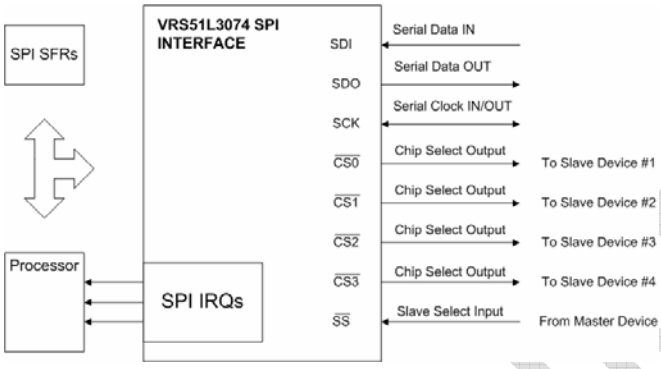
9 SPI Interface

The SPI interface of the VRS51L3074's provides numerous enhancements compared to other vendor offerings. The SPI interface's key features include:

- Supports four standard SPI modes (clock phase/polarity)
- Operates in master and slave modes
- Automatic control of up to four chip select lines
- Configurable transaction size (1 to 32 bits)
- Transaction size of >32 bits is possible
- Double Rx and TX data buffers
- Configurable MSB or LSB first transaction
- Generation frame select/load signals

FIGURE 14: SPI INTERFACE OVERVIEW



Before the SPI can be accessed it must first be enabled by setting the SPIEN bit of the PERIPHEN1 register to 1.

9.1 SPI Control Registers

The SPICTRL register controls the operating modes of the SPI interface in master mode.

TABLE 97:SPI CONTROL REGISTER - SPICTRL SFR C1H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	1

Bit	Mnemonic	Description
7	SPICLK[2:0]	SPI Communication Speed (Master Mode) 000 = Sys Clk / 2 (/ 8 if SPISLOW = 1) 001 = Sys Clk / 4 (/ 16 if SPISLOW = 1) 010 = Sys Clk / 8 (/ 32 if SPISLOW = 1) 011 = Sys Clk / 16 (/ 64 if SPISLOW = 1) 100 = Sys Clk / 32 (/ 128 if SPISLOW = 1) 101 = Sys Clk / 64 (/ 256 if SPISLOW = 1) 110 = Sys Clk / 128 (/ 512 if SPISLOW = 1) 111 = Sys Clk / 256 (/ 1024 if SPISLOW = 1)
4	SPICS[1:0]	SPI Active Chip Select Line (Master Mode) 00 = CS0 is active 01 = CS1 is active 10 = CS2 is active 11 = CS3 is active
2	SPICLKPH	SPI Clock Phase 0 = SD0 output on rising edge and SDI sampling on falling edge 1= SD0 output on falling edge and SDI sampling on rising edge
1	SPICLKPOL	SPI Clock Polarity 0 = SCK stays at 0 when SPI is inactive 1 = SCK stays at 1 when SPI is inactive
0	SPIMASTER	SPI Master Mode Enable 0 = SPI operates in slave mode 1 = SPI operate in master mode (default)

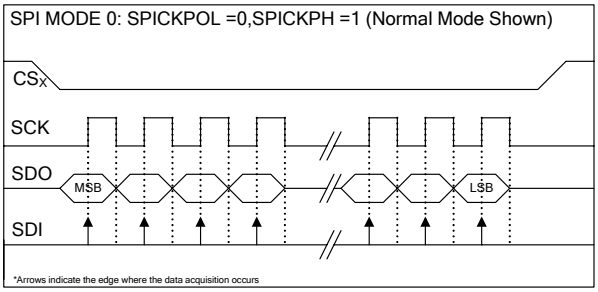
When the SPIMASTER bit is set to 1, the SPI interface operates in master mode. This is the default operating mode of the VRS51L3074 SPI interface after reset.

9.2 Setting Up Clock Phase and Polarity

The clock phase and polarity is controlled by the SPICLKPH and SPICLKPOL bits, respectively. The following diagrams show the communication timing associated with the clock phase and polarity.

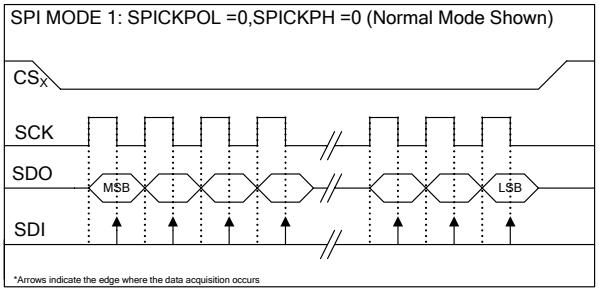
SPI Mode 0:

FIGURE 15: SPI MODE 0



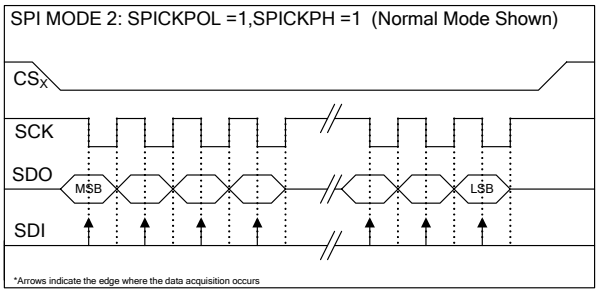
SPI Mode 1:

FIGURE 16: SPI MODE 1



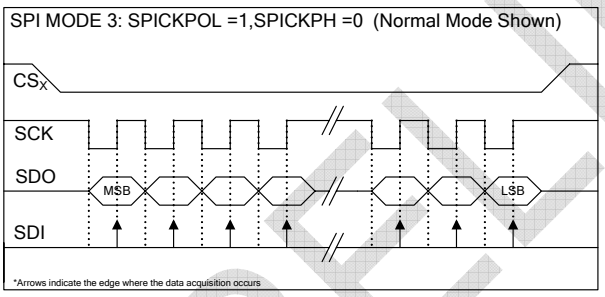
SPI Mode 2:

FIGURE 17: SPI MODE 2



SPI Mode 3:

FIGURE 18: SPI MODE 3



9.3 Defining active chip select line

As previously mentioned, only one chip select line is activated when communicating with an external SPI slave device. The SPICS bits of the SPICTRL register are used to select which CS line will be activated during the transfer.

Note that with the exception of the CS0 line, the SPICSEN bit of the PERIPHEN1 register must be set to 1 in order for the SPI be able to control the SPI CS lines.

9.4 Setting the SPI Communication Speed (Master Mode)

In master mode, the SPI interface communication speed is adjustable from “system clock /2” down to “system clock / 1024”. Slower communication speeds can be useful for interfacing with slower devices or to adjust the communication speed to specific bus conditions.

The SPICLK SFR register and the SPISLOW bit of the of the SPICONFIG SFR register control the SPI communication speed.

The SPI communication speed in master mode can be calculated using the following formula:

$$\text{SPI speed} = \frac{\text{Sys Clk}}{[\ 2^{(\text{SPICLK[2:0]}+1)} \times 4^{\text{SPISLOW}}\ ]}$$

Where:

- Sys Clk = Processor operating clock
- SPISLOW = can be either 0 or 1
- SPICLK[2:0] = from 0 to 7

The following tables provide example setting for SPI communication speeds with various system clock and SPICLK[2:0] and SPISLOW bit settings.

TABLE 98:SPI COMMUNICATION SPEED EXAMPLE (SPISLOW = 0)

SPICLK	Com Speed @ 40MHz	Com Speed @ 22.18MHz	Com Speed @ 4MHz
000	20 MHz	11.05 MHz	2 MHz
001	10 MHz	5.53 MHz	1 MHz
010	5 MHz	2.76 MHz	500 kHz
011	2.5 MHz	1.38 MHz	250 kHz
100	1.25 MHz	691.2 kHz	125 kHz
101	625 kHz	345.6 kHz	62.5 kHz
110	312.5 kHz	172.8 kHz	31.3 kHz
111	156.3 kHz	86.4 kHz	15.6 kHz

TABLE 99:SPI COMMUNICATION SPEED EXAMPLE (SPISLOW = 1)

SPICLK	Com Speed @ 40MHz	Com Speed @ 22.18MHz	Com Speed @ 4MHz
000	5 MHz	2.76 MHz	500 kHz
001	2.50 MHz	1.38 MHz	250 kHz
010	1.25 MHz	691.2 kHz	125 kHz
011	625 kHz	345.6 kHz	62.5 kHz
100	312.5 kHz	172.8 kHz	31.3 kHz
101	156.3 kHz	86.4 kHz	15.6 kHz
110	78.1 kHz	43.2 kHz	7.8 kHz
111	39.1 kHz	21.6 kHz	3.9 kHz

9.5 SPI Configuration and Status Registers

The SPI configuration and status registers allow the activation and the monitoring of the SPI interface interrupts. They also provide access to the advanced features of the SPI interface such as:

- Frame select/load generation on CS3
- Activating manual control of the chip select lines
- Bit reversed mode (Bitwise Endian Control)
- Interrupt activation and monitoring
- Monitoring the state of the SS pin

TABLE 100:SPI CONFIGURATION REGISTER - SPICONFIG - C2H

7	6	5	4	3	2	1	0
R/W	W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	SPIMANCS	SPI Manual CS Mode Enable 0 = SPI Chip select control is fully automatic 1 = SPI Chip select will be brought low by the SPI interface, and will stay low until 0 is written into SPIMANCS bit
6	SPIUNDERC	SPI Clear TX Underrun Flag (SPIUNDERF) Writing a 1 into this bit will clear the SPIUNDER bit of the SPISTATUS register This bit always reads 0
5	FSONCS3	Frame Select Pulse on CS3 0 = CS3 acts in standard ways 1 = The SPI interface will send an active low frame select pulse on CS3 Frame select has priority on SPILOAD function
4	SPILOADCS3	Load Pulse on CS3 0 = CS3 acts in standard way or as frame select pulse, if FSONCS3 is set to 1 1 = The SPI interface sends an active low load pulse on the CS3 pin, if FSONCS3 is cleared
3	SPISLOW	SPI Slow Speed mode 0 = SPI transaction occurs at normal speed 1 = SPI transaction is 4x slower
2	SPIRXOVEN	SPI RX Overrun Interrupt Enable 0 = SPI RX overrun interrupt is deactivated 1 = SPI RX overrun interrupt is enabled
1	SPIRXAVEN	SPI RX Available Interrupt Enable 0 = SPI RX available interrupt is deactivated 1 = SPI RX available interrupt is enabled
0	SPITXEEN	SPI TX Empty Interrupt Enable 0 = SPI TX empty interrupt is deactivated 1 = SPI TX empty interrupt is enabled

The SPISTATUS register's role is mainly for monitoring purposes.

TABLE 101:SPI STATUS REGISTER - SPISTATUS SFR C9H

7	6	5	4	3	2	1	0
R/W	R	R	R	R	R	R	R
0	0	0	1	1	0	0	1

Bit	Mnemonic	Description
7	SPIREVERSE	SPI Reverse Mode 0 = SPI operates in normal mode (MSB First) 1 = SPI operates in reverse mode (LSB First)
6	-	Not used
5	SPIUNDERF	SPI TX Underrun Flag 0 = No underrun condition noticed 1 = Indicates that the SPI transmit buffer has not been fed in time. This condition is likely to occur when the Transaction size is > 32 bits This bit is cleared by setting to 1, the SPICLRTXF bit of the SPICTRL bit of the SPICONFIG register
4	SSPINVAL	Slave Select Pin Value 0 =SS pin is low 1 = SS pin is high
3	SPINOCS	SPI No Chip Select 0 = At least on chip select line is active 1 = Indicates that all the chip select lines are inactive (high)
2	SPIRXOVF	SPI RX Overrun InterruptFlag 0 = No SPI RX Overrun condition detected 1 = SPI Data collision occurred
1	SPIRXAVF	SPI RX Available Interrupt Flag 0 = SPI receive buffer is empty 1 = Data is present in the SPI RX buffer
0	SPITXEMPF	SPI TX Empty Interrupt Flag 0 = SPI transmit buffer is full 1 = SPI transmit buffer is ready to receive new data

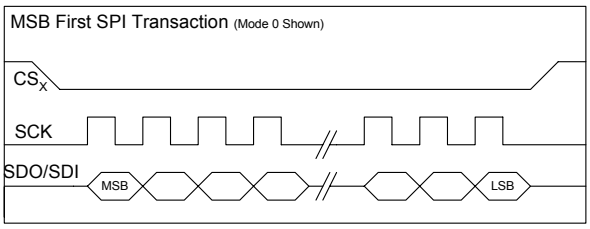
9.6 SPI Transaction Directions

The SPI interface can perform transactions in the standard SPI format (MSB first) as well as in the reverse format (LSB first). In applications where data must be transmitted (or received) in LSB first format, the user would normally need to perform bit reversal manually at the processor level and then send the data through the SPI interface. The SPI interface can automatically handle the bit reversal operations, unloading the processor for other tasks.

When the SPIREVERSE bit of the SPISTATUS register is set to 0, the SPI transactions will take place in MSB first format.

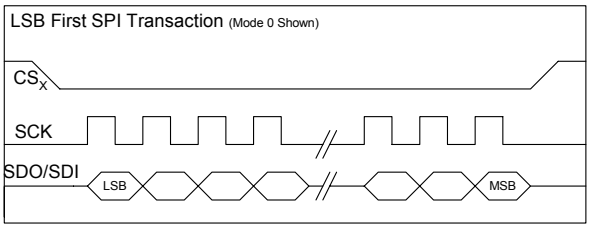
The following examples show the timing related to these transaction directions:

FIGURE 19: SPI MSB FIRST TRANSACTION



When the SPIREVERSE is set to 1, the SPI transactions are done in LSB first format, as shown in the next figure.

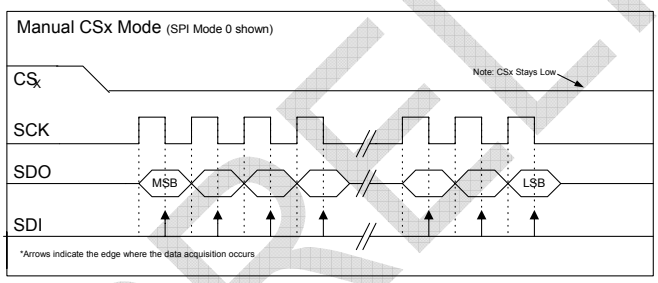
FIGURE 20: SPI LSB FIRST TRANSACTION



9.7 Manual Chip Select Control

When the SPIMANCS bit of the SPICONFIG register is set to 1, the active chip select line will stay at a logic low after the SPI master mode transaction is completed, as shown in the following figure.

FIGURE 21: SPI MANUAL CHIP SELECT



The chip select will remain at logic 0 until the SPIMANCS bit is cleared by the software.

9.8 SPI Interrupts

The SPI can trigger three interrupt sources that are handled by two interrupt vectors, as shown in the following table:

TABLE 102: SPI INTERRUPT SOURCES

Interrupt	Interrupt Number	Interrupt Vector
SPI TX Empty	Int_1	000Bh
SPI RX Available	Int_2	0013h
SPI RX Overrun		

The TX empty interrupt is set when the SPI transmit buffer is ready to receive more data. A double buffer is used in the SPI transmitter. Once transmission begins (after a write to the SPIRXTX0 register), the data is transferred to the final transmission buffer. This frees up the SPIRXTX SFR register, raises the SPITXEMPF flag of the status register and triggers an SPI TX empty interrupt if enabled. The SPI TX empty interrupt is enabled by setting the SPITXEEN bit of the SPICONFIG register to 1.

The priority of the SPI TX empty interrupt is set high in order to avoid buffer overrun in 32-bit SPI transfers.

The SPI RX available interrupt is activated when receive data has been transferred from the SPI RX buffer to the SPIRXTX register. The SPIRXTX register must be read by the processor before the next SPI bus data sequence is completed. The SPI RX available interrupt is enabled by setting the SPIRXAVEN bit of the SPICONFIG register to 1. The SPIRXAVF flag of the SPISTATUS register, when set to 1, indicates that data can be read. The SPIRXAVF flag is automatically reset when the SPIRXTX0 register is read.

The SPI RX overrun interrupt indicates that an overrun condition has taken place. The SPI RX overrun interrupt is enabled by setting the SPIRXOVEN bit of the SPICONFIG register to 1. The SPIRXOVF flag of the SPISTATUS register, when set to 1, indicates that a data collision has occurred.

All the SPI interface interrupt flags are active even if the associated interrupt is not activated and they can be monitored by the user program at any time.

Please consult the Interrupt Section for more details on the SPI interface interrupts and their interaction with other peripherals

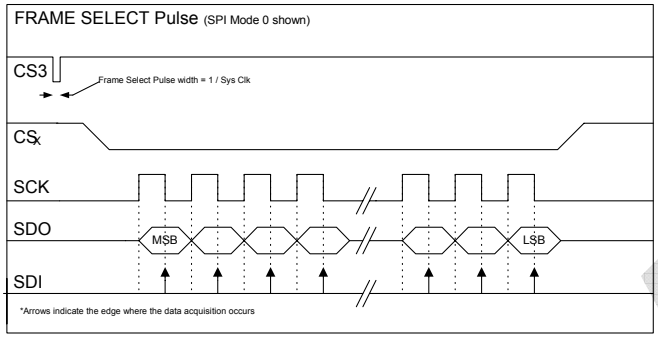
9.9 Alternate CS3 functions

For external SPI devices which require the use of a load or a frame select signal, the VRS51L3074 can be configured to either generate an active low frame select or active high load signal when operating in master mode.

9.9.1 Frame Select signal on CS3

When the FONCS3 bit of the SPICONFIG register is set to 1, the SPI interface will generate an active low frame select pulse on the CS3 pin (see the following timing diagram).

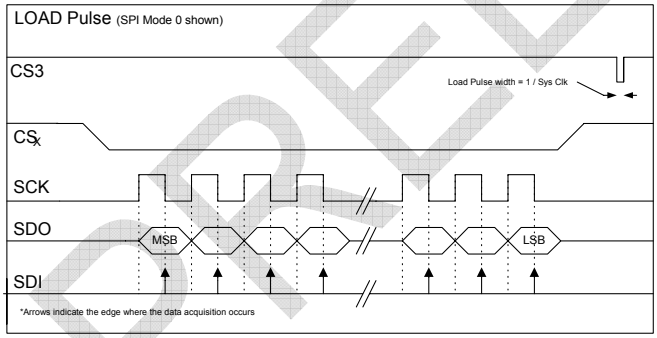
FIGURE 22: SPI FRAME SELECT PULSE TIMING



9.9.2 Load Signal on CS3

When the SPILOADCS3 bit of the SPICONFIG register is set to 1 *and* the FSONCS3 bit is cleared, an active low load signal will be generated on the CS3 line of the SPI interface.

FIGURE 23: SPI LOAD PULSE TIMING



Note that the frame select alternate function has priority over the load function. This means that if the FSONCS3 bit is set, the alternate function selected will be the frame select, independent of the value of the SPILOAD bit.

9.10 SPI Activity Monitoring

The ability to monitor the state of communication of the SPI interface can be useful in highly modular applications in which the SPI interface is handled by interrupts. The SPISTATUS register contains two flags that can be used to monitor the CS and SS signals of the SPI interface.

The SPINOCS bit of the SPISTATUS register returns the logical AND of all the SPI CS lines of the VRS51L3074. If all the CS lines are inactive (logic high), the SPI interface sets the SPINOCS to 1. The SPINOCS bit is used to verify that the SPI interface is idle before reconfiguring it or starting a new transaction.

The SPINOCS bit of the SPISTATUS register is valid four system clock cycles after the SPI transmission begins. This delay is independent of the SPI transaction speed.

As such, after a write operation to the SPIRXTX0 register, which will trigger a SPI transaction in master mode, a NOP instruction (1 cycle) must be added before the MOV Rn, SPISTATUS instruction (3 cycles).

The SSPINVAL bit of the SPISTATUS register returns the logic level on the SS pin.

9.11 SPI TX Underrun Flag

The SPI interface provides an underrun condition flag that can be used to flag whether the software has failed to update transmission buffer in time for the next transfer. This is especially useful when the SPI interface is used to transmit packets greater than 32 bits in length.

If an underrun condition occurs, the SPIUNDERF bit of the SPI status register will be set to 1. This bit can be cleared by writing a 1 to the SPIUNDERC bit of the SPICONFIG register.

Note that SPI underrun monitoring is not linked to any of the SPI interrupts, therefore, this flag can only be v manually by software

9.12 SPI Transaction Size

The standard SPI protocol is based on 8-bit transactions. However, many devices on the market, specifically A/D and D/A converters, require transactions greater than 8 bits. To communicate with these types of devices using a standard SPI interface, the user has no choice but to send multiple 8-bit streams or to manipulate the I/Os via software to emulate the timing control signals.

The VRS51L3074 SPI interface supports 8-bit transactions and can also be configured to support transactions that measure 1 to 32 bits in both transmit and receive directions. The value written into the SPISIZE register controls the transaction size. Upon reset, the SPI interface is configured for 8-bit transactions.

TABLE 103:SPI TRANSACTION SIZE – SPISIZE SFR C3H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	1	1	1

Bit	Mnemonic	Description
7:0	SPISIZE[7:0]	SPI transaction Size If < 32 : Transaction Size = SPISIZE + 1 If >= 32: Transaction Size = (SPISIZE *8) - 216 Default Transaction Size = 8 bits

Four formulas control the SPI transaction size:

For Transactions Size <= 32 bits

Transaction Size = SPISIZE[7:0] +1

Or

SPISIZE[7:0] = Transaction Size - 1

For Transactions Size > 32 bits

Transaction Size = [(SPISIZE[7:0] * 8) –216]

Or it can be expressed by:

$$SPISIZE[7:0] = \frac{[\text{ Transaction Size} + 216\ ]}{8}$$

The following table provides examples:

TABLE 104: TRANSACTION SIZE VS. SPISIZE[7:0]

SPISIZE[7:0]	Transaction Size
0x07	8-bit
0x0B	12-bit
0x0D	14-bit
0x10	17-bit
0x17	24-bit
0x1F	32-bit
0x20	40-bit
0x21	48-bit
0x23	64-bit

The transaction size must also be configured when the operating the SPI interface in slave mode.

9.13 SPI RX/TX Data Registers

Four SFR registers provide access to the SPI interface's receive and transmit data buffer. Performing a write operation to the SPI RX/TX buffer transfers the data to the transmit portion of the SPI interface, while a read operation reads the contents of the receive data buffer. The SPI 32-bit receive and transmit data buffers are double buffered to minimize the risk of data collision and to achieve optimal performance.

The SPI RXTX0 register contains bits 7:0 of the SPI interface RX/TX register.

