT51_OK;
            }
        }
    }
    return FT51_FAILED;

}
```

The endpoint features only apply to endpoint zero when the device is not configured. When the device is configured then it can apply to all endpoints.

3.2.3.3 Get Descriptors

The Get Descriptor response depends on the type of descriptor requested. Only device and configuration descriptors are required but string descriptors are commonly used.

A simple handler for device, configuration and string descriptors would be as follows. It is assumed that the byte arrays "config_descriptor", "device_descriptor" and "string_descriptor" are defined. See sections 3.2.2, 3.2.2.2 and 3.2.2.3.

```
FT51_STATUS standard_req_get_descriptor(USB_device_request *req)
{
    // pointer to descriptor (or part of) to send to host
    uint8_t   *src = NULL;
    // length of data requested by the host
    uint16_t   length = req->wLength;
    uint8_t    hValue = req->wValue >> 8;
    uint8_t    lValue = req->wValue & 0x00ff;
    uint8_t    I, slen;

    switch (hValue)
    {
    case USB_DESCRIPTOR_TYPE_DEVICE:
        src = (char *)&device_descriptor;
        if (length > sizeof(USB_device_descriptor)) // too many bytes requested
            length = sizeof(USB_device_descriptor); // Entire structure.
        break;        break;

    case USB_DESCRIPTOR_TYPE_CONFIGURATION:
        src = (char *)&config_descriptor;
        if (length > sizeof(config_descriptor)) // too many bytes requested
            length = sizeof(config_descriptor); // Entire structure.
        break;

    case USB_DESCRIPTOR_TYPE_STRING:
        // Find the nth string in the string descriptor table
        i = 0;
        while ((slen = string_descriptor[i]) > 0) {
            // Point to start of string descriptor in __code segment
            src = (uint8_t *)&string_descriptor[i];
            if (lValue == 0) {
                break;
            }
            i += slen;
            lValue--;
        }
        if (lValue > 0) {
             return FT51_FAILED; // String not found
        }
        // Update the length returned only if it is less than the requested
        // size
        if (length > slen) {
            length = slen;
        }
        break;

    default:
        return FT51_FAILED;
    }
    USB_transfer(USB_EP_0, USB_DIR_IN, src, length);
    return FT51_OK;
}
```

The device and configuration descriptors are defined as byte arrays in the __code segment, the string descriptor is an array of byte arrays in the __code segment.

3.2.4 Class and Vendor Requests

Like the standard request handler, class and vendor request handlers must return FT51_OK if the request was handled by the application or FT51_FAILED if not.

The format of the class and vendor request functions is the same as the standard requests. This example provides a class handler for a HID keyboard.

```
uint8_t input_report;
uint8_t report_mode;

FT51_STATUS class_req_cb(USB_device_request *req)
{
    FT51_STATUS status = FT51_FAILED;
    uint8_t interface = LSB(req->wIndex) & 0x0F;

    // Ensure the recipient is an interface...
    // Can also check if the interface number is correct
    if ((req->bmRequestType & USB_BMREQUESTTYPE_RECIPIENT_MASK) ==
        USB_BMREQUESTTYPE_RECIPIENT_INTERFACE)
    {
        // Handle HID class requests
        switch (req->bRequest)
        {
        case USB_HID_REQUEST_CODE_SET_IDLE:
            // Turn on report mode to start sending data to host
            report_mode = 1;
            USB_transfer(USB_EP_0, USB_DIR_IN, NULL, 0);
            status = FT51_OK;
            break;

        case USB_HID_REQUEST_CODE_SET_PROTOCOL:
            USB_transfer(USB_EP_0, USB_DIR_IN, NULL, 0);
            status = FT51_OK;
            break;

        case USB_HID_REQUEST_CODE_SET_REPORT:
            // dummy read of one byte
            USB_transfer(USB_EP_0, USB_DIR_OUT, &input_report, 1);
            // Acknowledge SETUP
            USB_transfer(USB_EP_0, USB_DIR_IN, NULL, 0);
            status = FT51_OK;
            break;
            }
        }
    }
    return status;
}
```

3.2.5 Call-backs

The suspend, resume and reset call-backs are optional.

3.2.5.1 Reset Call-back

The reset call-back is needed when the device is running self-powered. The USB state is part of the library rather than an internal state in the USB Full Speed device controller and therefore needs to update when the device is reset.

Endpoint creation is best placed in this function, as the host will reset a USB device during enumeration. This allows us to disable and then re-enable an endpoint during a reset or when the device is first powered up.

This is an example of making an interrupt IN endpoint with a maximum packet size of 8.