TABLE 105: SPIRXTX0 REGISTER CONTENT FOR NORMAL AND REVERSED TRANSACTIONS

Operation	SPI Mode	SPIRXTXx Data is…
Read	MSB First	Right Justified
	LSB First	Left Justified
Write	MSB First	Left Justified
	LSB First	Right Justified

When the SPI is configured in master mode, writing to the SPIRXTX0 will trigger a data transmission. For this reason, when the transaction size is larger than 8 bits, the SPIRXTX0 register must be written last.

TABLE 106:SPI RX / TX0 DATA REGISTER – SPIRXTX0 SFR C4H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	SPIRXTX0[7:0]	Read: SPI RXData[7:0] Right justified in normal mode, left justified in bit reversed mode Reading this register, clears the SPIAVF and SPIRXOVF flags
		Write: SPI TXData[7:0] Left justified in normal mode, right justified in bit reversed mode In master mode, writing to SPIRXTX0 triggers the transmission

TABLE 107:SPI RX / TX1 DATA REGISTER – SPIRXTX1 SFR C5H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	SPIRXTX1[7:0]	Read: SPI RXData[15:8] Right justified in normal mode, left justified in bit reverse mode
		Write: SPI TXData[15:8] Left justified in normal mode, right justified in bit reverse mode

TABLE 108:SPI RX / TX2 DATA REGISTER – SPIRXTX2 SFR C6H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	SPIRXTX2[7:0]	Read: SPI RXData[23:16] Right justified in normal mode, left justified in bit reverse mode
		Write: SPI TXData[23:16] Left justified in normal mode, right justified in bit reverse mode

TABLE 109:SPI RX / TX3 DATA REGISTER – SPIRXTX3 SFR C7H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	SPIRXTX3[7:0]	Read: SPI RXData[31:24] Right justified in normal mode, left justified in bit reverse mode
		Write: SPI TXData[31:24] Left justified in normal mode, right justified in bit reverse mode

9.14 SPI Data Input /Output

The VRS51L3074 SPI interface has the ability to perform data transactions in MSB first mode or LSB first. The SPIREVERSE bit of the SPISTATUS register controls whether the data will be transmitted MBS first or LSB first. Upon device reset, the SPIREVERSE bit equals 0 and data is transmitted in MSB first format.

The SPIREVERSE bit state will also affect the data transmission and the data reception buffer structure as shown in the following diagrams.

FIGURE 24: SPI TRANSACTION STANDARD MODE (SPIREVERSE = 0 : MSB FIRST)

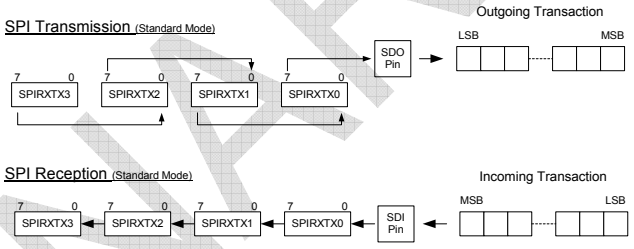
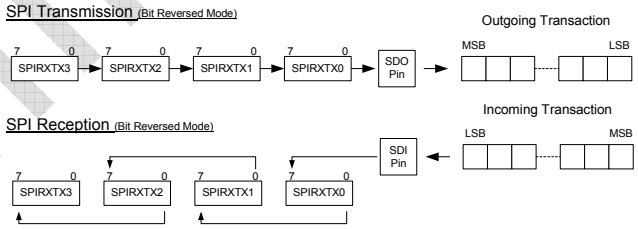


FIGURE 25: SPI TRANSACTION BIT REVERSE MODE (SPIREVERSE = 1: LSB FIRST)



The next tables gives examples of SPI transmission and reception in different modes if the SPI SDO pin is connected to the SDI pin.

SPISIZE = 0x0F (16 bit) / SPIREVERSE= 0 (MSB First

SPITX [3:0]				SPIRX [3:0]			
xx	xx	D3h	42h	xx	xx	42h	D3h
xx	xx	54h	A6h	xx	xx	A6h	54h

SPISIZE = 0x0F (32 bit) / SPIREVERSE= 0 (MSB First

SPITX [3:0]				SPIRX [3:0]			
45h	A3h	B2h	DF	DFh	B2h	A3h	45h
C3h	8Ah	49h	24h	24h	49h	8Ah	C3h

SPISIZE = 0x0F (32 bit) / SPIREVERSE= 1 (LSB First

SPITX [3:0]				SPIRX [3:0]			
45h	A3h	B2h	DF	DFh	B2h	A3h	45h
C3h	8Ah	49h	24h	24h	49h	8Ah	C3h

9.14.1 Performing Variable-Bit Data Transmission

For a variable-bit data transmission in master mode (when the data is not transmitted in multiples of 8 bits), the most significant bit of the data to be transmitted must first be placed at position 7 of the SPIRXTX0, with the remaining bits positioned as shown in the SPI transaction figures on the previous page.

For example if SPISIZE = 0x0B and SPIREVERSE = 0, the data transaction will measure 12 bits, MSB first. For the transmission to occur in the correct order, the lower 4 data bits must first be placed into bit positions 7:4 of the SPIRXTX1 register, with bits 11:8 written into bit position 7:0 of the SPIRXTX0 register. This will trigger the transmission.

The following is a sequence of steps to transmit 12 bits of data contained in an integer variable called *txmitdata*.

1. Clear the SPIRXTX3 and SPIRXTX2 registers (optional)
2. Put the lower quartet of the 12-bit data (bits 3:0) into the upper quartet of the SPIRXTX1 register
3. Write bit 7:0 of the 12-bit data into the SPIRXTX0 register
4. This will trigger a data transmission

In C, this is expressed as follows:

```
(…)
   SPIRXTX3 = 0x00;
   SPIRXTX2 = 0x00;
   SPIRXTX1 = (txmitdata << 4)&0xF0;        //Write the lower quartet of data
                                            //into the upper quartet of SPIRXTX1 register

   readflag = SPIRXTX0          //-Dummy Read the SPI RX buffer to clear the RXAV Flag
                                //(Facultative if SPINOCS is monitored)

  SPIRXTX0 = dacdata >> 4;   //Writing to SPIRXTX0 will trigger the transmission
```

For example to output 0x3A2 through the SPI interface configured in master mode and MSB first format, write 0x20 into the SPIRXTX1 SFR register and followed by 0xA2 into the SPIRXTX0 register.

The reception of non multiple of 8 data when the SPI interface is configured to MSB first transaction is very straight forward as the data enters into the receiving buffer through the bit 0 of the SPIRXTX0 register and propagates towards the bit 7 of SPIRXTX3 register.

9.15 SPI Example Programs

9.15.1 UART to SPI Data Transmission Example

```
//-------------------------------------------------------------------------------------//
// SPI Transmit example.c                                                              //
//-------------------------------------------------------------------------------------//
// This program sends characters received  on the UART to the SPI Interface
//-------------------------------------------------------------------------------------//

#include <VRS51L3074_SDCC.h>

//----Global variables ------//

int  cptr = 0x00;       //general purpose counter

// --- function prototypes
void txmit0( unsigned char charact);
void uart0config(void);

//-------------------------------------------------------//
//                  Main Function                        //
//-------------------------------------------------------//

void main (void){

    char value = 0x00;              //general purpose variable
    PERIPHEN1 = 0xC0;               //Enable SPI Interface

    INTCONFIG = 0x02;       //Erase  Bypass global int, before configuring the INT0 pin
event
                                    //This fix inadvertent INT0 interrupt that occurs when
                                    //INT0 cause  is set to Rising edge

    INTSRC1 = 0x01;                 //INT0 vector source = INT0 pin
    INTPINSENS1 = 0x01;             //Set INT0 sensitive on edge(1) or Level(0)
    INTPININV1 = 0x00;              //Set INT0 Pin sensitivity on Normal Level(0) / Inverted (1)
    INTEN1 = 0x01;                  //Enable INT0 (bit0) Interrupt

    INTCONFIG = 0x01;               //Enable Global interrupt

    while(1);

}//end of Main

//-------------------------------------------------------------------------------------//
//---------------------------- Interrupt Functions ------------------------------------//
//-------------------------------------------------------------------------------------//

//-------------------------------------------------------------------------------------//
//      Interrupt INT0                                                                 //
//      Send character received on the SPI Interface                                   //
//-------------------------------------------------------------------------------------//

void INT0Interrupt(void) interrupt 0
   {
   //-- Send "EXT INT0 Received" on UART0
   cptr = 0x00;                     // Init cptr to pint to message beginning
   INTEN1 = 0x00;                   /Disable  Interrupts
   SPICTRL = 0xE1;                  //SPI CLK = div by 256
                                    //SPI CS0 Active
                                    //SPI Mode 0
                                    //SPI Master

   SPISIZE = 0x07;                  //SPI SIZE = 8bit
   SPICONFIG = 0x10;                //LOAD on CS3
   SPIRXTX0 = S0BUF;                //Send Data Byte on SPI Interface

   INTEN1 = 0x01;                   //Enable Interrupt INT0
   }//end of INT0 interrupt
```

9.16 SPI Interface to 12-Bit ADC and DAC

The following example program shows the initialization and use of the SPI module of the VRS51L3074 as an interface to serial ADC and DAC.

```
.//-------------------------------------------------------------------------------------------------------------//
// VRS51L3074_Generic_SPI_based_ADC_DAC_Interf1.c
//-------------------------------------------------------------------------------------------------------------//
// DESCRIPTION:
//              This Program demonstrates the configuration and use of the SPI interface
//              for interface to typical serial 12 bit A/D and D/A Converters.
//              The program read the A/D and output the read value out on a D/A converter
//              To perform the conversion the ADC requires 16 clock cycles and
//              the DAC requires 12 clock cycles.
//-------------------------------------------------------------------------------------------------------------//
#include <VRS51L3074_SDCC.h>

//---Functions prototypes
void ReadGEN_12BIT_ADC(void);                   //GEN_12BIT_ADC Read
void WriteGEN_12BIT_DAC(unsigned int );         //GEN_12_BIT_DAC Write

void V2KDelay1ms(unsigned int);                 //Standard Delay function

// Global variables definitions

idata unsigned char cptr = 0x00;
unsigned int at 0x0060 adcdata= 0x00;

//-------------------------------------------------------------------------------------//
//                      MAIN FUNCTION
//-------------------------------------------------------------------------------------//
void main (void) {

        do{
          ReadGEN_12BIT_ADC();              //Read the A/D Converter
          WriteGEN_12BIT_DAC(adcdata);      //write into the D/A Converter
        }while(1);

}// End of main

//-------------------------------------------------------------------------------------//
// NAME:                ReadGEN_12BIT_ADC
//-------------------------------------------------------------------------------------//
// DESCRIPTION:
//                      Read the GEN_12BIT_ADC A/D
//                    ADC is connected to SPI interface using CS0
//                    Max clk speed is 3.2MHz, Fosc = 40MHz assumed
//-------------------------------------------------------------------------------------//
void ReadGEN_12BIT_ADC()
    {
    int cptr = 0x00;
    char readflag = 0x00;

    //SPI Configuration Section
    //(Can be moved to Main function if only one device is connected to the SPI Interface)

    //Make sure the SPI Interface is activated
    PERIPHEN1 |= 0xC0;

    //--Wait activity stops on the SPI interface (Monitor SPINOCS)
    while(!(SPISTATUS &= 0x08));

    SPICTRL = 0x65;             //SPICLK = /16 (2.5MHz)
                                //CS0 Active
                                //SPI Mode 1 Phase = 1, POL = 0
                                //SPI Master Mode

    SPICONFIG = 0x40;           //SPI Chip select is automatic
                                //Clear SPIUNDEFC Flag
                                //SPILOAD = 0 -> Manual CS3 behaviour
                                //No SPI Interrupt used

    SPISTATUS = 0x00;           //SPI transactions are in MSB First Format

    SPISIZE = 0x0E;             //SPI Transaction Size are 15 bit

    //-Dummy Read the SPI RX buffer to clear the RXAV Flag
    readflag = SPIRXTX0;
    //-Perform the SPI read
     SPIRXTX0 = 0x00;                   //Writing to the SPIRXTX0 will trigger the SPI
                                        //Transaction

    while(!(SPISTATUS &= 0x02));        //Wait for the SPI RX AV Flag being set
    /*
    // -- It is possible to monitor the SPINOCS Flag instead of the SPIRXAV Flag
            //The code piece below shows how to do it.  However in that case,
            //No that the reading of the SPISTATUS register must be done at
            //least 4 System clock cycles after the Write operation to the SPIRXTX0 register

    //-Wait for SPINOCS Flag have time to be updated
    _asm
        NOP;
    _endasm;

    while(!(SPISTATUS &= 0x08));        //Wait activity stops on the SPI interface
    */
    //Read SPI data
    adcdata= (SPIRXTX1 << 8);
    adcdata+= SPIRXTX0;
    adcdata&= 0x0FFF;                   //isolate the 12 lsb of the read value
    }//end of ReadGEN_12BIT_ADC

//-------------------------------------------------------------------------------------//
// NAME:                WriteGEN_12BIT_DAC
//-------------------------------------------------------------------------------------//
// DESCRIPTION:
//                      Write 12bit Data into the GEN_12BIT_DAC device
//                      ADC is connected to SPI interface using CS1
//                      Max clk speed is 12.5MHz, Fosc = 40MHz assumed
//                      We will set the SPI prescaler to sysclk / 8
//-------------------------------------------------------------------------------------//
void WriteGEN_12BIT_DAC(unsigned int dacdata)
    {
    char subdata = 0x00;
    char readflag = 0x00;
    PERIPHEN1 |= 0xC0;              //Make sure the SPI Interface is activated

    //--Wait activity stops on the SPI interface (Monitor SPINOCS)
    while(!(SPISTATUS &= 0x08));

    //SPI Configuration Section
    //Can be moved to Main function if only one device is connected to the SPI Interface

    SPICTRL = 0x4D;                 //SPICLK = /8 (MHz)
                                    //CS1 Active
                                    //SPI Mode 1 Phase = 1, POL = 0
                                    //SPI Master Mode

    SPICONFIG = 0x40;               //SPI Chip select is automatic
                                    //Clear SPIUNDEFC Flag
                                    //SPILOAD = 0 -> Manual CS3 behaviour
                                    //No SPI Interrupt used

    SPISTATUS = 0x00;               //SPI transactions are in MSB First Format
    SPISIZE = 0x0B;                 //SPI Transaction Size are 12 bit

    //-Format the 12 bit data so data bit 11 is positioned on bit 7 of SPIRXTX0
    // and data bit 0 is positioned on bit 4 of SPIRXTX1 and Perform the SPI write operation

    dacdata &= 0x0FFF;              //Make sure dacdata is <= 0FFFh  (12 bit)

    SPIRXTX3 = 0x00;
    SPIRXTX2 = 0x00;
    SPIRXTX1 = (dacdata << 4)&0xF0;

    //-Dummy Read the SPI RX buffer to clear the RXAV Flag
    // (Facultative if SPINOCS is monitored)
    readflag = SPIRXTX0;

    SPIRXTX0 = dacdata >> 4;        //Writing to SPIRXTX0 will trigger the transmission

    //--Wait the SPI transaction completes
    //      This section can be omitted if a check of activity on the SPI Interface
    //      is made before each access to it in master mode

    //Wait for the SPI RX AV Flag being set
    while(!(SPISTATUS &= 0x02));

    // -- It is possible to monitor the SPINOCS Flag instead of the SPIRXAV Flag
            //The code piece below shows how to do it.  However in that case,
            //No that the reading of the SPISTATUS register must be done at
            //least 4 System clock cycles after the Write operation to the SPIRXTX0 register
    /*
    //-Wait for SPINOCS Flag have time to be updated
    _asm
        NOP;
    _endasm;
    //--Wait activity stops on the SPI interface  (monitor SPINOCS Flag)
    while(!(SPISTATUS &= 0x08));
    */
    }//end of WriteGEN_12BIT_DAC
```

10 I²C Interface

The VRS51L3074 includes an I²C interface that can operate in master and slave mode. In master mode, the communication speed on the I²C is programmable, optimizing communication between I²C-based devices. Long or heavily loaded I²C bus applications are likely to require slower communication speeds.

10.1 I²C Bus Pull-Up Resistors

By definition, the I²C requires that the user include external pull-up resistors on the SCL and SDA lines. The pull-up voltage can be either 3.3 or 5 volts. Note that the VRS51L3074 I/Os are 5V–tolerant making it possible to interface 5V, I²C-based devices with the VRS51L3074.

The proper value for the pull-up resistor and the proper communication speed depend on bus characteristics such as length and capacitive load.

Note that the pull-up resistor value should not be below 1.25K ohms if running the I²C bus at 5V; and 750 ohms if operating at 3.3V. This is required in order to limit the current to 4mA (maximum current of the I/O port connected to the I²C interface).

10.2 I²C Phases

The I²C protocol includes five phases:

1. IDLE (SCL = 1, SDA = 1)
2. Device ID
3. Device ID Acknowledge
4. Data
5. Data Acknowledge

The VRS51L3074 I²C interface has provisions to monitor activity on the I²C bus, particularly the data acknowledge phase of a I²C transaction. There is also a mechanism that enables the detection of communication errors.

10.3 I²C Control and Status Registers

Four SFR registers are dedicated to the I²C interface. The I²C configuration register I2CCONFIG enables:

- Selection of master or slave operation
- Forcing a start condition after an acknowledge phase
- Manual control of the SCL line
- Activation of the master arbitration monitoring mechanism
- Interrupt activation

TABLE 110:I2C CONFIGURATION REGISTER - I2CCONFIG SFR D1H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	1	0	0

Bit	Mnemonic	Description
7	MASTRARB	Master Lost Arbitration and Mechanism and Interrupt 0 = Deactivated 1 = Master lost arbitration monitoring and interrupt is enabled
6	I2CRXOVEN	I²C RX Overrun Interrupt Enable 0 = I²C RX Overrun interrupt is deactivated 1 = I²C RX Overrun interrupt is enabled
5	I2CRXAVEN	I²C RX Available Interrupt Enable 0 = I²C RX Available interrupt is deactivated 1 = I²C RX Available interrupt is enabled
4	I2CTXEEN	I²C TX Empty Interrupt Enable 0 = I²C TX empty interrupt is deactivated 1 = I²C TX empty interrupt is enabled
3	I2CMASTART	I²C Master Create Start 0 = No start condition is created after data acknowledge phase 1 = Master will create a start condition after the next data acknowledge phase This bit will be cleared when the I²C is idle
2	I2CSCLLOW	Keep the I²C SCL Low Setting this bit to 1 will force the SCL line low. This bit is read by the I²C interface when it enters in the data I²C. This bit must not be set during the acknowledge phase.
1	I2CRXSTOP	I²C Reception Stop 0 = The I²C received will acknowledge after receiving a byte 1 = The I²C receiver will not acknowledge after the next data byte is received
0	I2CMODE	I2C Mode Enable 0 = I²C interface operates in slave mode 1 = I²C Interface operates in master mode

The I2CMODE bit of the I2CCONFIG register, when set to 1, will configure the I²C interface as a master.

In master mode, the VRS51L3074 I²C interface controls the I²C bus and initiates transmission and reception transactions. In master mode, the I²C interface also controls the communication speed.

Clearing the I2CMODE bit of the I2CCONFIG register will configure the I²C interface as a slave. Slave mode can be useful for applications in which the

VRS51L3074 operates as a peripheral in a host-controlled system.

When in master mode, the I²C interface can be forced to generate a start condition after the next data acknowledge phase. This is done by setting the I2CMASTART bit to 1.

When the MASTRARB bit is set to 1, communications of the I²C will be monitored and an interrupt will be generated if arbitration with slave devices on the bus is lost. The interrupt flag associated with this process is the I2CERROR bit of the I2CSTATUS register.

If the I2CRXSTOP bit is set to 1, the I²C interface will not acknowledge after reception of the next byte, but will generate a stop condition instead. This will, in effect, end the transaction with the master device.

When the I²C interface is configured as a master and the I2CSCLLOW bit of the I2CCONFIG register is set to 1, the SCL line will be driven low during the next data acknowledge phase. This feature enables the user to add the equivalent of wait states to the transfer in order to support "slow" devices connected to the I²C bus.

The I²C interface includes support for four interrupt conditions via two interrupt vectors.

- RX Data Available
- RX Overrun
- TX Empty
- Master lost arbitration

The following table summarizes the possible interrupt sources at the I²C interface level.

TABLE 111: I²C INTERRUPT SOURCES

I²C Interrupt	I2CCONFIG bit (Set to 1 to activate)	Interrupt Vector
RX Data Available	I2CRXAVEN	4Bh (Int 9)
RX Overrun	I2CRXOVEN	0x4B (Int 9)
TX Empty	I2CTXEEN	0x4B (Int 9)
Master Lost Arbitration	MASTRARB	0x53 (Int 10)

To activate the I²C interface interrupts, the corresponding enable bit of the I2CCONFIG register must be set to 1. This will allow the I²C interrupt to propagate to the VRS51L3074's interrupt controller. In order for the I²C interrupt to be recognized by the processor, the corresponding bit of the INTEN2 and INTSRC2 registers must be configured accordingly. See the VRS51L3074 interrupt section for more details.

10.4 I²C Timing Control Register

The I2CTIMING register controls the communication speed when the I²C interface is configured in master mode. When in slave mode, it defines the values of the setup and hold times.

TABLE 112:I²C TIMING REGISTER - I2CTIMING SFR D2H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	1	1	0	0

Bit	Mnemonic	Description
7:0	I2CTIMING[7:0]	I²C master/slave timing configuration register See Below

The following formulas demonstrate the impact of the I2CTIMING value on the communication speed and setup/hold times.

In master mode:

$$\text{SCL period} = \frac{\text{I2CCLK}}{32*(\text{I2CTIMING[7:0]} + 1)}$$

The following table provides examples of the I2CTIMING values and the corresponding communication speed:

TABLE 113: I²C COMMUNICATION SPEED VS. I2CTIMING REGISTER VALUE (FOSC = 40MHZ)

I2CTIMING	I2C Com Speed
00h	1.25 MHz
02h	416.77 kHz
0Ch (Reset)	96.15 kHz
7Ch	10kHz
FFh	4.88kHz

In Slave Mode:

Set-up/Hold Time = I2CCLKperiod * I2CTIMING[7:0]

In this case, the precision is: 2 x I2CCLKperiod

TABLE 114: I²C SETUP AND HOLD TIME VS. I2CTIMING REGISTER VALUE (FOSC = 40MHZ)

I2CTIMING	Setup/Hold Time
00h	0 uS
0Ch	0.3 uS
FFh	6.38 uS

10.5 I²C Slave Device ID and Advanced Configuration

When operating in slave mode, the device ID on the I²C interface is configurable. The seven upper bits of the I2CIDCFG register contain the user-selected device ID. Bit 0 of the I2CIDCFG register has two distinct roles.