```
void reset_cb(uint8_t status)
{
    (void) status;
    USB_free_endpoint(1, USB_DIR_IN);
    USB_set_state(DEFAULT);
    /* Make application specific BULK, INTERRUPT or
     * ISOCHRONOUS endpoints. */
    int_in = USB_create_endpoint(USB_EP_1, USB_EP_INT,
            USB_DIR_IN, USB_EP_SIZE_8);
    return;
}
```

3.2.5.2 Suspend Call-back

The suspend call-back is used when either the host puts the device into suspend mode or the device is self-powered and is disconnected from the host.

Any special programming required to handle suspend states should be placed in here.

3.2.5.3 Resume Call-back

The resume call-back is used when either the host moves the device from suspend to resume mode or the device is self-powered and is reconnected to the host.

Any special programming required for handling resume states should be placed in here.

3.2.6 Main Function

The main function needs to initialise the device then call USB_process() periodically to handle SETUP requests.

```
void main(void)
{
    // Initialise FT51
    device_initialise();

    // Initialise USB function
    usb_setup();

    // Endless loop.
    while (1)
    {
        // Check control endpoints and handle SETUPs
        USB_process();
    };
}
```

Code required to send data or receive data from non-control endpoints is placed within the while loop.

The only requirement for the main loop is the that USB_process() is called periodically. The periodicity of the call must ensure that any SETUP packet received is responded to within the time limits defined for the call in the USB specification or the class or vendor specification applicable.

3.2.7 Sending and Receiving Data

Data sent and received from control endpoints are handled by the USB_process() function using call-backs to process standard, class and vendor requests. Bulk, interrupt and isochronous endpoints use the USB_transfer() function to send data when required.

When receiving data from the USB host, if data is available then the USB_transfer() function will return a valid status and zero or more bytes of data. This happens asynchronously with the data only being available to USB_transfer() once the OUT transfer is complete.

To transmit data to the host, the USB_transfer() function will write the data to send into a buffer and the host will poll the endpoint to receive the data. If there is already data waiting to send to the host from this endpoint then the USB_transfer() function will wait a small amount of time for the host and then return from USB_transfer() with a failure.

All endpoints have 2 buffers of 64 bytes, allowing one transaction to be in progress while another is waiting. This is transparent to the application using the USB device.

3.2.8 Link Power Management

Link Power Management (LPM) feature of USB is an extra USB power state called L1 (Sleep) found in-between states L0 (On) and L2 (Suspend). The advantage of this, is its shorter transition latencies. Instead of latencies in excess of 20 ms, it ensures transitions in tens of microseconds, and leaves it up to the user to decide if to reduce the power or not.

The host or hub initiates entry to L1 by sending LPM extended transaction. It is a sequence of a setup token packet with a PID = 0000, followed by extended token packet with a PID = 0011 and with bmAttributes set appropriately. A downstream device transitions to L1 as soon as the host or hub detects an ACK handshake. Exiting this state is via remote wakeup, resume signalling, reset or disconnect.

Refer to the USB 2.0 Link Power Management Addendum found in http://www.usb.org/developers/docs/usb20_docs/ for further information.

The USB Full Speed device controller is capable of acknowledging LPM transaction. Furthermore, the FT51A sees the link state change as an interrupt in the ISR. Below is an extract of an example ISR that can be used to detect a LPM state transition, plus a function that can be invoked whenever a LPM transition occurred. Note that the function must be called from the USB_process().

```
uint8_t                                LPM_L1 = 0x00;
uint8_t                                lpm_status[2] = {42, 42};
uint8_t                                first_time = 1;

/**
    @brief   \par USB Interrupt Service Routine
    @details Called when an interrupt is received from USB.  This function
             determines the source of the interrupt and sets bits within the
             data structures accordingly.  For reset and suspend it will call
             the appropriate callback functions if specified.
    @note    Please refer to section 6.3.1 of the FT122 Data Sheet for more
             information.
**/
void USB_ISR(const uint8_t flags)
{
    uint8_t int_reg[4];

    (void)flags; // Suppress warning about unused parameter.