The I2CAVCFG provides advanced control on I²C interface operations.

TABLE 115:I²C DEVICE ID CONFIGURATION - I2CIDCFG SFR D3H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	I2CID[6:0]	Slave I²C device ID as selected by user
0	I2CADVCFG	Read: Indicates that the I²C slave has received ID that is different from the I2CID. This flag is cleared when the received ID corresponds with the I2CID
		Writing: Slave Mode: 1= The I2CRXAV flag is raised when the I²C slave receives a device ID Master Mode: 1 = Enables monitoring of the SCL line in wait state mode in case of mismatch of the SCL line vs. the expected value

When the I²C interface operates in master mode and the I2CADVCFG is cleared, the I²C interface module will continuously monitor the SCL line. If the slave device drives the SCL line into an incorrect state, the I²C interface will enter wait state mode until the slave device releases the SCL line. This mode can be useful for a I²C communication debug.

When the I2CADVCFG bit is set, no monitoring of the SCL line will be executed by the I²C module and the transaction will proceed independently of the level of the SCL line.

When the VRS51L3074 I²C interface module is configured as a slave, reading the I2CADVCFG bit as 1 indicates that the ID received does not match the current device ID. This bit will be cleared when the correct device ID is received.

In slave mode, writing a 1 into the I2CADVCFG bit of the I2CIDCFG register will make the I2CRXAVF flag of the I2CSTATUS register remain at 0, after the device ID is received. If the I2CADVCFG bit is cleared, the I2CRXAVF flag will be set either when a correct device ID, or when valid data, are received.

10.6 I²C Status Register

Monitoring of the I²C interface can be done via the I2CSTATUS register located at SFR address D4h. The I2CSTATUS register is read only and values written into that location have no effect.

The I2CERROR flag indicates that an error condition occurred on the I²C interface. In master mode, the I2CERROR flag will be set by the VRS51L3074 I²C interface, if it loses bus arbitration.

In slave mode, if an unexpected stop is received, the I2CERROR flag will be set. The I2CERROR flag will be automatically reset by the I²C interface the next time it exits an idle state.

If the I2CNOACK flag is set to 1, it signifies that the slave device did not acknowledge the last data byte it received.

The I²C interface also monitors the synchronization of the SDA line. When synchronization is lost, the I2CSDASYNC bit of the I2CSTATUS register will be set by the I²C interface.

The I2CSDASYNC bit of the I2CSTATUS register returns the value of the SDA line the moment a read operation is performed on the I2CSTATUS register.

The I2CACKPH bit when set, indicates that the I²C interface is currently in the data acknowledge phase.

Reading of the I2CSDASYNC and I2CCKPH bits can be used to determine whether the slave device has acknowledged. If both bits are set to 1 at a given time, the slave device did not acknowledge.

TABLE 116: I²C STATUS REGISTER - I2CSTATUS SFR D4H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	1	0	1	0	0	1

Bit	Mnemonic	Description
7	I2CERROR	Slave Mode Error Flag: 0 = No Error 1 = Indicates that the I²C interface received an unexpected stop This flag is reset the next time the I²C interface exits from an idle state (see below)
		Master Mode 0 = No Arbitration Error 1 = I²C interface has lost arbitration This flag is reset the next time the I²C interface exits from an idle state (see below)
6	I2CNOACK	I²C Acknowledge Error Flag 0 = Acknowledge was received normally 1 = No acknowledge was received during the last acknowledge phase This flag is reset the next time the I²C interface exit from the idle state (see below)
5	I2CSDASYNC	I²C SDA Sync Status Flag 0 = SDA Pin in not in sync 1 = SDA pin is in sync
4	I2CACKPH	When set, this flag indicates that the I²C interface is in 'Data Acknowledge Phase.' 5 phases of I²C protocol: 1. Idle 2. Device ID 3. Device ID Acknowledge 4. Data 5. Data Acknowledge
3	I2CIDLEF	I²C is idle 0 = I²C interface is communicating 1 = I²C interface is inactive (idle phase) and the SCL and SDA lines are high
2	I2CRXOVF	I²C RX Overrun Interrupt Flag 0 = No I²C RX overrun condition detected 1 = I²C data collision occurred
1	I2CRXAVF	I²C RX Available interrupt Flag 0 = I²C receive buffer is empty 1 = Data is present in the I²C RX buffer
0	I2CTXEMPF	I²C TX Empty interrupt Flag 0 = I²C transmit buffer is full 1 = I²C transmit buffer is ready to receive new data

When set, the I2CIDLEF indicates that the I²C bus is idle and that a transaction can be initiated. Before initiating an I²C data transfer, it is recommended to check the state of the I2CIDLEF bit. This bit indicates whether or not a data transfer is currently in progress.

When new data is received in the I²C receive buffer, the I2CRXAVF interrupt flag will be set. Data must be retrieved from the I2CRXTX buffer before the reception of the next data byte is complete.

The I2CRXOVF flag when set, indicates an overrun condition in the I²C interface receive buffer and the data is potentially corrupted.

The I2CTXEMPF interrupt flag is set by the I²C interface when the transmit data buffer is ready to receive another data byte.

10.7 I²C Transmit/Receive register

The I²C interface transmit and receive buffers are accessed via the I2CRXTX SFR register, which is accessible at SFR address D5h.

TABLE 117:I²C DATA RX / TX REGISTER I2CRXTX - SFR D5H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	I2CRXTX[7:0]	Read: I²C Receive Buffer Reading the I2CRXTX register will clear the I2CRXAV and I2CRXOV flags
		Write: I²C Transmit Buffer Writing into the I2CRXTX register will trigger the transmission

10.8 I²C Interface alternate pins

Upon reset, the I²C interface signal SCL and SDA are mapped into pins P3.4 and P3.5, respectively. However it is also possible to map these signal into the P1.6 and P1.7 pins.

Bit 5 of the DEVIOMAP register (SFR E1h) is used to configure the mapping of the I²C interface at the I/O level, as shown in the following table:

TABLE 118: I²C MODULE MAPPING

DEVIOMAP.5 Bit Value	SCL Mapping	SDA Mapping
0 (Reset)	P3.4	P3.5
1	P1.6	P1.7

10.9 I²C Interface Example Programs

The following programs provide example code for I²C control of EEPROM devices

```
//-------------------------------------------------------------------------------//
// VRS2k-I²C _EEPROM.c  //
//-------------------------------//
//
// This example program demonstrate the use of the I²C
// interface to perform basic read and write operations on a
// Standard EEPROM device.
//-------------------------------------------------------------------------------//

#include <VRS51L3074_SDCC.h>

//-----Global variables ------//
int  cptr = 0x00;       //general purpose counter

// --- Function prototypes
char EERandomRead(char,int);
char EERandomWrite(char, char, int);
void WaitTXEMP(void);
void WaitRXAV(void);
void WaitI2CIDLE(void);
void wait();


//-----------------------------------------------------------//
//---------------  MAIN FUNCTION ------------------//
//-----------------------------------------------------------//
void main (void){

    PERIPHEN1 = 0x20;       //Enable I²C  Interface

   INTCONFIG = 0x02;      //Erase Bypass global int, before configuring the INT0 pin event
                          //This fix inadvertent INT0 interrupt that occurs when
                          //INT0 cause  is set to Rising edge

    INTSRC1 = 0x01;           //INT0 vector source = INT0 pin
    INTPINSENS1 = 0x01;    //Set INT0 sensitive on edge(1) or Level(0)
    INTPININV1 = 0x00;     //Set INT0 Pin sensitivity on Normal Level(0) / Inverted (1)
    INTEN1 = 0x01;            //Enable INT0 (bit0) Interrupt

    INTCONFIG = 0x01;      //Enable Global interrupt

    while(1);
}//end of Main

//-------------------------------------------------------------------------------//
//---------------------- Interrupt Functions ------------------------//
//-------------------------------------------------------------------------------//

//-----------------------------//
//---- Interrupt INT0 ----//
//-----------------------------//
void INT0Interrupt(void) interrupt 0
   {
   char x;

   //-- Send I²C  stuff
   cptr = 0x00;           // Init cptr to pint to message beginning
   INTEN1 = 0x00;        //Disable  Interrupts

   x =  EERandomWrite(0xA0, 0x36, 0x0206);       //Perform Write operation
   Delay1ms(100);
   x = EERandomRead( 0xA0, 0x0206);              //Perform Read operation

   INTEN1 = 0x01;        //Enable Interrupt INT0
   }//end of INT0 interrupt

//-------------------------------------------------------------------------------//
//---------------------- Individual Functions ---------------------//
//-------------------------------------------------------------------------------//

//-------------------------------------------------------------------------------//
//----  Function EERandomRead(char eeidw,int address) -----//
//-------------------------------------------------------------------------------//
char EERandomRead(char eeidw,int address){
   I2CTIMING = 0x20;             // I²C  Clock Speed = about 100kHz
   I2CCONFIG = 0x01;             //I²C  is Master
   I2CRXTX = eeidw;              //Write I²C  device ID + W
   WaitTXEMP();
   I2CRXTX = address >> 8;     //Write I²C  ADRSH
   WaitTXEMP();
   I2CRXTX = address;            //Write I²C  ADRSL

   //--Wait for I²C IDLE  (This will generate a STOP)
   WaitI2CIDLE();

   //--Start a Preset ADRS read (This will generate a START)
   I2CRXTX = eeidw+1;            //Write I²C  device ID + R
   WaitTXEMP();
   I2CCONFIG |= 0x02;          //Force I²C  to Not Acknowledge after
                               //receiving the next data byte
   WaitRXAV();                 //Wait for RX Available bit, This will trigger I²C  Reception
   return I2CRXTX;             //Return Data Byte

   }//End of EERandomRead

//-------------------------------------------------------------------------------//
//-----  Function EERandomWrite(char eeid,char data, int address) ----------//
//-------------------------------------------------------------------------------//
char EERandomWrite(char eeidw, char eedata, int address){
   I2CTIMING = 0x20;          // I²C  Clock Speed = about 100kHz
   I2CCONFIG = 0x01;          //I²C  is Master
   I2CRXTX = eeidw;           //Write I²C  device ID + W

   WaitTXEMP();

   I2CRXTX = address >> 8; //Write I²C  device ID + W
   WaitTXEMP();

   I2CRXTX = address;         //Write I²C  device ID + W
   WaitTXEMP();

   I2CRXTX = eedata;          //Write I²C  device data
   WaitTXEMP();

   return I2CRXTX;            //Return Data Byte

   }//End of EERandomWrite

//-----------------------------------------------//
//--------  Function WaitTXEMP() ----------//
//-----------------------------------------------//
void WaitTXEMP()
   {
   wait();
   do{

   USERFLAGS = I2CSTATUS;
   USERFLAGS &= 0x01;          //isolate the I²C  TX EMPTY flag

   }while( USERFLAGS == 0x00);    //Wait for I²C  TX EMPTY
   }//end of Void WaitTXEMP()

//-----------------------------------------------//
//--------  Function WaitRXAV() ------------//
//-----------------------------------------------//
void WaitRXAV()
   {
   wait();
   do{

   USERFLAGS = I2CSTATUS;
   USERFLAGS &= 0x02;          //isolate the I2CRXAV flag

   }while( USERFLAGS == 0x00);    //Wait for I²C  RX AVAILABLE

   }//end of Void WaitRXAV()

//-----------------------------------------------//
//--------  Function WaitI2CIDLE() ---------//
//-----------------------------------------------//
void WaitI2CIDLE()
   {
   wait();
    do{
   USERFLAGS = I2CSTATUS;
   USERFLAGS &= 0x08;          //isolate the I²C  idle flag
   }while( USERFLAGS == 0x00);

   }//end of Void WaitI2CIDLE()

//-----------------------------------------//
//--------  Function Wait() ----------//
//-----------------------------------------//
void wait(){
   char i=0;
   while (i<25) {i++;};
   }
```

11 Pulse Width Modulators (PWMs)

The VRS51L3074 includes eight independent PWM channels, each based on a 16-bit timer.

All of the PWM modules can be configured to operate as a regular PWM with adjustable resolution, or as a general purpose 16-bit timer. The PWMEN register is used to enable the different PWM modules.

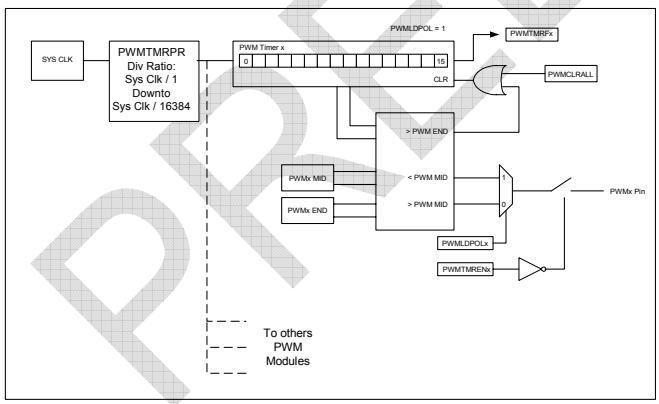
TABLE 119: PWM ENABLE REGISTER - PWMEN SFR AAH

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	PWM7EN	PWM7 Channel Enable 0 = PWM channel 7 is deactivated 1 = PWM channel 7 is activated
6	PWM6EN	PWM6 Channel Enable 0 = PWM channel 6 is deactivated 1 = PWM channel 6 is activated
5	PWM5EN	PWM5 Channel Enable 0 = PWM channel 5 is deactivated 1 = PWM channel 5 is activated
4	PWM4EN	PWM4 Channel Enable 0 = PWM channel 4 is deactivated 1 = PWM channel 4 is activated
3	PWM3EN	PWM3 Channel Enable 0 = PWM channel 3 is deactivated 1 = PWM channel 3 is activated
2	PWM2EN	PWM2 Channel Enable 0 = PWM channel 2 is deactivated 1 = PWM channel 2 is activated
1	PWM1EN	PWM1 Channel Enable 0 = PWM channel 1 is deactivated 1 = PWM channel 1 is activated
0	PWM0EN	PWM0 Channel Enable 0 = PWM channel 0 is deactivated 1 = PWM channel 0 is activated

The following figure provides an overview of the PWM modules.

FIGURE 26: PWM MODULES OVERVIEW



11.1 PWM MID and END registers

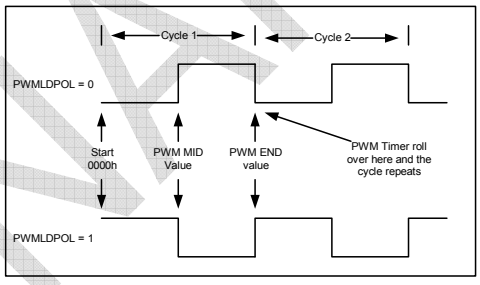
Each PWM module includes two 16-bit registers:

- o PWM MID value register
- o PWM END value register

The PWM MID register is a 16-bit register that configures the point at which the PWM output will change it's polarity.

The PWM END register is a 16-bit register that defines the maximum PWM internal timer count value, after which it rolls over to 0000h. See the following timing diagram.

FIGURE 27: PWM POLARITY SETTING



This configuration allows the user to adjust the resolution of the PWM up to 16 bits. Access to the PWM internal registers and the PWM configuration is handled by the PWMCFG register located at address A9h.

TABLE 120:PWM CONFIGURATION REGISTER - PWMCFG SFR A9H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	-	
6	PWMWAIT	PWM Waits Before Loading New Configuration 0 = New PWM configuration is loaded at the end of PWM cycle 1 = The update of the PWM configuration only occurs when the end of the PWM is reached and the bit is set to 0
5	PWMCLRALL	PWM Clears All Channels 0 = No Action 1 = Simultaneously clears all the flags and all the PWM channel timers This bit is automatically cleared by hardware
4	PWMLSBMSB	PWM LSB/MSB Select 0 = Selected PWM LSB SFR is addressed 1 = Selected PWM MSB SFR is addressed
3	PWMMIDEND	PWM MID/END Register 0 = Selected PWM MID SFR is addressed 1 = Selected PWM END SFR is addressed
2:0	PWMCH[2:0]	PWM Channel Select 000 = PWM0 on P2.0 (P5.0) 001 = PWM1 on P2.1 (P5.1) 010 = PWM2 on P2.2 (P5.2) 011 = PWM3 on P2.3 (P5.3) 100 = PWM4 on P2.4 (P5.4) 101 = PWM5 on P2.5 (P5.5) 110 = PWM6 on P2.6 (P5.6) 111 = PWM7 on P2.7 (P5.7)

The PWM channels are configured one at the time. This topology has been adopted in order to minimize the number of SFR registers required to access the PWM modules.

In applications where multiple PWM channels need to be configured simultaneously, the user can set the PWMWAIT bit of the PWMCFG register, configure each one of the PWM channels, and then clear the PWMWAIT bit. The PWM configurations will then be updated at the end of the next PWM cycle, after the PWMWAIT bit has been cleared.

TABLE 121:PWM DATA REGISTER SFR ACH

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	PWMDATA[7:0]	PWM Data Register

The PWM data register serves to configure the selected channel MSB/LSB value of either the MID or END point, as specified in the PWMCFG register.

The PWMIDx defines the actual timer value and the PWMEND defines the maximum timer count value before it rolls over.

The PWMLDPOL register controls the output polarity of each one of the PWM modules or clears the timer's value when the PWM modules operate as general purpose timers.

TABLE 122:PWM POLARITY AND CONFIG LOAD STATUS – PWMLDPOL ABH

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	PWMLDPOL7	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded
		Write **In PWM Mode** 0 = PWM 7 cycle starts with a low level 1 = PWM 7 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 7 value is cleared to 0
6	PWMLDPOL6	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded
		Write **In PWM Mode** 0 = PWM 6 cycle starts with a low level 1 = PWM 6 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 6 value is cleared to 0
5	PWMLDPOL5	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded
		Write **In PWM Mode** 0 = PWM 5 cycle starts with a low level 1 = PWM 5 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 5 value is cleared to 0
4	PWMLDPOL4	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded
		Write **In PWM Mode** 0 = PWM 4 cycle starts with a low level 1 = PWM 4 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 4 value is cleared to 0
3	PWMLDPOL3	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded
		Write **In PWM Mode** 0 = PWM 3 cycle starts with a low level 1 = PWM 3 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 3 value is cleared to 0
2	PWMLDPOL2	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded
		Write **In PWM Mode** 0 = PWM 2 cycle starts with a low level 1 = PWM 2 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 2 value is cleared to 0

1	PWMLDPOL1	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded **Write** **In PWM Mode** 0 = PWM 1 cycle starts with a low level 1 = PWM 1 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 1 value is cleared to 0
0	PWMLDPOL0	**Read:** 0 = Last configuration has been loaded in PWM 1 = Last configuration has not been loaded **Write** **In PWM Mode** 0 = PWM 0 cycle starts with a low level 1 = PWM 0 cycle starts with a high level **In Timer Mode** 0 = No action 1 = PWM timer 0 value is cleared to 0

11.2 PWM Module Clock Configuration Register

One system clock prescaler is associated with PWM modules 0 to 3, while another is associated with PWM modules 4 to 7. The PWM clock prescalers enables the PWM output frequency to be adjusted to match specific application needs, if required. The PWM clock prescalers are configured via the PWMCLKCFG register. The four upper bits of this register control the clock for PMM modules 4 to 7, and the four lower bits control the clock source for PWM modules 0 to 3.

The PWM module clock configuration register controls the prescale value applied to the PWM modules' input clock, when the PWM modules are configured to operate as either PWMs or general purpose timers.

TABLE 123: PWM CLOCK PRESCALER CONFIGURATION REGISTER - PWMCLKCFG AFH

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:4	U4PWMCLK3[3:0]	PWM Timer 7, 6, 5,:4 Clock Prescaler * see table below
3:0	L4PWMCLK3[3:0]	PWM Timer 3, 2, 1,:0 Clock Prescaler * see table below

The following table shows the system clock division factor applied to the PWM modules for a given PWMCLKCFG nibble.

TABLE 124: PWM PRESCALER VALUES

U4/L4PWMCLK Value (4 bit)	Clock Prescaler	U4/L4PWMCLK Value (4 bit)	Clock Prescaler
0000	Sys Clk / 1	1000	Sys Clk / 256
0001	Sys Clk / 2	1001	Sys Clk / 512
0010	Sys Clk / 4	1010	Sys Clk / 1024
0011	Sys Clk / 8	1011	Sys Clk / 2048
0100	Sys Clk / 16	1100	Sys Clk / 4096
0101	Sys Clk / 32	1101	Sys Clk / 8192
0110	Sys Clk / 64	1110	Sys Clk/ 16384
0111	Sys Clk / 128	1111	Sys Clk/ 16384

11.3 PWM Alternate Mapping

Bit 6 of the DEVIOMAP register (SFR E1h) controls the mapping of the PWM module outputs, as shown in the following table:

TABLE 125: PWM MODULES OUTPUT MAPPING

DEVIOMAP.6 Bit Value	PWM 7-0
0 (Reset)	P2.7 – P2.0
1	P5.7 – P5.0

Note that the PWM5 and PWM6 outputs have priority over the T0EX and T1EX inputs.

11.4 PWM Examples Program

11.4.1 PWM Basic Configuration

The following example program shows the basic configuration of PWM modules #0, 1,2, 4 & 5

```
//-------------------------------------------------------------------------------------------------------//
// VRS51L3074-PWM_basic_SDCC.c  //
////////////////////////////
//
// DESCRIPTION:    VRS51L3074 PWMs Basic initialization Demonstration Program.
//                 Configure PWM0 as 8 bit resolution  (25% duty)
//                 Configure PWM1 as 12 bit resolution  (50% duty)
//                 Configure PWM2 as 16 bit resolution   (75% duty)
//                 Configure PWM4 as 8 bit resolution and prescaler = 4 (25% duty)
//                 Configure PWM5 as 16 bit resolution and prescaler = 4 (75% duty)
//-------------------------------------------------------------------------------------------------------//
//   Rev   1.0
//   Date:    June 2005
//-------------------------------------------------------------------------------//

#include <VRS51L3074_SDCC.h>

// --- function prototypes

void delay(unsigned int);

void main (void) {
    PERIPHEN2 = 0x02;               //Enable PWM SFR

    //CLEAR All PWM Channels
    PWMCFG = 0x20;

    // Configure the PWM prescaler
    PWMCLKCFG = 0x20;               // Apply a clock prescaler (div / 4) on PWM 7:4

    // Configure PWM Polarity
    PWMPOL = 0x00;                  //Set all PWM in normal polarity
                                    //PWM output = 0 until
                                    //PWMMID Value is reached

    //---------------------------------//
    //Configure PWM0 END value = 0x00FF (8bit)
    PWMCFG = 0x58;                  //Point to PWM0 END MSB
    PWMDATA = 0x00;                 //Set Max Count MSB = 0xFF
    PWMCFG = 0x48;                  //Point to PWM0 END LSB
    PWMDATA = 0xFF;                 //Set PWM MID MSB = 0x00 (8bit)


    //Configure PWM0 MID value (Duty = 25%)
    PWMCFG = 0x50;                  //Point to PWM0 MID MSB
    PWMDATA = 0x00;                 //Set PWM MID MSB = 0x00
    PWMCFG = 0x40;                  //Point to PWM0 MID LSB
    PWMDATA = 0xBF;                 //Set PWM MID LSB = 0xBF

    //---------------------------------//
    //Configure PWM1 END value = 0x0FFF (12bit)
    PWMCFG = 0x59;                  //Point to PWM1 END MSB
    PWMDATA = 0x0F;                 //Set Max Count MSB = 0x0F
    PWMCFG = 0x49;                  //Point to PWM1 END LSB
    PWMDATA = 0xFF;                 //Set Max Count = 0xFF

    //Configure PWM1 MID value (Duty = 50%)
    PWMCFG = 0x51;                  //Point to PWM0 MID MSB
    PWMDATA = 0x08;                 //Set PWM MID MSB = 0x08
    PWMCFG = 0x41;                  //Point to PWM0 MID LSB
    PWMDATA = 0x00;                 //Set PWM MID LSB = 0x00

    //---------------------------------//
    //Configure PWM2 END value = 0xFFFF (16bit)
    PWMCFG = 0x5A;                  //Point to PWM2 END MSB
    PWMDATA = 0xFF;                 //Set Max Count MSB = 0xFF
    PWMCFG = 0x4A;                  //Point to PWM2 END LSB
    PWMDATA = 0xFF;                 //Set Max Count = 0xFF

    //Configure PWM2 MID value (duty = 75%)
    PWMCFG = 0x52;                  //Point to PWM2 MID MSB
    PWMDATA = 0x40;                 //Set PWM MID MSB = 0x04
    PWMCFG = 0x42;                  //Point to PWM2 MID LSB
    PWMDATA = 0x00;                 //Set PWM MID LSB = 0x00


    //---------------------------------//
    //Configure PWM4 END value = 0x00FF (8 bit) (Clock Prescaler = 4)
    PWMCFG = 0x5C;                  //Point to PWM4 END MSB
    PWMDATA = 0x00;                 //Set Max Count MSB = 0xFF
    PWMCFG = 0x4C;                  //Point to PWM4 END LSB
    PWMDATA = 0xFF;                 //Set Max Count LSB = 0xFF

    //Configure PWM4 MID value (duty = 25%)
    PWMCFG = 0x54;                  //Point to PWM4 MID MSB
    PWMDATA = 0x00;                 //Set PWM MID MSB = 0x00
    PWMCFG = 0x44;                  //Point to PWM4 MID LSB
    PWMDATA = 0xBF;                 //Set PWM MID LSB = 0xBF

    //---------------------------------//
    //Configure PWM5 END value = 0xFFFF (16bit)   (Clock Prescaler = 4)
    PWMCFG = 0x5D;                  //Point to PWM5 END MSB
    PWMDATA = 0xFF;                 //Set Max Count MSB = 0xFF
    PWMCFG = 0x4D;                  //Point to PWM5 END LSB
    PWMDATA = 0xFF;                 //Set Max Count = 0xFF

    //Configure PWM5 MID value (duty = 75%)
    PWMCFG = 0x55;                  //Point to PWM5 MID MSB
    PWMDATA = 0x40;                 //Set PWM MID MSB = 0x04
    PWMCFG = 0x45;                  //Point to PWM5 MID LSB
    PWMDATA = 0x00;                 //Set PWM MID LSB = 0x00

    //Enable PWM0, PWM1, PWM2, PWM4 & PWM5 Modules
    PWMEN = 0x37;


    PWMCFG &= 0x1F;                 //Clear the PWMWAIT bit to initiate
                                    //the PWMs operation
    while(1);

}// End of main
```

11.4.2 PWM Configuration and Control Functions

```
//-------------------------------------------------------------------------------------------------------//
// VRS51L3074-PWM_CFG_function_SDCC.c                                                                    //
//-------------------------------------------------------------------------------------------------------//
// DESCRIPTION:    PWM configuration and control Functions
//-------------------------------------------------------------------------------------------------------//
#include <VRS51L3074_SDCC.h>

// --- functions prototypes
void PWMConfig(char channel,int endval,int midval);
void PWMdata8bit(char,char);
void PWMdata16bit(char,int);
void delay(unsigned int);

void delay(unsigned int);

void main (void) {
     int cptr = 0x00;

  //  PERIPHEN2 = 0x02;             //Enable PWM SFR

    //CLEAR All PWM Channels
    PWMCFG = 0x20;

    // Configure the PWM prescaler
    PWMCLKCFG = 0x00;               // Apply a clock prescaler (div / 1) on all PWM

    // Configure PWM Polarity
    PWMLDPOL = 0x00;                //Set all PWM in normal polarity
                                    //PWM output = 0 until

    //--Configure PWM5 as 8bit resolution,  END = 0xFF, PWM MID = 0x000
   PWMConfig(0x05, 0x0FF,0x000);

   //--Configure PWM0 as 8bit resolution,  END = 0xFFF, PWM MID = 0x0000
    PWMConfig(0x02, 0xFFF,0x000);

    //Continuously vary the PWM2 and PWM5 values

    do{
        for(cptr = 0xFF0; cptr > 0x00; cptr--)
            {
            PWMdata16bit(0x02,cptr);
            PWMdata8bit(0x05,cptr>>4);
            delay(1);
            }
    }while(1);
}// End of main
```

```
//-------------------------------------------------------------------------
// ---------      Individual Functions      -------------------------
//-------------------------------------------------------------------------

//-------------------------------------------------------------------------------//
// -- PWMConfig                                                                  //
// -------------------------------------------------------------------------------//
// Description: configure PWM channel                                           //
//-------------------------------------------------------------------------------//
void PWMConfig(char channel,int endval,int midval)
    {
    char pwmch;
    char pwmready = 0x00;

    channel &= 0x07;                  //Make sure PWM ch number <= 7

    //Wait Last configuration to be loaded
    do{
        pwmready = PWMLDPOL;
        }while(pwmready != 0x00);

    //Define PWM Enable section

    PERIPHEN2 |= 0x02;                //Enable PWM SFR
    //--Define the value to put into the PWMEN register
    switch(channel)
        {
        case 0x00 : pwmch = 0x01;
                break;
        case 0x01 : pwmch = 0x02;
                break;
        case 0x02 : pwmch = 0x04;
                break;
        case 0x03 : pwmch = 0x08;
                break;
        case 0x04 : pwmch = 0x10;
                break;
        case 0x05 : pwmch = 0x20;
                break;
        case 0x06 : pwmch = 0x40;
                break;
        case 0x07 : pwmch = 0x80;
                break;
        }//end of switch

     PWMEN |= pwmch;                    //Enable the Selected channel

    //Configure PWM END point

    PWMCFG = (channel + 0x58);          //Set PWM configuration register to point to
                                        //the MSB of End value and set the PWMWAIT bit
                                        //to prevent the PWM configuration to be loaded
                                        //before the configure sequence is completed
    PWMDATA = endval >> 8;

    PWMCFG &= 0xEF;                     //Set PWM configuration register to point to
                                        //the LSB of End value
    PWMDATA = endval;

    //Configure PWM MID point

    PWMCFG = (channel + 0x50);          //Set PWM configuration register to point to
                                        //the MSB of MID value and set the PWMWAIT bit
                                        //to prevent the PWM configuration to be loaded
                                        //before the configure sequence is completed
    PWMDATA = midval >> 8;

    PWMCFG &= 0xEF;                     //Set PWM configuration register to point to
                                        //the LSB of End value
    PWMDATA = midval;

    PWMCFG &= 0x3F;                     //Allows PWM update upon end of next PWM cycle

    }//end of PWMData16bit()
```

```
//-------------------------------------------------------------------------------//
// -- PWMdata8bit                                                                //
// -------------------------------------------------------------------------------//
// Description: Allow PWM channel data update                                   //
//                  ( 8bit data )l                                              //
//-------------------------------------------------------------------------------//
void PWMdata8bit(char channel,char pwmdata)
    {
    channel &= 0x07;                    //Make sure PWM ch number <= 7

    //--check that te last configuration has been loaded


    PWMCFG = (channel + 0x40);          //Write new value in PWM Config
                                        //prevent PWM configuration to be loaded
                                        //before the configure sequence is completed
    PWMDATA = pwmdata;                  //Write new Data into the PWM registers

    PWMCFG &= 0x3F;                     //Allows PWM update upon end of next PWM cycle

    }//end of PWMData8bit()


//-------------------------------------------------------------------------------//
// -- PWMdata16bit                                                               //
// -------------------------------------------------------------------------------//
// Description: Allow PWM channel data update                                   //
//                  ( 16bit data )l                                             //
//-------------------------------------------------------------------------------//
void PWMdata16bit(char channel,int pwmdata)
    {


    channel &= 0x07;                    //Make sure PWM ch number <= 7
    PWMCFG = (channel + 0x50);          //Set PWM configuration register to point to
                                        //the MSB of Data value and set the PWMWAIT bit
                                        //and set the PWMWAIT bit to prevent the
                                        //PWM configuration to be loaded
                                        //before the configure sequence is completed
    PWMDATA = pwmdata >>8;

    PWMCFG &= 0xEF;                     //Set PWM configuration register to point to
                                        //the LSB of Data value
    PWMDATA = pwmdata;

    PWMCFG &= 0x3F;                     //Allows PWM update upon end of next PWM cycle

    }//end of PWMData16bit()


//;-------------------------------------------------------------------------------//
//;- DELAY1MSTO : 1MS DELAY USING TIMER0                                         //
//;                                                                             //
//; CALIBRATED FOR 40MHZ                                                       //
//;-------------------------------------------------------------------------------//
void delay(unsigned int dlais){

idata unsigned char x=0;
idata unsigned int dlaisloop;

    x = PERIPHEN1;                  //LOAD PERIPHEN1 REG
    x |= 0x01;                      //ENABLE TIMER 0
    PERIPHEN1 = x;

    dlaisloop = dlais;
     while ( dlaisloop > 0)
       {
        TH0 = 0x63;                 //TIMER0 RELOAD VALUE FOR 1MS AT 40MHZ
       TL0 = 0xC0;

       T0T1CLKCFG = 0x00;           //NO PRESCALER FOR TIMER 0 CLOCK
       T0CON = 0x04;                //START TIMER 0,  COUNT UP

         do{
             x=T0CON;
             x= x & 0x80;
             }while(x==0);

        T0CON = 0x00;               //Stop Timer 0
        dlaisloop = dlaisloop-1;
        }//end of while dlais...

       x = PERIPHEN1;               //LOAD PERIPHEN1 REG
       x = x & 0xFE;                //DISABLEBLE TIMER 0
       PERIPHEN1 = x;
    }//End of function delais
```

11.5 Using PWM Modules as Timers

By appropriately configuring the PWMTMREN SFR, the PWM modules can also operate as general purpose 16-bit timers. The following table describes the PWMTMREN register:

TABLE 126: PWM TIMER MODE ENABLE REGISTER - PWMTMREN SFR ADH

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	PWM7TMREN	PWM 7 Module Operating Mode 0 = PWM 7 module is configured as PWM 1 = PWM 7 module is configured as timer
6	PWM6TMREN	PWM 6 Module Operating Mode 0 = PWM 6 module is configured as PWM 1 = PWM 6 module is configured as timer
5	PWM5TMREN	PWM 5 Module Operating Mode 0 = PWM 5 module is configured as PWM 1 = PWM 5 module is configured as timer
4	PWM4TMREN	PWM 4 Module Operating Mode 0 = PWM 4 module is configured as PWM 1 = PWM 4 module is configured as timer
3	PWM3TMREN	PWM 3 Module Operating Mode 0 = PWM 3 module is configured as PWM 1 = PWM 3 module is configured as timer
2	PWM2TMREN	PWM 2 Module Operating Mode 0 = PWM 2 module is configured as PWM 1 = PWM 2 module is configured as timer
1	PWM1TMREN	PWM 1 Module Operating Mode 0 = PWM 1 module is configured as PWM 1 = PWM 1 module is configured as timer
0	PWM0TMREN	PWM 0 Module Operating Mode 0 = PWM 0 module is configured as PWM 1 = PWM 0 module is configured as timer

When operating in timer mode, the PWM module timer will count from 0000h up to the maximum PWM timer value defined by the PWM MID sub registers, which are accessible through the PWMCFG register.

TABLE 127: SUMMARY OF PWM MID SUB REGISTERS ACCESS

	PWMCFG bit PWMLSBMSB	PWMCFG bit PWMMIDEND
PWM timer MSB max count value	0	1
PWM timer MSB max count value	1	1

Once the PWM MID value is reached, the PWM timer overflow is set and the PWM timer rolls over to 0000h.

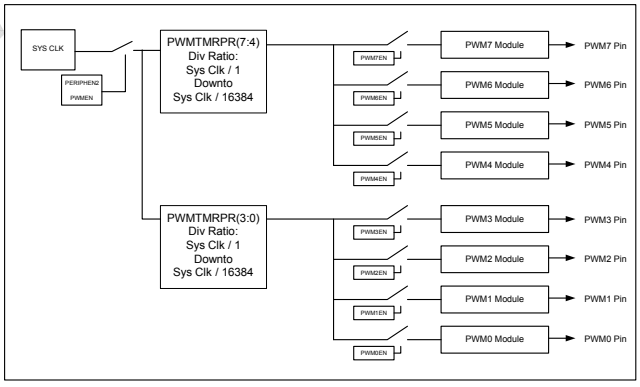
The PWM timer flags are raised when the timer reaches the maximum value set by PWMMIDH and PWMMIDL. The PWMxTMRF bit must be cleared manually by the interrupt service routine.

TABLE 128: PWM TIMER FLAGS REGISTER - PWMTMRF SFR AEH

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	PWM7TMRF	PWM 7 Module Timer Flag 0 = No Overflow 1 = PWM Timer 7 Overflow
6	PWM6TMRF	PWM 6 Module Timer Flag 0 = No overflow 1 = PWM Timer 6 Overflow
5	PWM5TMRF	PWM 5 Module Timer Flag 0 = No Overflow 1 = PWM Timer 5 Overflow
4	PWM4TMRF	PWM 4 Module Timer Flag 0 = No Overflow 1 = PWM Timer 4 Overflow
3	PWM3TMRF	PWM 3 Module Timer Flag 0 = No Overflow 1 = PWM Timer 3 Overflow
2	PWM2TMRF	PWM 2 Module Timer Flag 0 = No Overflow 1 = PWM Timer 2 Overflow
1	PWM1TMRF	PWM 1 Module Timer Flag 0 = No Overflow 1 = PWM Timer 1 Overflow
0	PWM0TMRF	PWM 0 Module Timer Flag 0 = No Overflow 1 = PWM Timer 0 Overflow

FIGURE 28: PWM AS TIMERS OVERVIEW



11.6 Configuring the PWM Timers

Configuring the PWM modules to operate in PWM timer mode requires the following steps:

1. Activate the PWMSFR register
2. Configure the PWM clock prescaler (if required)
3. Set the PWMLDPOL register to 00h
4. Configure the PWM timer maximum count value by setting the PWM MID sub-registers
5. Configure the PWM timer interrupts (if required)
6. Configure the PWM modules as timers
7. Enable the PWM modules

Follow the code example below to perform these seven steps :

```
(…)
    PERIPHEN2 |= 0x02;              //Enable PWM SFR

    //--Configure the PWM prescaler
    PWMCLKCFG = 0x03;               //Apply a clock prescaler (div / 8) on PWM 3:0

    //--Configure PWM Polarity
    PWMLDPOL = 0x00;                //Set all PWM in normal polarity
                                    //PWM output = 0 until

    //--Configure PWM5 as timer
    //  PWM Timer 5 counts from 0000 to F000h
    PWMCFG =  0x15;                 //Point to MSB MID
    PWMDATA = 0xF0;                 //Set PWM as Timer Max MSB

    PWMCFG =  0x05;                 //Point to LSB MID
    PWMDATA = 0x00;                 //Set PWM as Timer Max LSB

    //--Configure and Enable PWM as timer Interrupt  to monitor PWM5 only
    INTSRC2 &= 0xDF;                //PWM7:4 Timer module interrupt
    INTPINSENS1 = 0xDF;             // sensitive on high level(0)
    INTPININV1 = 0xDF;              //Set INT0 Pin sensitivity on normal level(0)
    INTEN2 |= 0x20;                 //Enable PWM7:4 Timer module interrupt


    //--Activate the PWM module and cofigure the PWM modules 5 as timer
    PWMEN |= 0x20;                  //Enable PWM 5
    PWMTMREN |= 0x20;               //Enable PWM 5 as Timer

    GENINTEN = 0x03;                //Enable Global interrupt
```

11.7 PWMs as Timers Example Programs

11.7.1 Configuring PWM0 and PWM5 as Timers

The following example program demonstrates how to initialize PWM0 and PWM5 as general purpose timers, and how to monitor the PWM timer's overflow flags by pooling or via an interrupt.

```
//-------------------------------------------------------------------------------------------------------//
// VRS51L3074-PWM_as_Timer1_SDCC.c.c                                                                     //
//-------------------------------------------------------------------------------------------------------//
// DESCRIPTION:  PWM as Timer Example Program
//               Enable and configure PWM Timer 0
//               Apply a clock prescaler on PWM Timer 0 (div/8)
//               Enable and configure PWM Timer 5
//               Monitor PWM Timer 0 OV Flag by pooling
//               When PWM Timer 0 Overflow, toggle P1.0 pin
//               Monitor PWM Timer 5 OV Flag by interrupt
//               When PWM Timer 5 Overflow interrupt occurs toggle P1.5 pin
//-------------------------------------------------------------------------------------------------------//
#include <VRS51L3074_SDCC.h>
void main (void) {
     int cptr = 0x00;
     char flagread;

    PERIPHEN2 |= 0x02;              //Enable PWM SFR

    //Configure Port1 as output

    P1PINCFG = 0x00;
    //Clear All PWM Channels
   // PWMCFG = 0x20;

    // Configure the PWM prescaler
    PWMCLKCFG = 0x03;               // Apply a clock prescaler (div / 8) on PWM 3:0

    // Configure PWM Polarity
    PWMLDPOL = 0x00;                //Set all PWM in normal polarity
                                    //PWM output = 0 until

    //--Configure PWM0 as Timer  (will be monitored by pooling)
    //  PWM Timer 0 counts from 0000 to 01F0h

    PWMCFG =  0x10;                 //Point to MSB MID
    PWMDATA = 0x01;

    PWMCFG =  0x00;                 //Point to LSB MID
    PWMDATA = 0xF0;

    //--Activate the PWM modules and configure the PWM modules as timers
    PWMEN |= 0x01;
    PWMTMREN |= 0x01;               //Enable PWM 0 as Timer

    //--Configure PWM5 as Timer  (will be monitored by interrupt)
    //  PWM Timer 5 counts from 0000 to F000h
    PWMCFG =  0x15;                 //Point to MSB MID
    PWMDATA = 0xF0;                 //

    PWMCFG =  0x05;                 //Point to LSB MID
    PWMDATA = 0x00;

    //--Configure and enable PWM as timer interrupt  to monitor PWM5 only
    INTSRC2 &= 0xDF;                //PWM7:4 Timer module interrupt
    INTPINSENS1 = 0xDF;             // sensitive on high level(0)
    INTPININV1 = 0xDF;              //Set INT0 Pin sensitivity on normal level(0)
    INTEN2 |= 0x20;                 //Enable PWM7:4 timer module interrupt

    //--Activate the PWM modules and configure the PWM modules as timers
    PWMEN |= 0x20;                  //Enable PWM 5
    PWMTMREN |= 0x20;               //Enable PWM 5 as Timer

    GENINTEN = 0x03;                //Enable global interrupt

   while(1){

   //Wait for PWM0 as timer overflow Flag  PWM0 timer flag pooled
   do
        {
        flagread = PWMTMRF;
        flagread &=0x01;
        }while(flagread == 0);

   PWMTMRF &= 0xFE;                 //Clear the PWM0 Timer Flag
   P1 = P1^0x01;                    //Toggle P1.0
   }//end of while(1)
}// End of main

//------------------------------------------------------- //
//----- Interrupt INT13 - PWM7:4 as Timer  //
//-------------------------------------------------------//
void INT13Interrupt(void) interrupt 13
    {
    char flagread;

    INTEN2 = 0x00;                  //Disable PWM7:4 Timer module interrupt

    flagread = PWMTMRF;             //Read PWM Timer OV Flags
    flagread &= 0x20;               //Check if PWM Timer 5 OV Flag is active
    if(flagread != 0x00)
      P1 = P1^0x20;                 //Toggle P1.5

    PWMTMRF &= 0xDF;                //Clear the PWM Timer 5 OV Flag

    INTEN2 |= 0x20;         //Enable PWM7:4 Timer module interrupt
    }//end of INT0 interrupt
```

12 Enhanced Arithmetic Unit

The VRS51L3074 includes a hardware-based, calculation engine that executes very fast arithmetic operations. With the exception of 16-bit division, which requires 5 cycles, the enhanced arithmetic unit performs multiplication, addition and data shifting in 1 system clock cycle.

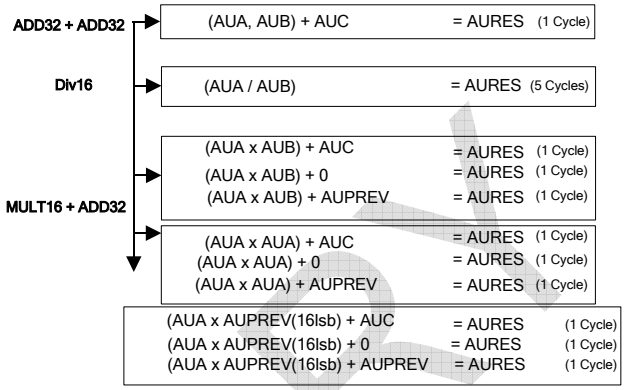
This enables a tremendous performance gain of approximately 30% to 50% for multiplication and accumulation and 700% faster for 16-bit division compared to a standard C compiler when implementing mathematical and digital signal processing (DSP) operations.