    FT122_CMD = RD_INTERRUPT_REG;
    int_reg[0] = FT122_DATA;
    int_reg[1] = FT122_DATA;

    if (int_reg[1] & LPM_CHANGE_INT)
    {
```

```
        LPM_L1 = 1; // Set this bit to indicate LPM-triggered transition from L0 (Active) to
L1 (Sleep).
                    if (first_time)
                    {
                              first_time = 0;
                              FT122_CMD = READ_LPM_STATUS;
                              lpm_status[0] = FT122_DATA;
                              lpm_status[1] = FT122_DATA;
                    }
    }
}

void process_setup_packet(void) // called from USB_process()
{
    USB_request_callback cb = NULL;
    uint8_t              data;
    uint8_t              i;
    FT51_STATUS          status = FT51_FAILED;

    debug_printf("--- process_setup_packet ---\r\n");

    if (LPM_L1 == 1)
    {
         debug_printf("+++ LPM_L1 = 1\r\n");
         debug_printf("+++ LPM Status = %02x %02x\r\n", lpm_status[0], lpm_status[1]);
    }
    else
    {
         debug_printf("+++ LPM_L1 = %d\r\n", LPM_L1);
         debug_printf("+++ LPM Status = %02x %02x\r\n", lpm_status[0], lpm_status[1]);
    }
}
```

4 Contact Information

Head Quarters – Singapore

Bridgetek Pte Ltd
178 Paya Lebar Road, #07-03
Singapore 409030
Tel: +65 6547 4827
Fax: +65 6841 6071

E-mail (Sales) sales.apac@brtchip.com
E-mail (Support) support.apac@brtchip.com

Branch Office – Taipei, Taiwan

Bridgetek Pte Ltd, Taiwan Branch
2 Floor, No. 516, Sec. 1, Nei Hu Road, Nei Hu District
Taipei 114
Taiwan, R.O.C.
Tel: +886 (2) 8797 5691
Fax: +886 (2) 8751 9737

E-mail (Sales) sales.apac@brtchip.com
E-mail (Support) support.apac@brtchip.com

Branch Office - Glasgow, United Kingdom

Bridgetek Pte. Ltd.
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales) sales.emea@brtichip.com
E-mail (Support) support.emea@brtchip.com

Branch Office – Vietnam

Bridgetek VietNam Company Limited
Lutaco Tower Building, 5th Floor, 173A Nguyen Van Troi,
Ward 11, Phu Nhuan District,
Ho Chi Minh City, Vietnam
Tel : 08 38453222
Fax : 08 38455222

E-mail (Sales) sales.apac@brtchip.com
E-mail (Support) support.apac@brtchip.com

Web Site

http://brtchip.com/

Distributor and Sales Representatives

Please visit the Sales Network page of the Bridgetek Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – References

Document References

FTDI FT51A web page: http://www.ftdichip.com/Products/ICs/FT51.html

FT51A Datasheet: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT51.pdf

- AN_344 FT51A DFU Sample
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_344_FT51A_DFU_Sample.pdf
- AN_345 FT51A Keyboard Sample
  http://www.ftdichip.com/Support/Documents/AppNotes/an_345_ft51a_keyboard_sample.pdf
- AN_346 FT51A Mouse Sample
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_346_FT51A_Mouse_Sample.pdf
- AN_347 FT51A Test and Measurement Sample
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_347_FT51A_Test_and_Measurement_Sample.pdf
- AN_348 FT51A FT800 Sensors Sample
- http://www.ftdichip.com/Support/Documents/AppNotes/AN_348_FT51A_FT800_Sensors_Sample.pdf
- AN_349 FT51A FT800 Spaced Invaders Sample
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_349_FT51A_FT800_Spaced_Invaders_Sample.pdf
- AN_351 FT51A Compatibility Module
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_351_FT51A_Compatibility_Module.pdf
- AN_354 FT51A Standalone Demo Application
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_354_FT51A_Standalone_Demo_Application.pdf
- AN_352 FT51A Installation Guide
  http://www.ftdichip.com/Support/Documents/AppNotes/AN_352_FT51A_Installation_Guide.pdf

USB Test and Measurement Class specification:

http://www.usb.org/developers/docs/devclass_docs/USBTMC_1_006a.zip

USB Device Firmware Update Class specification:

http://www.usb.org/developers/docs/devclass_docs/DFU_1.1.pdf

SDCC (Small Device C Compiler) reference:

http://sdcc.sourceforge.net/

FTDI FT122 USB full-speed device controller:

http://www.ftdichip.com/Products/ICs/FT122.html

Acronyms and Abbreviations

Terms	Description
USB	Universal Serial Bus
USB-IF	USB Implementers Forum
MTP	Multiple Time Program – non-volatile memory used to store program code on the FT51A.
SDCC	Small Device C Compiler
GPL	Gnu Public License
EPL	Eclipse Public License

Appendix B – List of Tables & Figures

List of Tables

List of Figures

Appendix C – Revision History

Document Title: AN_289 FT51A Programming Guide

Document Reference No.: BRT_000034

Clearance No.: BRT#041

Product Page: http://brtchip.com/product

Document Feedback: Send Feedback

Revision	Changes	Date
1.0	Initial Release	2015-12-21
1.1	Updated contact details Dual branding to reflect the migration of the product to the Bridgetek name – logo changed, copyright changed	2016-09-19