The enhanced arithmetic unit features:

- Hardware calculation engine
- Calculation result is ready as soon as the input registers are loaded
- Signed mathematical calculations
- Unsigned MATH operations are possible if the MUL engine operands are limited to 15 bits in length
- Auto/manual reload of AU result register
- Easy implementation of complex mathematical operations
- 16-bit and 32-bit overflow flag
- 32-bit overflow can set an interrupt
- Arithmetic unit operand registers can be cleared individually or simultaneously
- Overflow flags can be configured to stay active until manually cleared
- Can store and use results from previous operations
- Hardware arithmetic unit features a 32-bit barrel shifter in front of the AURES register, which can be employed to scale up/down the result of the operation being performed
- Data shifting operation is performed within the 1 cycle required for multiplication/addition

The arithmetic unit can be configured to perform the operations in the following figure. It can also perform data shifting.

FIGURE 29: VRS51L3074 ARITHMETIC UNIT OPERATION



Where AUA (multiplier), AUB (multiplicand), AUC (accumulator), AURES (result) and AUPREV (previous result) are 16-, 16-, 32-, 32- and 32-bits wide, respectively.

Applications that require arithmetic and DSP operations will benefit from the execution of such calculations on the enhanced arithmetic unit. These include digital filtering, data encryption, sensor output data processing, lookup table replacement, etc. More specifically, applications like FIR filtering that require the repeated execution of 16-bit multiplication and accumulation will benefit tremendously from the arithmetic unit.

12.1 Using the Enhanced Arithmetic Unit

The VRS51L3074's enhanced arithmetic unit operates in signed binary. Access to its registers is executed via the SFR registers, located on SFR Page 1. This page is accessed by setting the SFRPAGE bit of DEVMEMCFG register to 0x01. The DEVMEMCFG register is located at address F6h on both SFR pages.

Before accessing the enhanced arithmetic unit SFR registers, the module must be enabled. This is done by setting AUEN bit 5 of the PERIPHEN2 register to 1. AUEN bit 5 is located at address F5h on both SFR pages.

12.2 Arithmetic Unit Control Registers

With the exception of the barrel shifter, the arithmetic unit's operation is controlled by two SFR registers:

- AUCONFIG1
- AUCONFIG2

The following tables describe these control registers:

TABLE 129: ARITHMETIC CONFIG REGISTER 1 – AUCONFIG1 SFR C2H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	CAPPREV	Read: Always Read as 0 Capture Previous Result Enable 0 = Previous result capture is disabled 1 = Capture the previous result if CAPMODE bit is set to 1
6	CAPMODE	0 = The capture of previous result is automatic each time a write operation is done to the AU0 1 = The capture of the previous result is manual and occurs when the CAPPREV bit is set to 1
5	OVCAPEN	Capture Result on 32-Bit Overflow 0 = No result capture is performed 1 = The AU result is captured and stored when a 32-bit overflow condition occurs
4	READCAP	Read Stored Result 0 = AURES contains current operation result 1 = AURES contains previous result
3:2	ADDSRC[1:0]	AU Adder Input n 32-bit Addition Source **B Input** 00 = 0 (No Add) **01 = C (std 32-bit reg)** 10 = AUPREV 11 = AUC (std 32-bit reg) **A Input** 00=Multiplication 01=Multiplication 10=Multiplication 11= Concatenation of {A, B} + C for 32-bit addition
1:0	MULCMD[1:0]	AU Multiplication Command 00 = AUA x AUB 01 = AUA x AUA 10 = AUA x AUPREV (16 LSB) 11 = AUA x AUB Notes In Divider Mode MULTA_IN = MULT_IN = 0x0000 In Multiplier Mode DIVA_IN = 0x0000 and DIVB_IN = 0x0001

TABLE 130: ARITHMETIC CONFIG REGISTER 2 – AUCONFIG2 SFR C3H

7	6	5	4	3	2	1	0
W	W	W	R/W	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:5	AUREGCLR [2:0]	Read: Always read as 0 Arithmetic Unit Operand Registers Clear 000 = No clear 001 = Clear AUA 010 = Clear AUB 011 = Clear AUC 100 = Clear AUPREV 101 = Clear all AU module registers and overflow flags 110 = Clear overflow flags only
4	AUINTEN	Arithmetic Unit Interrupt Enable 0 = Arithmetic unit interrupt is disabled 1 =-Arithmetic unit interrupt is enabled in divider mode
3	-	Not used, Read as 0
2	DIVOUTRG	AU division is out of range flag This flag is set if AUB = 0x0000 or (AUA = 0x8000 and AUB = 0xFFFF)
1	AUOV16	Arithmetic Unit 16-Bit Overflow Flag 0 = No 16 bit overflow condition detected 1 = a 16-bit overflow occurred Will occur if there is a carry on from bit 15 to bit 1,6 but also from bit 31 to bit 32
0	AUOV32	Arithmetic Unit 32-Bit Overflow Flag 0 = No 16 bit overflow condition detected 1 = Operation result is larger than 32 bits

12.3 Arithmetic Unit Data Registers

The arithmetic unit data registers include operand and result registers that serve to store the numbers being manipulated in mathematical operations. Some of these registers are uniquely for addition (such as AUC), while others can be used for all operations. The use of the arithmetic unit operation registers is described in the following sections.

12.4 AUA and AUB Multiplication (Addition) Input Registers

The AUA and AUB registers serve as 16-bit input operands when performing multiplication.

When the arithmetic unit is configured to perform 32-bit addition, the AUA and the AUB registers are concatenated. In this case, the AUA register contains the upper 16 bits of the 32-bit operand and the AUB contains the lower 16 bits.

TABLE 131: ARITHMETIC UNIT A REGISTER BIT [7:0] - AUA0 SFR A2H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUA[7:0]	LSB of the A Operand Register

TABLE 132: ARITHMETIC UNIT A REGISTER BIT [15:8]- AUA1 SFR A3H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUA[15:8]	MSB of the A Operand Register

TABLE 133:ARITHMETIC UNIT B REGISTER BIT [7:0] - AUB0 SFR B2H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUB[7:0]	LSB of the B Operand Register for Multiplication and Addition Operations

TABLE 134:ARITHMETIC UNIT DIVISION MODE REGISTER – AUB0DIV SFR B1H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUB0DIV[7:0]	Writing to this byte instead of AUB0 will set the arithmetic unit to divisor mode

TABLE 135: ARITHMETIC UNIT B REGISTER BIT [15:8] - AUB1 SFR B3H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUB[15:8]	MSB of the B Operand Register

12.5 AUC Input Register

The AUC register is a 32-bit register used to perform 32-bit addition. The AUPREV register can be substituted with the AUC register or by 0 in the 32-bit addition.

TABLE 136:ARITHMETIC UNIT C REGISTER BIT [7:0] - AUC0 SFR A4H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUC[7:0]	Bit [7:0]of the C Operand Register

TABLE 137: ARITHMETIC UNIT C REGISTER BIT [15:8] - AUC1 SFR A5H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUC[15:8]	Bit [15:8] of the C Operand Register

TABLE 138:ARITHMETIC UNIT C REGISTER BIT [23:16] - AUC2 SFR A6H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUC[23:16]	Bit [23:16] of the C Operand Register

TABLE 139:ARITHMETIC UNIT C REGISTER BIT [31:24] – AUC3 SFR A7H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUC[31:24]	Bit [31:24] of the C Operand Register

12.6 The Arithmetic Unit AURES Register

The AURES register, which is 32 bits wide, is read-only and contains the result of the last arithmetic unit operation. The AURES register is located at the output of the barrel shifter.

When the arithmetic unit is configured to perform multiplication and/or addition, the AURES operates as a 32-bit register that contains the result of the previous operation(s).

However when the arithmetic unit has performed a 16-bit division, the upper 16 bits of the AURES register contain the quotient of the operation, while the lower 16 bits contain the remainder of the division operation.

The barrel shifter is deactivated when the arithmetic unit is performing 16-bit division.

Four SFR registers located in SFR Page 1 provide access to the arithmetic unit AURES register.

TABLE 140: ARITHMETIC UNIT RESULT REGISTER BIT [7:0] - AURES0 SFR B4H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AURES[7:0]	Bit [7:0]of the RESULT Register

TABLE 141: ARITHMETIC UNIT RESULT REGISTER BIT [15:8] - AURES1 SFR 5H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AURES[15:8]	Bit [15:8] of the RESULT Register

TABLE 142: ARITHMETIC UNIT RESULT REGISTER BIT [23:16] – AURES2 SFR B6H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AURES[23:16]	Bit [23:16] of the RESULT Register

TABLE 143: ARITHMETIC UNIT RESULT REGISTER BIT [31:24] – AURES3 SFR B7H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AURES[31:24]	Bit [31:24] of the RESULT Register

12.7 AUPREV Register

The AUPREV register can automatically or manually save the contents of the AURES register and re-inject it into the calculation. This feature is especially useful in applications where the result of a given operation serves as one of the operands for the next one.

As previously mentioned, there are two ways to load the AUPREV register. This is controlled by the CAPMODE bit value as follows:

CAPMODE = 0:
Auto AUPREV load, by writing into the AUA0 register. Selected when CAPPREV = 0.

CAPMODE = 1:
Manual load of AUPREV when the CAPPREV bit is set to 1.

Auto loading of the AUPREV register is useful in FIR filter calculations. For example, it is possible to save a total of eight MOV operations per tap calculation.

TABLE 144: ARITHMETIC UNIT PREVIOUS RESULT BIT [7:0] - AUPREV0 SFR C4H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUPREV[7:0]	Bit [7:0]of the Previous Result Register

TABLE 145:ARITHMETIC UNIT PREVIOUS RESULT BIT [15:8] - AUPREV1 SFR C5H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUPREV[15:8]	Bit [15:8] of the Previous Result Register

TABLE 146:ARITHMETIC UNIT PREVIOUS RESULT BIT [23:16] – AUPREV2 SFR C6H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUPREV[23:16]	Bit [23:16] of the Previous Result Register

TABLE 147:ARITHMETIC UNIT PREVIOUS RESULT BIT [31:24] – AUPREV3 SFR C7H

7	6	5	4	3	2	1	0
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:0	AUPREV[31:24]	Bit [31:24] of the Previous Result Register

12.8 Multiplication and Accumulate Operations

The multiplication and accumulate operations of the arithmetic unit are defined by the MULCMD[1:0] and ADDSRC[1:0] bits of the AUCONFIG1 register.

TABLE 148: MULTIPLICATION OPERATIONS VS. MULCMD BIT OF THE AUCONFIG1

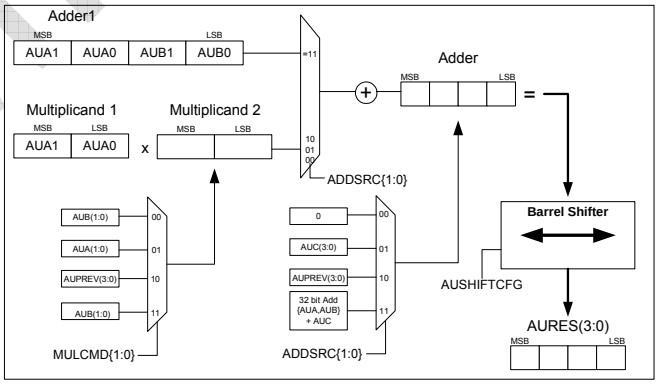
MULCMD[1:0]	Multiplication Operation
00	AUA x AUB
01	AUA x AUA
10	AUA x AUPREV (16LSB)
11	AUA x AUA

TABLE 149: ADDITION OPERATIONS VS. ADDSRC BIT OF THE AUCONFIG1

ADDSRC[1:0]	Addition operation
00	No addition
01	AUC
10	AUPREV[31:0]
11	32-bit addition of [AUA,AUB] + AUC

The following figure provides a block diagram representation of the arithmetic unit operation for multiplication and addition.

FIGURE 30: ARITHMETIC UNIT MULTIPLICATION AND ADDITION OVERVIEW



The following table provides examples of the AUCONFIG and AUSHIFTCFG register values and the corresponding math operations performed by the arithmetic unit. It also provides the value that would be present in the AURES register if the arithmetic unit input registers were initialized to the following values:

- AUA = 3322h
- AUB = 4411h
- AUC = 11111111h
- AUPREV = 12345678h

TABLE 150:CONFIGURATION OF THE ARITHMETIC UNIT, OPERATION AND OUTPUT RESULT

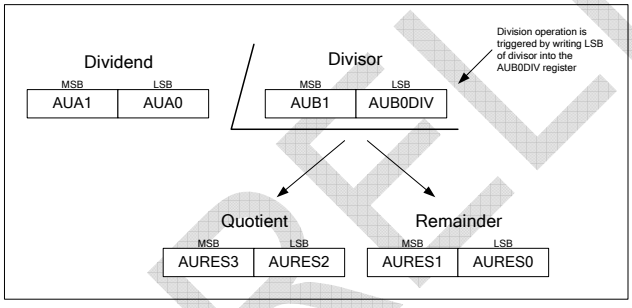
AUCONFIG1	AUSHIFTCFG	Operation	AURES
01h	00h	AUA x AUA	0A369084h
00h	00h	AUA x AUB	0D986D42h
03h	00h	AUA x AUB	0D986D42h
02h	00h	AUA x AUPREV15:0	114563F0h
0Ch,0Dh,0Eh ,0Fh	00h	(AUA,AUB) +AUC] 32 bit addition	44335522h
04h, 07h	00h	(AUA x AUB)+ AUC	1EA97E53h
04h, 07h	01h	((AUA x AUB)+ AUC) x 2 (shift 1 left)	3D52FCA6h
04h, 07h	3Eh	((AUA x AUB)+ AUC) / 4 (shift 2 right)	7AA5F94h

Multiplication and accumulate operations take place within one system clock cycle.

12.9 Division Operation (AUA / AUB1:AUB0DIV)

The VRS51L3074 arithmetic unit can be configured to perform 16-bit division operations: the division of AUA by AUB1,AUB0DIV. The quotient of this operation is stored in the AURES3, AURES2 registers, with the remainder stored in the AURES1, AURES0 registers The following figure represents a 16-bit division.

FIGURE 31: ARITHMETIC UNIT DIVISION OVERVIEW



Writing the LSB of the divisor into the AUB0DIV register will trigger a division operation. Once the division starts, the value written in the AUB0DIV register will be automatically transferred into the AUB0 register.

This operation is neither affected by the barrel shifter nor the multiplication/addition operation, defined by the AUCONFIG register.

The division operation takes five system clock cycles to be complete.

12.10 Barrel Shifter

The arithmetic unit includes a 32-bit barrel shifter at the output of the 32-bit addition unit. The barrel shifter is used to perform right/left shift operations on the arithmetic unit output. The shift operation takes only one cycle.

The barrel shifter can be used to scale the output result of the arithmetic unit.

The shifting range is adjustable from 0 to 16 in both directions. The “shifted” value can be routed to:

- AURES
- AUPREV
- AUOV32

Moreover, the shift left operation can be configured as an arithmetic or logical shift, in which the sign bit is discarded.

TABLE 151: ARITHMETIC UNIT SHIFT REGISTER CONFIG - AUSHIFTCFG SFR C1H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	SHIFTMODE	AU Barrel SHIFTER Shift Mode 0 = Shift value is unsigned 1 = Shift value is signed
6	ARITHSHIFT	AU Arithmetic Shift Enable 0 = Left shift is considered as logical shift (sign bit is lost) 1 = Left shift is arithmetic shift where sign bit is kept
5:0	SHIFT[5:0]	The value of SHIFT[5:0] equals the amplitude of the shift performed on the arithmetic unit result register AURES Positive value represent shift to the left Negative value represent shift to the right

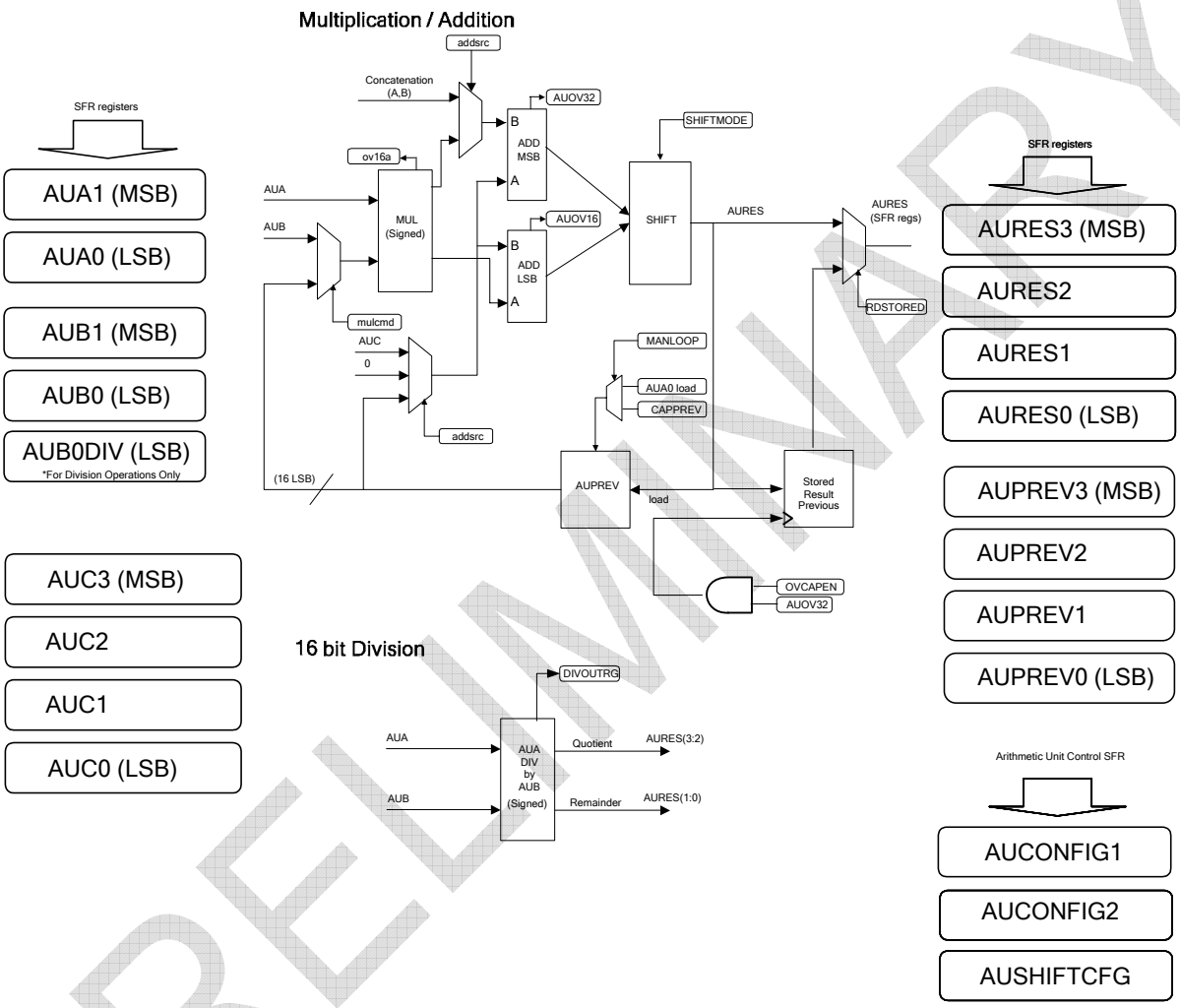
The barrel shifter section operates independently of the multiply and accumulate sections on the arithmetic unit. As such, if the AUSHIFTCFG register bits 5:0 are set to a value other than 0, the value of AUPREV, if derived from the AURES register either automatically or manually, will be affected by the barrel shifter.

When the arithmetic unit is configured to perform multiplication and addition operations, the barrel shifter is active and the shift operation performed depends on the current value of the AUSHIFTCFG register. When the arithmetic unit is configured to perform 16-bit division, the barrel shifter is deactivated.

12.11 VRS51L3074 Arithmetic Unit Block Diagram

The following block diagram provides a hardware description of the registers and the other components that comprise the arithmetic unit on the VRS51L3074.

FIGURE 32: ARITHMETIC UNIT FUNCTIONAL DIAGRAM



12.12 Arithmetic Unit Example Programs

12.12.1 Basic Arithmetic Operations Using the Arithmetic Unit

The following example program demonstrates the required arithmetic unit configuration to perform mathematical operations

```
//-------------------------------------------------------------------------------------------/
// VRS51L3074_MULTACCU1_SDCC.c  //
//-------------------------------------------------------------------------------------------/
//
// DESCRIPTION:    VRS51L3074 Arithmetic Unit Demonstration Program
//
//-------------------------------------------------------------------------------------------/
#include <VRS51L3074_SDCC.h>

//-------------------------------------------------------------/
//                MAIN FUNCTION
//-------------------------------------------------------------/
 void main (void) {
    PERIPHEN2 = 0x20;              //Enable Arithmetic Unit

    DEVMEMCFG = 0x01;             //SELECT SFR PAGE 1


    //Configure Arithmetic Unit to perform math operations

    //Place Value in AUA

    AUA1 = 0x33;
    AUA0 = 0x22;

    //Place Value in AUB
    AUB1 = 0x44;
    AUB0 = 0x11;

    //Place Value in AUC

    AUC3 = 0x11;
    AUC2 = 0x11;
    AUC1 = 0x11;
    AUC0 = 0x11;

    //Place Value in AUPREV
    AUPREV3 = 0x12;
    AUPREV2 = 0x34;
    AUPREV1 = 0x56;
    AUPREV0 = 0x78;

    //--Some operation examples--

    // To perform:  [(AUAxAUA)+0]
    AUCONFIG1  = 0x01;             //Set operation (AUA x AUA) + 0
                                   //AURES = 0A369084h

    // To perform:  [(AUAxAUB)+0]
    AUCONFIG1  = 0x00;             //Set operation (AUA x AUB) + 0
                                   //AURES = 0D986D42h
    // or

    AUCONFIG1  = 0x03;             //Set operation (AUA x AUB) + 0
                                   //AURES = 0D986D42h

    // To perform:  [(AUA x AUPREV[15:0]))+0]
    AUCONFIG1  = 0x02;             //Set operation (AUAxAUPREV)+0
                                   //AURES = 114563F0h

    // To perform:  [ (AUA,AUB) + AUC ]   32 bit addition
    AUCONFIG1  = 0x0C;             //Set operation (AUA,AUB)+ AUC
                                   //AURES = 44335522h

    //or...
    AUCONFIG1  = 0x0D;             //Set operation (AUA,AUB)+ AUC
                                   //AURES = 44335522h
    //or...
    AUCONFIG1  = 0x0E;             //Set operation (AUA,AUB)+ AUC
                                   //AURES = 44335522h
    //or...
    AUCONFIG1  = 0x0F;             //Set operation (AUA,AUB)+ AUC
                                   //AURES = 44335522h

    // To perform:  [ (AUA x AUB)+ AUC ]  No shift
    AUCONFIG1  = 0x04;             //Set operation (AUA x AUB)+ AUC
    AUSHIFTCFG = 0x00;             //No Shift
                                   //AURES = 1EA97E53h
       // To perform:  [ (AUA x AUB)+ AUC ] x 2 (Shift one LEFT)
    AUCONFIG1  = 0x04;             //Set operation (AUA x AUB)+ AUC
    AUSHIFTCFG = 0x01;             //Set barrel shifter to perform one SHIFT LEFT (logical)
                                   //No need to preset the AUSHIFTCFG register for every
                                   //operations
                                   //AURES = 3D52FCA6h

    // To perform:  [ (AUA x AUB)+ AUC ] / 2 (Shift one Right)
    AUCONFIG1  = 0x04;             //Set operation (AUA x AUB)+ AUC
    AUSHIFTCFG = 0x3F;             //Set barrel shifter to perform one SHIFT right
                                   //No need to preset the AUSHIFTCFG register for every
                                   //operations
                                   //AURES = F54BF29h


    DEVMEMCFG = 0x00;             //SELECT SFR PAGE 0

   while(1);


}// End of main
```

12.12.2 FIR Filter Function

The following example program shows the implementation 3074of a FIR filter computation function for one iteration; a data shifting operation; and the definition of the FIR filter coefficient table. The FIR computation algorithm is simple to implement, but requires a lot of processing power. For each new data point, multiplication with the associated coefficients and addition operations must be performed N times (N=number of filter taps).

Since it is hardware-based, the VRS51L3074 arithmetic unit is very efficient in performing operations such as FIR filter computation. In the example below, the COMPUTEFIR loop is the “heart” of the FIR computation. Note that because of the arithmetic unit’s features, very few instructions are needed to perform mathematical operations and the calculation results are ready at the next instruction. This provides a dramatic performance improvement when compared to having to perform all math operations manually, using general processor instructions.

```
//-------------------------------------------------------------//
// VRS51L3074_AU_FIR_asm_c_-SDCC.c   //
//-------------------------------------------------------------------------------------------------------------------------//
//
// DESCRIPTION:  FIR filter demonstration program - mixed ASM and C coding to optimize
//                      the FIR loop speed.
//
//          This program demonstrates the configuration and use of the SPI interface
//          for interface to serial 12-bit A/D and D/A converters.
//          The program reads the A/D and outputs the read value on a D/A converter
//
//          At 40MHzm the 16-tap FIR loop + data shifting of the VRS51L3074 provide the
//          following performances:
//
//          FIR computation using AU module (asm) = 10.4 uSeconds
//          Data shifting (asm) =                   17.2 useconds
//          FIR Computation + Datashift =           27.6 uSeconds  (1/T = 36.2 KHz)
//
//   Rev   1.0
//   Date: August 2005
//-------------------------------------------------------------------------------------------------------------------------//
#include <VRS51L3074_SDCC.h>

//--FIR Filter Coefficient Tables
//;FSAMPLE 480HZ, N=16, LOW PASS 0.1HZ -78DB @ 60HZ
```

```
const int flashfircoef[] =
{0x023D,0x049D,0x086A,0x0D2D,0x1263,0x1752,0x1B30,0x1D51,
0x1D51,0x1B30,0x1752,0x1263,0x0D2D,0x086A,0x049D,0x023D};
//-- Global variables definition
int at 0x30 fircoef[16];
int at 0x50 datastack[16];
unsiged int at 0x75 dacdata;

//---- Functions Declaration ----//
//-- FIR Filter computation function
void FIRCompute(void);
void CopyFIRCoef(void);

//--Gen_ADC
void ReadGen_ADC(void);                 //

//- Gen_DAC
void WriteGen_DAC(unsigned int );

//---Generic functions prototype
void V2KDelay1ms(unsigned int);   //Standard delay function

// Global variables definitions
idata unsigned char cptr = 0x00;

unsigned int adcdata = 0x00;

//-------------------------------------------------------------//
//---------        MAIN FUNCTION   ----------------//
//-------------------------------------------------------------//
void main (void) {
   PERIPHEN2 |= 0x02;               //Enable PWM SFR
   P2PINCFG = 0xF0;                 //P2[3:0] is output
   PWMCLKCFG = 0x10;                //PWM Timer 7 Prescaler = Sys Clock / 2
   //--Configure PWM7 as timer  (will be monitored by interrupt)

   //  PWM Timer 7 counts from 0000 to A2C2h
   PWMCFG =  0x17;                  //Point to MSB MID
   PWMDATA = 0xA2

   PWMCFG =  0x07;                  //Point to LSB MID
   PWMDATA = 0xC2;

   //--Configure and enable PWM as timer Interrupt  to monitor PWM5 only
   INTSRC2 &= 0xDF;                 //PWM7:4 Timer module Interrupt
   INTPINSENS1 = 0xDF;              // sensitive on high level(0)
   INTPININV1 = 0xDF;               //Set INT0 Pin sensitivity on normal level(0)
   INTEN2 |= 0x20;                  //Enable PWM7:4 Timer module interrupt

   //-- Copy FIR filter coefficients to IRAM
   CopyFIRCoef();

   //--Activate the PWM modules and configure the PWM modules as timers
   PWMEN |= 0x80;                   //Enable PWM 7
   PWMTMREN |= 0x80;                //Enable PWM 7 as Timer
   GENINTEN = 0x01;                 //Enable global interrupt
   while(1);

}// End of main

//-------------------------------------------------------------------------//
//---------------------- Interrupt Function-----    ---------------------//
//-------------------------------------------------------------------------//

//-------------------------------------------------------------------------
// NAME:                INT13Interrupt PWMTMR7:4  as Timer
//-------------------------------------------------------------------------
void INT13Interrupt(void) interrupt 13
   {
   char flagread;

   INTEN2 = 0x00;                   //Disable PWM7:4 Timer module interrupt

   flagread = PWMTMRF;              //read PWM Timer OV Flags
   flagread &= 0x80;                //check if PWM Timer 7 OV Flag is Active
   if(flagread != 0x00)
      {
      P2 = P2^0x01;                 //Toggle P2.0 (test)
      ReadGen_ADC();                //Read the A/D Converter
      FIRCompute();                 //Perform the FIR filter computation and write into DAC
      }
   PWMTMRF &= 0x7F;                 //Clear the PWM Timer 7 OV Flag
   INTEN2 |= 0x20;                  //Enable PWM7:4 Timer module interrupt
   }//end of PWM as timer interrupt

//-------------------------------------------------------------------------//
//---------------------- Individual Functions ----------------------------//
//-------------------------------------------------------------------------//

//-------------------------------------------------------------------------
// NAME:                FIRCompute
//-------------------------------------------------------------------------
void FIRCompute()
   {
   char *coef = &fircoef;
   char *ydata = &datastack;
   char fircptr = 0x00;

   PERIPHEN2 |= 0x20;               //Enable the Arithmetic Unit
   P2 = 0xFF;                       //Set P2 = 0xFF to monitor duration for FIR Loop
   *ydata = adcdata & 0x0FF;        //Store the LSB of adc read data
   ydata += 1;
   *ydata = (adcdata >> 8)&0x00FF;  //Store the MSB of adc read data
   DEVMEMCFG = 0x01;                //Switch to SFR Page 1
   AUCONFIG1 = 0x08;                //CAPREV  = 0  : Previous Res capture is automatic
                                    //CAPMODE = 1  : Capture of previous Result
                                    //occurs when AUA0 is written into
                                    //OVCAPEN = 0  : Capture on OV32 disabled
                                    //READCAP = 0  : AURES contains current result
                                    //ADDSRC  = 10 : Add SCR = AUC
                                    //MULCMD  = 00 : Mul cmd = AUA x AUB

   AUCONFIG2 = 0xA0;                //Clear the Arithmetic Unit registers

   _asm
   MOV R0,#0x30;                    //Copy Start address of FIR Coefficient Table into R0
   MOV R1,#0x50;                    //Copy Start address of FIR Data Table into R1
   _endasm;

   // Yn Computation mostly in assembler -- Faster...
   for(fircptr = 0; fircptr < 16; fircptr++)
         {
         _asm
         MOV  0xA2,@R0;             //copy LSB of pointed coefficient to AUA0
         INC  R0;
         MOV  0xA3,@R0;             //copy MSB of pointed coefficient to AUA1
         INC  R0;
         MOV  0xB2,@R1;             //copy LSB of pointed coefficient to AUB0
         INC  R1;
         MOV  0xB3,@R1;             //copy MSB of pointed coefficient to AUB1
         INC  R1;
         _endasm;
         }//end of For cptr

   //-- Performing the data stack shifting allows to save 8.8uS @ 40MHz
   _asm
   MOV R0,#0x6F;
   MOV R1,#0x71;
   _endasm;

   for(fircptr = 16; fircptr > 0; fircptr--)
   {
      _asm
      mov  A,@R0;
      mov  @R1,A;
      dec  R0;
      dec  R1;
      mov  A,@R0;
      mov  @R1,A;
      dec  R0;
      dec  R1;
      _endasm;
      }//end of shift for loop

      //-Scale down the AURES output by 16 using the barrel shifter
      // the coefficient had been scaled up by a factor of 65536
      AUSHIFTCFG = 0x30;
      _asm
      NOP;
      _endasm;
      P2 = 0x00;                    //Set P2 = 0x00 to signal the end of the FIR Loop

      dacdata = (AURES1 << 8) + AURES0;

   //Reset the Barrel shifter
   AUSHIFTCFG = 0x00;
   // Note:
   //      In this case, 6 System clock cycles could be saved
   //      by reading AURES3 and AURES2 directly
   DEVMEMCFG = 0x00;                //Switch to SFR Page 0
   WriteGen_DAC(dacdata);           //Write data to SPI DAC
   }//End of FIRCompute
```

```
//-------------------------------------------------------------------------
// NAME:                CopyFIRCoef
//-------------------------------------------------------------------------
// DESCRIPTION: Copy the FIR Filter Coefficient into
//              SRAM variable which is faster access
//              than Flash
//-------------------------------------------------------------------------
void CopyFIRCoef(void)
   {
   char cptr = 0x00;
         for(cptr = 0x00; cptr < 16; cptr++)
      fircoef[cptr]= flashfircoef[cptr];
   }//End of CopyFIRCoef

//-------------------------------------------------------------------------
// NAME:                ReadGen_ADC
//-------------------------------------------------------------------------
// DESCRIPTION: Read the Gen_ADC A/D
//      ADC is connected to SPI interface using CS0
//      Max clk speed is 3.2MHz, Fosc = 40MHz assumed
//-------------------------------------------------------------------------
void ReadGen_ADC()
   {
   int cptr = 0x00;
   char readflag = 0x00;

   //SPI Configuration Section
   /(Can be moved to Main function if only one device is connected to the SPI interface)

   PERIPHEN1 |= 0xC0;              //Make sure the SPI interface is activated

   //--Wait activity stops on the SPI interface (Monitor SPINOCS)
   while(!(SPISTATUS &= 0x08));

   SPICTRL = 0x65;                 //SPICLK = /16 (2.5MHz)
                                   //CS0 Active
                                   //SPI Mode 1 Phase = 1, POL = 0
                                   //SPI Master Mode

   SPICONFIG = 0x40;               //SPI Chip select is automatic
                                   //Clear SPIUNDEFC flag
                                   //SPILOAD = 0 -> Manual CS3 behaviour
                                   //No SPI interrupt used
   SPISTATUS = 0x00;               //SPI transactions are in MSB first format
   SPISIZE = 0x0E;                 //SPI transaction size are 15-bit

   //-Dummy Read the SPI RX buffer to clear the RXAV flag
   readflag = SPIRXTX0;

   //-Perform the SPI read
    SPIRXTX0 = 0x00;               //Writing to the SPIRXTX0 will trigger the SPI
                                   //Transaction
   //Wait for the SPI RX AV Flag being set
   while(!(SPISTATUS &= 0x02));
   /*
   // -- It is possible to monitor the SPINOCS flag instead of the SPIRXAV flag
   //The code piece below shows how to do it.  However in that case,
   //No that the reading of the SPISTATUS register must be done at
   //least 4 system clock cycles after the write operation to the SPIRXTX0 register

   //-Wait for SPINOCS Flag have time to be updated
   _asm
   NOP;
   _endasm;

   //--Wait activity stops on the SPI interface
   while(!(SPISTATUS &= 0x08));
   */

   //Read SPI data
   adcdata= (SPIRXTX1 << 8);
   adcdata+= SPIRXTX0;
   adcdata&= 0x0FFF;                //isolate the 12 lsb of the read value
   }//end of ReadGen_ADC


//-------------------------------------------------------------------------//
// NAME:                WriteGen_DAC
//-------------------------------------------------------------------------//
// DESCRIPTION: Write 12bit Data into the Gen_DAC device
//              ADC is connected to SPI interface using CS1
//              Max clk speed is 12.5MHz, Fosc = 40MHz assumed
//              We will set the SPI prescaler to sysclk / 8
//
void WriteGen_DAC(unsigned int dacdata)
   {
   char subdata = 0x00;
   char readflag = 0x00;

   PERIPHEN1 |= 0xC0;          //Make sure the SPI interface is activated

   //--Wait activity stops on the SPI interface (Monitor SPINOCS)
   while(!(SPISTATUS &= 0x08));

   //SPI Configuration Section
   //Can be moved to main function if only one device is connected to the SPI interface

   SPICTRL = 0x4D;             //SPICLK = /8 (MHz)
                               //CS1 Active
                               //SPI Mode 1 Phase = 1, POL = 0
                               //SPI Master Mode

   SPICONFIG = 0x40;           //SPI Chip select is automatic
                               //Clear SPIUNDEFC Flag
                               //SPILOAD = 0 -> Manual CS3 behaviour
                               //No SPI interrupt used

   SPISTATUS = 0x00;           //SPI transactions are in MSB first format
   SPISIZE = 0x0B;             //SPI transaction size are 12 bit

   //-Format the 12 bit data so data bit 11 is positioned on bit 7 of SPIRXTX0
   // and data bit 0 is positioned on bit 4 of SPIRXTX1 and perform the SPI write operation

   dacdata &= 0x0FFF;          //Make sure dacdata is <= 0FFFh  (12 bit)
   SPIRXTX3 = 0x00;
   SPIRXTX2 = 0x00;
   SPIRXTX1  = (dacdata << 4)& 0xF0;

   //-Dummy read the SPI RX buffer to clear the RXAV Flag  (facultative if SPINOCS is
monitored)
   readflag = SPIRXTX0;

   SPIRXTX0 = (dacdata >> 4);  //Writing to SPIRXTX0 will trigger the transmission

   //--Wait the SPI transaction completes
   //  This section can be omitted if a check of activity on the SPI interface
   //  is made before each access to it in master mode

   //Wait for the SPI RX AV flag being set
   while(!(SPISTATUS &= 0x02));
   // -- It is possible to monitor the SPINOCS flag instead of the SPIRXAV flag
   //The code piece below shows how to do it.  However in that case,
   //No that the reading of the SPISTATUS register must be done at
   //least 4 system clock cycles after the write operation to the SPIRXTX0 register
   /*
   //-Wait for SPINOCS flag have time to be updated
   _asm
      NOP;
   _endasm;
   //--Wait activity stops on the SPI interface  (monitor SPINOCS Flag)
   while(!(SPISTATUS &= 0x08));
   */
   }//end of WriteGen_DAC

//-------------------------------------------------------------------------------------//
// NAME:                V2KDelay1ms
//-------------------------------------------------------------------------------------//
// DESCRIPTION: VRS3074 specific 1 millisecond delay function
//                    Using Timer 0 and calibrated for 40MHz oscillator
//-------------------------------------------------------------------------------------//
void V2KDelay1ms(unsigned int dlais){
idata unsigned char x=0;
idata unsigned int dlaisloop;

   PERIPHEN1 |= 0x01;                  //LOAD PERIPHEN1 REG

   dlaisloop = dlais;
   while ( dlaisloop > 0)
      {
      TH0 = 0x63;              //TIMER0 RELOAD VALUE FOR 1MS AT 40MHZ
      TL0 = 0xC0;
      T0T1CLKCFG = 0x00;             //NO PRESCALER FOR TIMER 0 CLOCK
      T0CON = 0x04;              //START TIMER 0,  COUNT UP
   do{
   x=T0CON;
   x= x & 0x80;
   }while(x==0);

   T0CON = 0x00;                       //Stop Timer 0
   dlaisloop = dlaisloop-1;
   }//end of while dlais...
      PERIPHEN1 &= 0xFE;               //Disable Timer 0
   }//End of function V2KDelay1ms
```

13 Watchdog Timer

The VRS51L3074 includes a watchdog timer which resets the processor in case of a program malfunction. The watchdog timer is composed of a 14-bit prescaler, which derives its source from the active system clock. An overflow of the watchdog timer resets the VRS51L3074. The WDTCFG SFR register controls the watchdog timer operations.

TABLE 152: THE WATCHDOG TIMER REGISTER - WDTCFG 91H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:4	WDTPERIOD	Watchdog Timer Period Configuration *see table below
3	WTIMEROVF	WDT as Timer Overflow Flag 0 = WDT as timer as not expired 1 = WDT as timer has overflow
2	ASTIMER	Watchdog as Timer 0 = WDT mode 1 = WDT operate as a regular timer (no reset) Writing to this bit will clear the timer
1	WDTOVF	Read: 0 = Watchdog is counting 1 = Watchdog timer period has expired
		Write: 0 = No action 1 = Clear the watchdog timer flag
0	WDTRESET	Read: No Action
		Watchdog Timer Reset To reset the watchdog timer, two consecutive writes to the WDTRESET bit must be made: First clear the WDTRESET bit and second, set it to 1

13.1 WDT Timeout Period

The watchdog timer timeout period is controlled by adjusting bit 7:4 of the WDTCFG register. The following table provides the approximate timeout vs. the selected WDTPERIOD.

TABLE 153: WATCHDOG TIMER REGISTER TIMEOUT PERIOD

WDTPERIOD Value (4 bit)	Actual WDT Period**	Approx Timeout** (40MHz)
0000	0x3FFF*	409 – 600us
0001	0x3FFE	819-1000 us
0010	0x3FFD	1.23 – 1.36 ms
0011	0x3FFB	2.05 – 2.2 ms
0100	0x3FF4	4.92 ms
0101	0x3FE8	9.83 ms
0110	0x3FCF	20.07 ms
0111	0x3F86	49.97 ms
1000	0x3F49	74.96 ms
1001	0x3F0C	99.94 ms
1010	0x3E9E	249.86 ms
1011	0x3B3B	500.12 ms
1100	0x38D9	749.98 ms
1101	0x3677	999.83 ms
1110	0x2364	2.99 s
1111	0x0000	6.71s

*Not available in timer mode

The watchdog timer timeout period is calculated as follows:

$$\text{WDT Period*} = \frac{16384*(0x4000 - \text{WDT Period})}{\text{Fosc}}$$

**For a given configuration, the timeout period of the watchdog timer may vary by about 200us. This delay is caused by internal timing of the watchdog timer module.*

13.2 Resetting the Watchdog Timer

To reset the watchdog timer, two consecutive write operations to the WDTCFG register must be performed. During the first write operation, the WDTRESET bit must be cleared. During the second write operation, the WDTRESET should be set to 1.

This sequence is also required to set a new value for WDTPERIOD. For example, if the watchdog period is set to 100ms, the following sequence of operations will reset the watchdog timer:

```
MOV     WDTCFG,#92h
MOV     WDTCFG,#93h
```

13.3 Using the Watchdog as a Timer

The VRS51L3074 watchdog timer can also be used as a timer. In this case, the timeout period is defined by the watchdog timer period value. Due to the presence of the 14-bit prescaler, long timeout periods can be achieved.

Configuring the watchdog timer operation as a general purpose timer is achieved by:

- Setting the ASTIMER bit of the WDTCFG register to 1
- Selecting the timer maximum time value of WDTPeriod
- Performing a watchdog timer reset sequence to clear the timer and apply the timer configuration

The WTIMERFLAG bit of the WDTCFG register is used to monitor the timer overflow. When configured in timer mode, the watchdog timer does not reset the VRS51L3074 and cannot trigger an interrupt.

13.4 Watchdog Timer Example Programs

13.4.1 Initialization and Reset of the Watchdog Timer

```
//-------------------------------------------------------------------------------------------------//
// VRS51L3074-WDT_Demo_SDCC.c  //
//-------------------------------------------------------------------------------------------------//
// DESCRIPTION:    VRS51L3074 Watchdog Timer Demonstration Program
//              *This Program Set P1 as output
//              *P1 is set to 0xFF for 100ms
//              *Initialize the watchdog timer with a timeout period of 20ms
//              *Clear P1
//              *Start a delay function
//              *If the Delay parameter of the delay function is larger than the
//              Timeout period of the watchdog timer, the WDT will reset the VRS51L3074
//              which will bring back P1 to high level
//-------------------------------------------------------------------------------------------------//
#include <VRS51L3074_SDCC.h>

// --- function prototypes

void delay(unsigned int);

//-----------------------------------------------------//
//              MAIN FUNCTION                  //
//-----------------------------------------------------//

void main (void) {

    PERIPHEN1 = 0x01;              //Enable Timer 0
    PERIPHEN2 = 0x08;              //Enable IOPORT

    P1PINCFG = 0x00;               //Config port 1 as output


    //-- Enable the Watchdog Timer
    PERIPHEN2 |= 0x04;
    P1 = 0xFF;                      //Set P1 to output 0xFF
    delay(100);                    //Keep P1 high for 100ms

    //-- Configure the watchdog timer
    WDTCFG = 0x62;          //Configure and Reset the Watchdog Timer
    WDTCFG = 0x63;          //Bit 7:4  = WDTPERIOD : Define the timeout period (20ms)
                              //Bit 3    = WTIMEROVF : WDT as timer overflow flag
                              //Bit 2    = ASTIMER   : WDT mode (0=WDT, 1=Timer)
                              //Bit 1    = WDTOVF    : WDT overflow (Timeout) Flag
                              //Bit 0    = WDTRESET  : WDT reset. To reset WDT
                              //this bit must be cleared, then set

    P1 = 0x00;                //Clear P1
    do{
     delay(10);               //If delay > 20ms then the WDT will reset the VRS51L3074
                               //and P1 will return to high

     WDTCFG = 0x62;          //Reset the watchdog timer
     WDTCFG = 0x63;
    }while(1);                //Loop Forever

}// End of main

//;-----------------------------------------------------------------//
//;- DELAY1MSTO : 1MS DELAY USING TIMER0
//;
//; CALIBRATED FOR 40MHZ
//;-----------------------------------------------------------------//
void delay(unsigned int dlais){

idata unsigned char x=0;
idata unsigned int dlaisloop;

    x = PERIPHEN1;                  //LOAD PERIPHEN1 REG
    x |= 0x01;                      //ENABLE TIMER 0
    PERIPHEN1 = x;

    dlaisloop = dlais;
    while ( dlaisloop > 0)
      {
       TH0 = 0x63;                  //TIMER0 RELOAD VALUE FOR 1MS AT 40MHZ
      TL0 = 0xC0;

       T0T1CLKCFG = 0x00;          //NO PRESCALER FOR TIMER 0 CLOCK
       T0CON = 0x04;                //START TIMER 0,  COUNT UP

        do{
            x=T0CON;
            x= x & 0x80;
            }while(x==0);

        T0CON = 0x00;                //Stop Timer 0

        dlaisloop = dlaisloop-1;

        }//end of while dlais...

       x = PERIPHEN1;               //LOAD PERIPHEN1 REG
       x = x & 0xFE;                    //DISABLEBLE TIMER 0
       PERIPHEN1 = x;
    }//End of function delais
```

14 VRS51L3074 Interrupts

The VRS51L3074 has a comprehensive set of 49 interrupt sources and uses 16 interrupt vectors to handle them. The interrupts are categorized in two distinct groups:

- Module interrupt
- Pin change interrupts

The module interrupts include interrupts that are generated by VRS51L3074 peripherals such as the UARTs, SPI, I²C , PWC and port change monitoring modules.

As their name implies, the pin change interrupts are interrupts that are generated by predefined conditions at the physical pin level: . The pin change interrupts can be caused by a level or an edge (rising or falling) on a given pin. Standard 8051 INT0 and INT1 interrupts are considered pin change interrupts. The VRS51L3074 includes INT0 and INT1, as well as 14 other pin interrupts distributed on ports 0 and 3.

The interrupt sources share 16 interrupt vectors from 00h to 7Bh. Each interrupt vector can be configured to respond to either a pin change interrupt or a module interrupt. The two following diagrams provide an overview of the VRS51L3074 modules/pin interrupt structure, the associated SFR registers and the interaction among the interrupt management SFRs.

FIGURE 33: INTERRUPT SOURCES DETAILED VIEW

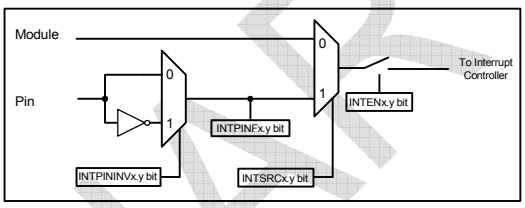
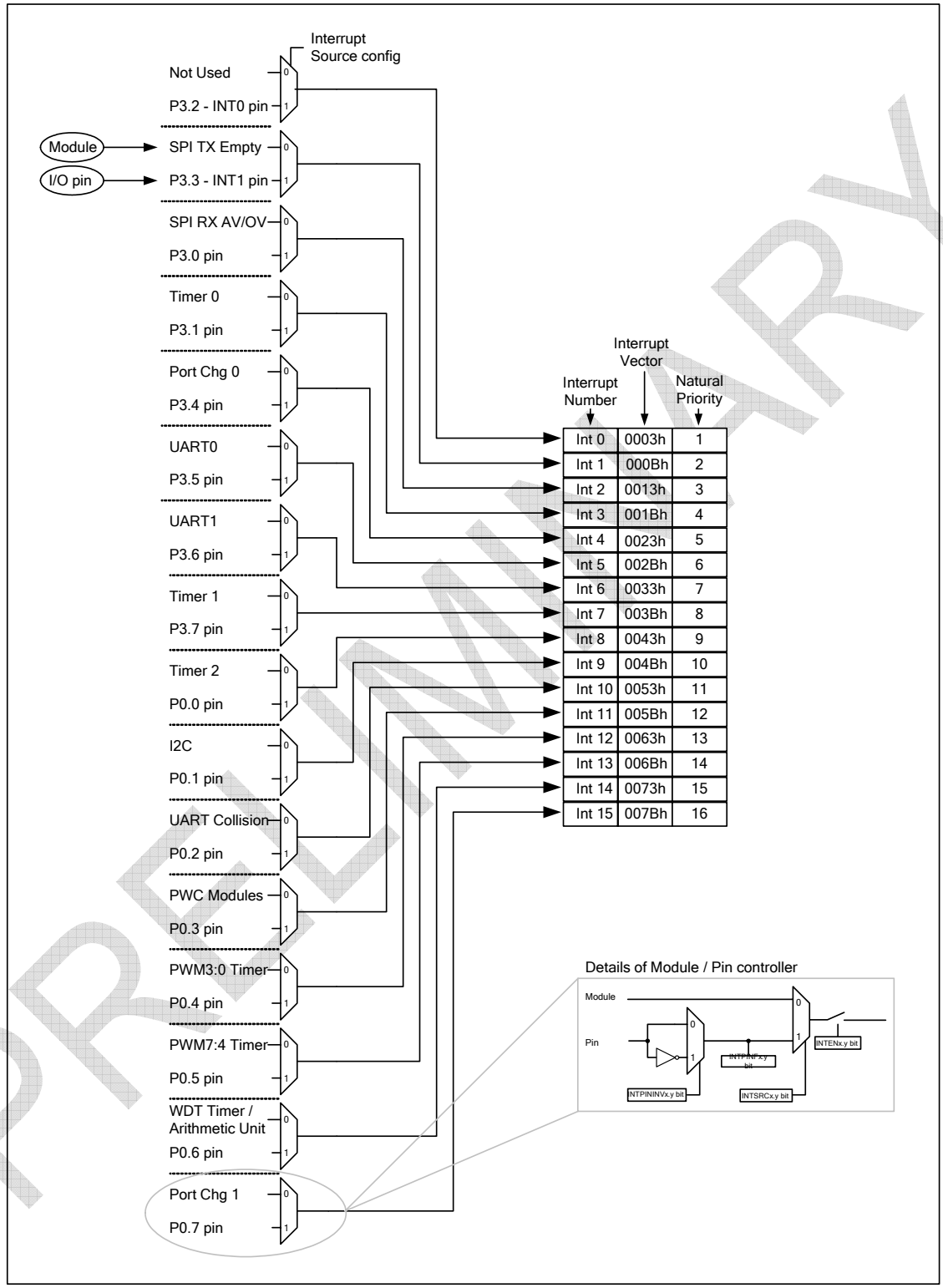


FIGURE 34: INTERRUPT SOURCES OVERVIEW



The interaction between the interrupt management configuration registers is summarized in the following table. The paragraphs below describe each one of these registers in detail.

TABLE 154:VRS51L3074 INTERRUPT CONFIGURATION SUMMARY

Int #	Priority	Interrupt Vector	Interrupt Enable	Interrupt Priority	Interrupt Source	Connected Modules	Connected Pin	Pin Inversion	Pin Sensitivity	Pin Interrupt Flag
INT 0	1	0003h	INTEN1.0	INTPRI1.0	INTSRC1.0	None	P3.2-INT0	IPINTINV1.0	IPINSENS1.0	IPINFLAG1.0
Int 1	2	000Bh	INTEN1.1	INTPRI1.1	INTSRC1.1	SPI TX Empty	P3.3-INT1	IPINTINV1.1	IPINSENS1.1	IPINFLAG1.1
Int 2	3	0013h	INTEN1.2	INTPRI1.2	INTSRC1.2	SPI RX Available SPI RX Overrun	P3.0	IPINTINV1.2	IPINSENS1.2	IPINFLAG1.2
Int 3	4	001Bh	INTEN1.3	INTPRI1.3	INTSRC1.3	Timer 0	P3.1	IPINTINV1.3	IPINSENS1.3	IPINFLAG1.3
Int 4	5	0023h	INTEN1.4	INTPRI1.4	INTSRC1.4	Port Change 0	P3.4	IPINTINV1.4	IPINSENS1.4	IPINFLAG1.4
Int 5	6	002Bh	INTEN1.5	INTPRI1.5	INTSRC1.5	UART0 Tx Empty UART0 RX Available UART0 RX Overrun UART0 Timer OV	P3.5	IPINTINV1.5	IPINSENS1.5	IPINFLAG1.5
Int 6	7	0033h	INTEN1.6	INTPRI1.6	INTSRC1.6	UART1 Tx Empty UART1 RX Available UART1 RX Overrun UART1 Timer OV	P3.6	IPINTINV1.6	IPINSENS1.6	IPINFLAG1.6
Int 7	8	003Bh	INTEN1.7	INTPRI1.7	INTSRC1.7	Timer 1	P3.7	IPINTINV1.7	IPINSENS1.7	IPINFLAG1.7
Int 8	9	0043h	INTEN2.0	INTPRI2.0	INTSRC2.0	Timer 2	P0.0	IPINTINV2.0	IPINSENS2.0	IPINFLAG2.0
Int 9	10	004Bh	INTEN2.1	INTPRI2.1	INTSRC2.1	I²C Tx Empty I²C RX Available I²C RX Overrun	P0.1	IPINTINV2.1	IPINSENS2.1	IPINFLAG2.1
Int 10	11	0053h	INTEN2.2	INTPRI2.2	INTSRC2.2	UART0 Collision UART1 Collision I²C Master Lost Arbitration	P0.2	IPINTINV2.2	IPINSENS2.2	IPINFLAG2.2
Int 11	12	005Bh	INTEN2.3	INTPRI2.3	INTSRC2.3	PWC 0 End Condition PWC 1 End Condition	P0.3	IPINTINV2.3	IPINSENS2.3	IPINFLAG2.3
Int 12	13	0063h	INTEN2.4	INTPRI2.4	INTSRC2.4	PWM3 as Timer OV PWM2 as Timer OV PWM1 as Timer OV PWM0 as Timer OV	P0.4	IPINTINV2.4	IPINSENS2.4	IPINFLAG2.4
Int 13	14	006Bh	INTEN2.5	INTPRI2.5	INTSRC2.5	PWM7as Timer OV PWM6as Timer OV PWM5as Timer OV PWM4as Timer OV	P0.5	IPINTINV2.5	IPINSENS2.5	IPINFLAG2.5
Int 14	15	0073h	INTEN2.6	INTPRI2.6	INTSRC2.6	Watchdog as Timer OV Arithmetic Unit OV	P0.6	IPINTINV2.6	IPINSENS2.6	IPINFLAG2.6
Int 15	16	007Bh	INTEN2.7	INTPRI2.7	INTSRC2.7	Port Change 1	P0.7	IPINTINV2.7	IPINSENS2.7	IPINFLAG2.7

14.1 Interrupt Enable Registers

The interrupt enable and the general interrupt enable registers establish the link between the peripheral module/pin interrupt signals and the processor interrupt system.

The GENINTEN register controls activation of the global interrupt. On the VRS51L3074, only the least significant bit of the GENINTEN is used. The GENINTEN register is similar to the standard 8051 EA bit. When the GENINTEN bit is set to 1, all the enabled interrupts emanating from the modules/pins will reach the interrupt controller.

TABLE 155:GENINTEN SFR REGISTER - NAME SFR E8H

7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	R/W
							0

Bit	Mnemonic	Description
7:2	Unused	
1	CLRPININT	It is recommended to set this bit to 1 before enabling a pin interrupt to avoid receiving an interrupt right after GENINTEN bit is set
0	GENINTEN	General Interrupt Enable 0 = All enabled interrupts are masked (deactivated) 1 = All enabled interrupt can raise an interrupt

When a given interrupt bit is set to 1, the corresponding interrupt path is activated.

TABLE 156: INT ENABLE 1 REGISTER - INTEN1 (MODULES /PIN/INT VECTOR) SFR 88H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	T1IEN	Timer 1 Interrupt Enable
	P3.7 pin	P3.7 pin if interrupt source is set to pin
	Int 7	Interrupt vector 7 at address 003Bh
6	U1IEN	UART1 Interrupt Enable o UART1 Tx Empty o UART1 Rx Available o UART1 Rx Overrun o UART1 Baud Rate Generator as Timer Overflow
	P3.6 pin	P3.6 pin if interrupt source is set to pin
	Int 6	Interrupt vector 6 at address 0033h
5	U0IEN	UART0 Interrupt Enable o UART0 Tx Empty o UART0 Rx Available o UART0 Rx Overrun o UART0 Baud Rate Generator as Timer Overflow
	P3.5 pin	P3.5 pin if interrupt source set to pin
	Int 5	Interrupt vector 5 at address 0002Bh
4	PCHGIEN0	Port Change Interrupt Module 0 Enable
	P3.4 pin	P3.4 pin if interrupt source is set to pin
	Int 4	Interrupt vector 4 at address 0023h
3	T0IEN	Timer 2 Interrupt Enable
	P3.3 pin	P3.3 pin if interrupt source is set to pin
	Int 3	Interrupt vector 3 at address 001Bh
2	SPIRXOVIEN	SPI Interrupt Enable SPI Rx Available SPI Rx Overrun
	P3.0	P3.0 pin if interrupt source is set to pin
	Int 2	Interrupt vector 2 at address 0013h
1	SPITXEIEN	SPI Tx Empty Interrupt Enable
	P3.3 pin	P3.3 pin if interrupt source is set to pin
	Int 1	Interrupt vector 0 at address 000Bh
0	No Module	Unused
	P3.2 pin	P3.2 pin if interrupt source is set to pin
	Int 0	Interrupt vector 0 at address 0003h

TABLE 157: INT ENABLE 2 REGISTER INTEN2 (MODULES /PIN/INT VECTOR) SFR A8H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	PCHGIEN1	Port Change Interrupt Module 1 Enable
	P0.7 pin	P0.7 pin if interrupt source is set to pin
	Int 15	Interrupt vector 8 at address 007Bh
6	AUWDTIEN	Watchdog Timer and Arithmetic Unit Interrupt Enable o Watchdog as Timer Overflow o Arithmetic Unit 32-bit Overflow
	P0.6 pin	P0.6 pin if interrupt source is set to pin
	Int14	Interrupt vector 8 at address 0073h
5	PWMT74IEN	PWM as Timer 7 to 4 Overflow Interrupt Enable o PWM as Timer Module 7 Overflow o PWM as Timer Module 6 Overflow o PWM as Timer Module 5 Overflow o PWM as Timer Module 4 Overflow
	P0.5 pin	P0.5 pin if interrupt source set to pin
	Int 13	Interrupt vector 8 at address 006Bh
4	PWMT30IEN	PWM as Timer 3 to 0 Overflow Interrupt Enable o PWM as Timer Module 3 Overflow o PWM as Timer Module 2 Overflow o PWM as Timer Module 1 Overflow o PWM as Timer Module 0 Overflow
	P0.4 pin	P0.4 pin if interrupt source is set to pin
	Int 12	Interrupt vector 8 at address 0063h
3	PWCIEN	Pulse Width Counter Interrupt Enable o PWC0 END condition occurred o PWC1 END condition occurred
	P0.3 pin	P0.3 pin if interrupt source set to pin
	Int 11	Interrupt vector 11 at address 005Bh
2	I2CUCOLIEN	I²C and UARTs Interrupts Enable o I²C Master Lost Arbitration o UART0 Collision Interrupt o UART1 Collision Interrupt
	P0.2 pin	P0.2 pin if interrupt source is set to pin
	Int 10	Interrupt vector 10 at address 0053h
1	I2CIEN	I²C Interrupts Enable o TX Empty o RX Available o RX Overrun
	P0.1 pin	P0.1 pin if interrupt source set to pin
	Int 9	Interrupt vector 9 at address 004Bh
0	T2IEN	Timer 2 Interrupt Enable (INTSCR
	P0.0 pin	P0.0 pin if interrupt source is set to pin
	Int 8	Interrupt vector 8 at address 0043h

14.2 Interrupt Source

Each one of the 16 interrupt vectors on the VRS51L3074 can be configured to function as either a peripheral module or a pin change interrupt. The selection of the interrupt source is handled by the INTSRC1 and the INTSRC2 registers.

By default, the interrupt source is set to peripheral module. However, setting the INTSRC bit to 1 will "associate" the corresponding interrupt vector to the corresponding pin interrupt.

When a given interrupt vector is associated with a module, the corresponding bit of the IPINSENSx must be set to 0, so it is level sensitive (reset value).

TABLE 158:INTERRUPT SOURCE 1 REGISTER - INTSRC1 SFR E4H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	INTSRC1.7	Interrupt 7 Source 0 = Timer 1 1 = P3.7
6	INTSRC1.6	Interrupt 6 Source 0 = UART1 1 = P3.6
5	INTSRC1.5	Interrupt 5 Source 0 = UART0 1 = P3.5
4	INTSRC1.4	Interrupt 4 Source 0 = Port Change 0 1 = P3.4
3	INTSRC1.3	Interrupt 3 Source 0 = Timer 0 1 = P3.1
2	INTSRC1.2	Interrupt 2 Source 0 = SPI RXAV, SPI RXOV 1 = P3.0
1	INTSRC1.1	Interrupt 1 Source 0 = SPI Tx EMPTY 1 = P3.3
0	INTSRC1.0	Interrupt 0 Source 0 = - 1 = P3.2

TABLE 159:INTERRUPT SOURCE 2 REGISTER - INTSRC2 SFR E5H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	INTSRC2.7	Interrupt 15 Source 0 = Port Change 0 1 = P0.7
6	INTSRC2.6	Interrupt 14 0 = WDT Timer OV, AU OV 1 = P0.6
5	INTSRC2.5	Interrupt 13 Source 0 = PWM7:4 Timer 1 = P0.5
4	INTSRC2.4	Interrupt 12 Source 0 = PWM3:0 Timer OV 1 = P0.4
3	INTSRC2.3	Interrupt 11 Source 0 = PWC0, PWC1 1 = P0.3
2	INTSRC2.2	Interrupt 10 Source 0 = UARTs Coll, I²C Lost Arbitration 1 = P0.2
1	INTSRC2.1	Interrupt 9 Source 0 = I²C 1 = P0.1
0	INTSRC2.0	Interrupt 8 Source 0 = Timer 2 1 = P0.0

14.3 Interrupt Priority

The INTPRIx registers enable the user to modify the interrupt priority of either the module or the pin interrupts. When the INTPRIx is set to 0, the natural priority of module/pin interrupts prevails. Setting the INTPRIx register bit to 1 will set the corresponding module/pin priority to high.

If more than two module/pin interrupts are simultaneously set to high priority, the natural priority order will apply: Priority will be give to the module/pin interrupts with high priority, over normal priority.

TABLE 160:INTERRUPT PRIORITY 1 REGISTER - INTPRI1 SFR E2H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	T1P37PRI	Interrupt 7 Priority Level (Timer 1 / P3.7) 0 = Normal Priority 1 = High Priority
6	U1P36PRI	Interrupt 6 Priority Level (UART1 / P3.6) 0 = Normal Priority 1 = High Priority
5	U0P35PRI	Interrupt 5 Priority Level (UART0 / P3.5) 0 = Normal Priority 1 = High Priority
4	PC0P34PRI	Interrupt 4 Priority Level (Port Chg 0 / P3.4) 0 = Normal Priority 1 = High Priority
3	T0P31PRI	Interrupt 3 Priority Level (Timer 0 / P3.1) 0 = Normal Priority 1 = High Priority
2	SRP30PRI	Interrupt 2 Priority Level (SPI RX / P3.0) 0 = Normal Priority 1 = High Priority
1	STP33PRI	Interrupt 1 Priority Level (SPI TX / P3.3) 0 = Normal Priority 1 = High Priority
0	INT0P32PRI	Interrupt 0 Priority Level (INT0 / P3.2) 0 = Normal Priority 1 = High Priority

TABLE 161:INTERRUPT PRIORITY 2 REGISTER - INTPRI2 SFR E3H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	PC1P07PRI	Interrupt 15 Priority Level (Port Chg 1 / P0.0) 0 = Normal Priority 1 = High Priority
6	AIP06PRI	Interrupt 14 Priority Level (WDT, AU / P0.6) 0 = Normal Priority 1 = High Priority
5	PWHP05PRI	Interrupt 13 Priority Level (PWM7:4 timer / P0.5) 0 = Normal Priority 1 = High Priority
4	PWLP04PRI	Interrupt 12 Priority Level (PWM3:0 timer / P0.4) 0 = Normal Priority 1 = High Priority
3	PWCP02PRI	Interrupt 11 Priority Level (PWC0, PWC1 / P0.3) 0 = Normal Priority 1 = High Priority
2	INT10P01PRI	Interrupt 10 Priority Level (UARTs Coll, I²C Lost Arbitration / P0.2) 0 = Normal Priority 1 = High Priority
1	I2CP01PRI	Interrupt 9 Priority Level (I²C / P0.1) 0 = Normal Priority 1 = High Priority
0	T2P00PRI	Interrupt 8 Priority Level (Timer 2 / P0.0) 0 = Normal Priority 1 = High Priority

14.4 Pin Inversion Setting

TABLE 162: IMPACT OF PIN INVERSION SETTING ON PIN INTERRUPT SENSITIVITY

Pin Inversion	Interrupt Condition
0	Normal Interrupt Polarity Sensitivity
1	Inverted Interrupt Polarity Sensitivity

TABLE 163:INTERRUPT PIN INVERSION 1 REGISTER - IPININV1 SFR D6H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	P37IINV	Interrupt 7 Pin Polarity 0 = P3.7 1 = P3.7 Inverted
6	P36IINV	Interrupt 6 Pin Polarity 0 = P3.6 1 = P3.6 Inverted
5	P35IINV	Interrupt 5 Pin Polarity 0 = P3.5 1 = P3.5 Inverted
4	P34IINV	Interrupt 4 Pin Polarity 0 = P3.4 1 = P3.4 Inverted
3	P31IINV	Interrupt 3 Pin Polarity 0 = P3.1 1 = P3.1 Inverted
2	P30IINV	Interrupt 2 Pin Polarity 0 = P3.0 1 = P3.0 Inverted
1	P33IINV	Interrupt 1 Pin Polarity 0 = P3.3 1 = P3.3 Inverted
0	P32IINV	Interrupt 0 Pin Polarity 0 = P3.2 1 = P3.2 Inverted

TABLE 164: INTERRUPT PIN INVERSION 2 REGISTER - IPININV1 SFR D7H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	P07IINV	Interrupt 15 Pin Polarity 0 = P0.7 1 = P0.7 Inverted
6	P06IINV	Interrupt 14 Pin Polarity 0 = P0.6 1 = P0.6 Inverted
5	P05IINV	Interrupt 13 Pin Polarity 0 = P0.5 1 = P0.5 Inverted
4	P04IINV	Interrupt 12 Pin Polarity 0 = P0.4 1 = P0.4 Inverted
3	P03IINV	Interrupt 11 Pin Polarity 0 = P0.3 1 = P0.3 Inverted
2	P02IINV	Interrupt 10 Pin Polarity 0 = P0.2 1 = P0.2 Inverted
1	P01IINV	Interrupt 9 Pin Polarity 0 = P0.1 1 = P0.1 Inverted
0	P00IINV	Interrupt 8 Pin Polarity 0 = P0.0 1 = P0.0 Inverted

14.5 Pin Interrupt Sensitivity Setting

The pin interrupt can be configured as level sensitive or edge triggered. The pin interrupt sensitivity is set via the IPINSENSx and IPININVx registers. The following table summarizes the pin interrupt trigger condition settings for IPINSENx and IPININVx.

TABLE 165:IMPACT OF PIN SENSITIVITY AND PIN INVERSION SETTING ON PIN INTERRUPT

Pin Sensitivity	Pin Inversion	Interrupt Condition
0	0	High level on pin
0	1	Low level on pin
1	0	Rising edge on pin
1	1	Falling edge on pin

The following tables provide the bit definitions for the IPINSENS1 and IPINSENS2 registers. It is assumed that the corresponding IPININVx bit is set to 0. If the corresponding IPININVx bit is set to 1, the corresponding interrupt event will be inverted.

TABLE 166:INTERRUPT PIN SENSITIVITY 1 REGISTER - IPINSENS1 SFR E6H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	P37ISENS	Interrupt 7 Pin Sensitivity (IPININV1.7 = 0) 0 = P3.7 High Level 1 = P3.7 Rising Edge
6	P36ISENS	Interrupt 6 Pin Sensitivity (IPININV1.6 = 0) 0 = P3.6 High Level 1 = P3.6 Rising Edge
5	P35ISENS	Interrupt 5 Pin Sensitivity (IPININV1.5 = 0) 0 = P3.5 High Level 1 = P3.5 Rising Edge
4	P34ISENS	Interrupt 4 Pin Sensitivity (IPININV1.4 = 0) 0 = P3.4 High Level 1 = P3.4 Rising Edge
3	P31ISENS	Interrupt 3 Pin Sensitivity (IPININV1.3 = 0) 0 = P3.1 High Level 1 = P3.1 Rising Edge
2	P30ISENS	Interrupt 2 Pin Sensitivity (IPININV1.2 = 0) 0 = P3.0 High Level 1 = P3.0 Rising Edge
1	P33ISENS	Interrupt 1 Pin Sensitivity (IPININV1.1 = 0) 0 = P3.3 High Level 1 = P3.3 Rising Edge
0	P32ISENS	Interrupt 0 Pin Sensitivity (IPININV1.0 = 0) 0 = P3.2 High Level 1 = P3.2 Rising Edge

TABLE 167:INTERRUPT PIN SENSITIVITY 2 REGISTER - IPINSENS2 SFR E7H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	P07ISENS	Interrupt 7 Pin Sensitivity (IPININV2.7 = 0) 0 = P0.7 High Level 1 = P0.7 Rising Edge
6	P06ISENS	Interrupt 6 Pin Sensitivity (IPININV2.6 = 0) 0 = P0.6 High Level 1 = P0.6 Rising Edge
5	P05ISENS	Interrupt 5 Pin Sensitivity (IPININV2.5 = 0) 0 = P0.5 High Level 1 = P0.5 Rising Edge
4	P04ISENS	Interrupt 4 Pin Sensitivity (IPININV2.4 = 0) 0 = P0.4 High Level 1 = P0.4 Rising Edge
3	P03ISENS	Interrupt 3 Pin Sensitivity (IPININV2.3 = 0) 0 = P0.3 High Level 1 = P0.3 Rising Edge
2	P02ISENS	Interrupt 2 pin Sensitivity (IPININV2.2 = 0) 0 = P0.2 High Level 1 = P0.2 Rising Edge
1	P01ISENS	Interrupt 1 Pin Sensitivity (IPININV2.1 = 0) 0 = P0.1 High Level 1 = P0.1 Rising Edge
0	P00ISENS	Interrupt 0 Pin Sensitivity (IPININV2.0 = 0) 0 = P0.0 High Level 1 = P0.0 Rising Edge

14.6 Interrupt Pin Flags

For each pin interrupt there is an interrupt flag that can be monitored. When the selected interrupt event is detected on a given pin, the corresponding pin interrupt flag is set to 1 by the system.

The interrupt pin flags are automatically cleared when the RETI (return from interrupt) instruction is executed. They can also be cleared by the software at any time.

The pin interrupt flags can be monitored via the software, even if the corresponding pin interrupt is not activated. If all the corresponding interrupts are routed to modules and all the interrupts are disabled, the IPINFLAGx registers can be used as general purpose scratchpad registers. However this is not recommended.

TABLE 168:INTERRUPT PIN FLAG 1 REGISTER - IPINFLAG1 SFR B8H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	P37IF	Interrupt 7 Pin Flag Set to 1 if P3.7 pin Interrupt occurs
6	P36IF	Interrupt 6 Pin Flag Set to 1 if P3.6 pin Interrupt occurs
5	P35IF	Interrupt 5 Pin Flag Set to 1 if P3.5 pin Interrupt occurs
4	P34IF	Interrupt 4 Pin Flag Set to 1 if P3.4 pin Interrupt occurs
3	P31IF	Interrupt 3 Pin Flag Set to 1 if P3.1 pin Interrupt occurs
2	P30IF	Interrupt 2 Pin Flag Set to 1 if P3.0 pin Interrupt occurs
1	P33IF	Interrupt 1 Pin Flag Set to 1 if P3.3 pin Interrupt occurs
0	P32IF	Interrupt 0 Pin Flag Set to 1 if P3.2 pin Interrupt occurs

TABLE 169:INTERRUPT PIN FLAG 2 REGISTER - IPINFLAG2 SFR D8H

7	6	5	4	3	2	1	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7	P07IF	Interrupt 15 Pin Flag Set to 1 if P0.7 pin Interrupt occurs
6	P06IF	Interrupt 14 Pin Flag Set to 1 if P0.6 pin Interrupt occurs
5	P05IF	Interrupt 13 Pin Flag Set to 1 if P0.5 pin Interrupt occurs
4	P04IF	Interrupt 12 Pin Flag Set to 1 if P0.4 pin Interrupt occurs
3	P03IF	Interrupt 11 Pin Flag Set to 1 if P0.3 pin Interrupt occurs
2	P02IF	Interrupt 10 Pin Flag Set to 1 if P0.2 pin Interrupt occurs
1	P01IF	Interrupt 9 Pin Flag Set to 1 if P0.1 pin Interrupt occurs
0	P00IF	Interrupt 8 Pin Flag Set to 1 if P0.0 pin Interrupt occurs

15 VRS51L3074 JTAG Interface

The VRS51L3074 includes a JTAG interface that enables programming of the on-board Flash as well as code debugging. In order to free up as many I/Os as possible, the JTAG interface pins are shared with regular I/O pins that can be used as general I/Os when the JTAG interface is not being used.

The JTAG interface is mapped into the following pins:

TABLE 170: JTAG INTERFACE PIN MAPPING

JTAG Pin	Function	Corresponding Pin
TDI	JTAG Data Input	P4.3
TDO	JTAG Data Output	P4.2
CM0	Chip Mode 0	ALE
TMS	Test Mode Select	P4.1
TCK	JTAG Clock	P2.7

Activation of the JTAG interface is controlled by the CM0/ALE pin. The CM0/ALE pin includes an internal pull-up resistor. When the CM0 pin is held at a logic low and a power-on reset is performed, the JTAG interface is activated.

15.1 Impact of JTAG interface activation

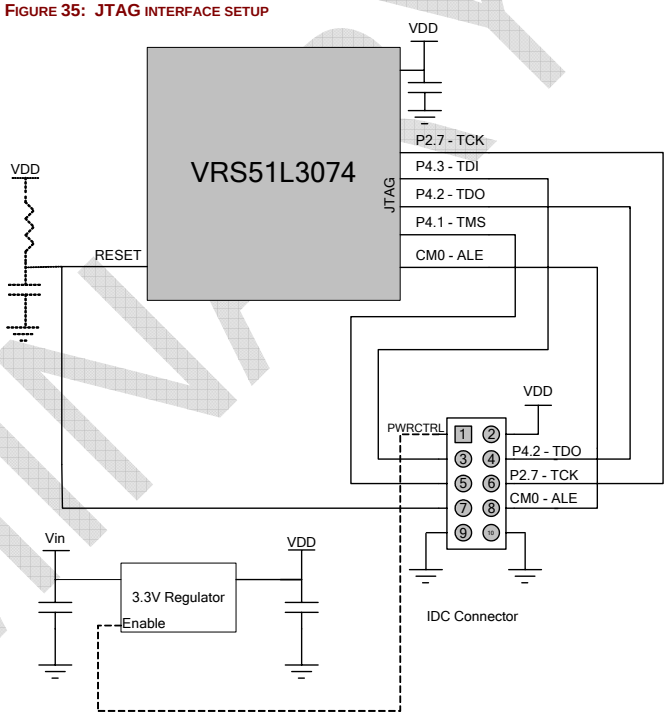
When the JTAG interface is activated, it has the following consequences on VRS51L3074 operation:

- The PWM 7 output is deactivated. The PWM7 module can still be active.
- The P2.7, P4.3, P4.2, P4.1 I/O pins are deactivated.
- The ALE pin is reserved for the JTAG interface. To efficiently debug code accessing the external SRAM memory, place a 1k Ohms resistor in the path of CM0 to the JTAG interface module.

15.2 Board Level JTAG Interface Implementation

To perform in-circuit programming and debugging of the VRS51L3074, access to the device's JTAG port should be provided at the board level. The following figure demonstrates a typical setup for JTAG port access.

FIGURE 35: JTAG INTERFACE SETUP



The configuration of the IDC connector shown in Figure 35 matches that on the Versa-JTAG interface IDC connector.

Please note that if the target PCB's regulator includes a power control feature, the power control line can be routed to the JTAG IDC connector, enabling the Versa-JTAG to control the target board power during device programming and in-circuit debugging. The other option is to leave the PWRCTRL pin of the IDC connector unconnected.

For the RESET control line, the presence of an external RC reset circuit is optional.

15.3 VRS51L3074 Debugger

The VRS51L3074 includes advanced debugging features that enable real-time, in-circuit debugging and emulation via the JTAG interface. When the VRS51L3074 debugger is activated, the upper 1024 bytes of the Flash memory are not available for user program.

The VRS51L3074 debugger is intended to be used in conjunction with the Versa Ware JTAG software, developed by Ramtron. This software provides an easy-to-use interface for device programming and in-circuit debugging. For more information on the VRS51L3074 debugger's features and use, please consult the Versa Ware JTAG user guide.

16 Flash Programming Interface (FPI)

The FPI module allows the processor to perform in-application management of the Flash memory content. The following operations are supported by the FPI module :

- Mass Erase
- Page Erase
- Byte Write

Six SFR registers are associated with the FPI module operation, as shown in the table below:

TABLE 171: FLASH PROGRAMMING INTERFACE REGISTERS

SFR	Name	Function	Reset Value
E9h	FPICONFIG	Configures the FPI operations	34h
EAh	FPIADDRL	Address for operation (lower byte)	00h
EBh	FPIADDRH	Address for operation (upper byte)	00h
ECh	FPIDATAL	Data to write	00h
EDh	FPIDATAH	Upper byte of data to write	00h
EEh	FPICLKSPD	Clock speed during FPI operations	00h

The FPI module is activated by setting bit 0 of the PERIPHEN2 register. There are two ways to perform read and write operations to the Flash using the FPI module: the standard 8-bit mode, which writes 1 byte at a time and an extended 16-bit mode, which writes 2 bytes at a time (1 word), effectively doubling the writing speed. In addition, whenever a write or read is performed, the address is incremented automatically by the FPI module, saving processor cycles and code space.

16.1 FPI Configuration Register

Flash operations are activated via the FPI configuration register. The following table describes the FPI configuration register:

TABLE 172: FPI CONFIGURATION REGISTER - FPICONFIG SFR E9H

7	6	5	4	3	2	1	0
R	R	R	R	R/W	R/W	R/W	R/W
0	0	1	1	0	1	0	0

Bit	Mnemonic	Description
7:6	FPILOCK[1:0]	These bits indicate the stage of the unlock operation: 00 : IAP protection on (no unlock steps done) 01 : IAP first unlock step done: FPI_DATA_LO received 0xAA 10 : IAP protection off: second step done (FPI_DATA_LO received 0x55) 11 : Disables write/erase operations until the next system reset. This occurs if a wrong sequence is used.
5	FPIIDLE	Indicates that the FPI is idle
4	FPIRDY	Indicates that the FPI is idle in all modes except "write byte" mode, in which the double buffer is ready for a new value
3	RESERVED	Keep this bit at 0
2	FPI8BIT	0 = FPI operates in 16-bit mode 1 = FPI operates in 8-bit mode
0	FPITASK[1:0]	Operation: 00: Read Mode 01: Mass Erase 10: Page Erase 11: Write (Writing to FPIDATAL starts operation) Note that actions are only started if FPIRDY is high, otherwise the action is cancelled

16.2 FPI Flash Address and Data Registers

The FPIADDRH and FPIADDRL registers are used to specify the address where the IAP function will be performed.

TABLE 173: FPI ADDRESS HIGH FPIADDRH SFR EBH

7	6	5	4	3	2	1	0
R/W, Reset = 0x00							
FPIADDR[15:8]							

The FPIADDRH register contains the MSB of the destination address. For page erase operations, it contains the page number where page erase operations are performed.

TABLE 174:FPI ADDRESS LOW -FPIADDRL SFR EAH

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
R/W							
FPIADDR[7:0]							

The FPIADDRL register contains the LSB of the destination address where the operation is performed. For page erase it must contain the value 0x00.

The FPIDATAH and FPIDATAL registers contain the data byte(s) required to perform the FPI function.

TABLE 175: FPI DATA HIGH - FPIDATAH SFR EDH

7	6	5	4	3	2	1	0
R/W, Reset = 0x00							
FPIDATA[15:8]							

When Read: MSB of last word read[15:8] from Flash
When Write: Byte[15:8] to write in Flash

TABLE 176:FPI DATA LOW - FPIDATAL SFR ECH

7	6	5	4	3	2	1	0
R/W, Reset = 0x00							
FPIDATA[7:0]							

Read: Last read byte[7:0] from Flash

Writing to this byte in 'FPI write mode' triggers the FPI state machine to start the write action.

16.3 FPI Clock Speed Control Register

The FPI clock speed control register sets the FPI module to an optimal speed based on the speed of the system clock.

TABLE 177:FPI CLOCK SPEED CONTROL REGISTER - FPICLKSPD SFR EEH

7	6	5	4	3	2	1	0
R	R	R	R	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Bit	Mnemonic	Description
7:4	Unused	
3:0	FPICLKSPD [3:0]	Specifies speed of the system clock entering the FPI module Frequency range: 0000 : 20MHz to 40 MHz 0001 : 10MHz to 20 MHz 0010 : 5MHz to 10 MHz 0011 : 2.5MHz to 5 MHz 0100 : 1.25MHz to 2.5 MHz 0101 : 625kHz to 1.25 MHz 0110 : 312.5kHz to 625 kHz 0111 : 156.25kHz to 312.5 kHz 1000 : 78.12kHz to 156.25 kHz 1001 : 39.06kHz to 78.125 kHz 1010 : 19.53kHz to 39.0625 kHz Others : 9.76kHz to 19.53125 kHz

Use the settings found in the following table when using the FPI at a speed other than the nominal speed of the internal oscillator.

TABLE 178: SETTING THE FPICLKSPD REGISTER

Value	Range	
	Minimum	Maximum
0 (default)	20.000 MHz	40.000 MHz
1	10.000 MHz	20.000 MHz
2	5.000 MHz	10.000 MHz
3	2.500 MHz	5.000 MHz
4	1.250 MHz	2.500 MHz
5	625.000 KHz	1.250 MHz
6	312.500 KHz	625.000 KHz
7	156.250 KHz	312.500 KHz
8	78.125 KHz	156.250 KHz
9	39.063 KHz	78.125 KHz
10	19.531 KHz	39.063 KHz
Other	9.766 KHz	19.531 KHz

The FPICLKSPD register must be set to the corresponding system clock speed for proper operation of the FPI module. For example, a 20.0 MHz clock requires FPICLKSPD to be set to 1, while a 20.1 MHz clock requires FPICLKSPD to be set to 0. If FPICLKSPD is set incorrectly, the Flash write operation may not process correctly, causing data corruption.

16.4 Using the FPI Interface

16.4.1 Write protection

The VRS51L3074 provides a safety mechanism to prevent accidental writing or erasing of the Flash. The following sequence must be written to the FPIDATAL register to unlock the VRS51L3074 each time a write is performed.

FPIDATAL ← AAh
FPIDATAL ← 55h

Not performing the above sequence will lock the FPI module until a reset of the VRS51L3074 is performed.

Bit 7 and 6 of the FPICONFIG provide the status of the FPI write protection circuitry.

16.4.2 FPIIDLE

This bit indicates whether the previous action is complete and the FPI is idle. The FPIIDLE bit must be checked before performing any FPI operation, to ensure that the module is ready.

16.4.3 FPIRDY

When writing a stream of bytes or words, this bit indicates whether the FPI is ready for the next write. Note that AAh then 55h must first be written in order to unlock the FPI module.

16.4.4 FPI8BIT

The FPI8BIT bit of the FPICONFIG register defines whether the FPI module read and write operations will be performed in 8 or 16-bit format. When the FPI8BIT bit is set to 1, the FPI module will operate in 8-bit mode. The 16-bit address of the Flash memory, where the FPI operation will be performed, is defined by the value of the FPIADDRH and FPIADDRL registers.

When the FPI module is used to write data into the Flash memory, the FPIDATAL register holds the value of the data to be written. When the FPI module is used to read the Flash, the read value is returned via the FPIDATAL register.

When the FPI8BIT bit is cleared, the FPI module will operate in 16-bit mode. In this case, the address range is defined by a 15-bit address (0000h to 7FFFh) and must be written into the FPIADDRH and FPIADDRL registers.

When a 16-bit FPI write operation is performed, the 16-bit data must be stored in the FPIDATAH and FPIDATAL registers. When a Flash memory read operation is performed, the 16-bit data will be returned to the FPIDATAH and FPIDATAL registers.

16.5 Performing a Read

There are three ways to read directly from the VRS51L3074 Flash memory:

1. Use the MOVC instruction
2. Use the FPI in 8-bit mode
3. Use the FPI in 16-bit mode

It may be preferable to use the FPI over the MOVC instruction, because some compilers will optimize code that repeatedly checks the Flash. To perform a read, perform the following steps:

- Make sure the FPI module is enabled
- Set FPIADDRH and FPIADDRL to the appropriate address (see section 1.1.4)
- Write 00000X00 to the FPICONFIG register, where X = 1 if reading in 8-bit mode, and X = 0 if reading in 16-bit mode
- Loop until FPIIDLE is raised
- Get the results from FPIDATAH and FPIDATAL if in 16-bit mode, or from FPIDATAL if in 8-bit mode

16.5.1 FPI Flash Read in 8-Bit Mode Example

The following code sequence follows the above algorithm to read address ABCDh in 8-bit mode:

```
ORL PERHIPHEN2, #1 ; Enable FPI
MOV FPIADDRH, #0ABh ; Move in upper address
MOV FPIADDRL, #0CDh ; Move in lower address

MOV FPICONFIG, #004h ; Trigger the read in 8-bit mode

Wait:
   MOV A, FPICONFIG ; Get the FPI status
   JNB ACC.7, Wait ; Jump if not ready

; The read is now done. The result in FPIDATAL
```

16.5.2 FPI Flash Read in 16-Bit Mode Example

The following code sequence will read 16 bits from address ABCD:

```
#include <VRS51L3074.h>
unsigned char ucupper;
unsigned char uclower;

void readFPI(int address)
{
   unsigned char result;

PERIPHEN2 |= 1; /* Enable FPI */
FPIADDRH = (unsigned char) (address >> 8); /* Upper address */
FPIADDRL = (unsigned char) address; /* Lower address – automatically truncates */
FPICONFIG = 0; /* Trigger the read */
do
{
   result = FPICONFIG & 0x20; /* Check for the FPI_IDLE bit */
}
while(!result)
ucupper = FPIDATAH;
uclower = FPIDATAL;
}

void main()
{
   /*** SOME CODE***/
   readFPI(0x55e6);   /* This is address ABCD converted to 16 bit addressing */

/*** SOME CODE***/
   while(1);
}
```

16.6 Erasing Flash

16.6.1 Page Erase

When storing nonvolatile data, it is necessary to erase the Flash before writing to it. Programming is done by byte or word boundary, while erase is done by page boundary. A page is a contiguous block of 512 addresses. Page numbers can be calculated from the following formula:

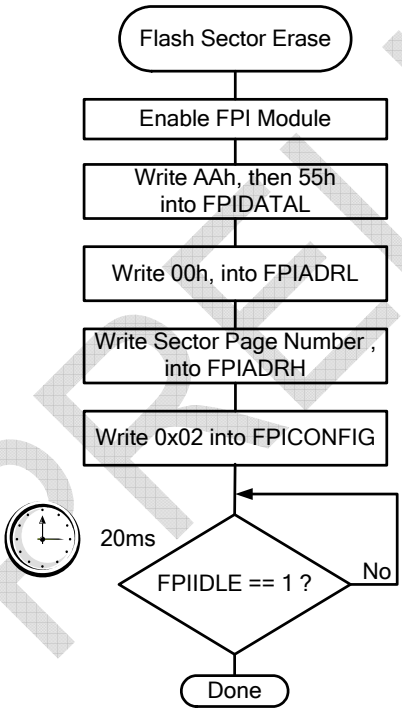
Page = address / 512

Page 0 contains all the addresses from 0000h to 01FFh, page 1 contains all the addresses from 0200h to 03FFh, and so on. There are 128 Flash pages on the VRS51L3074 (64KB Flash).

To erase a page, follow these steps:

1. Ensure that the FPI module is enabled
2. Write AAh to the FPIDATAL register
3. Write 55h to the FPIDATAL register
4. Write 0 to the FPIADDRL register
5. Write the page number to the FPIADDRH register
6. Write 2 to the FPICONFIG register
7. Wait for FPIIDLE to go high

FIGURE 36: FPI FLASH SECTOR ERASE ALGORITHM



16.6.2 FPI Page Erase Example

This code sequence will erase page 64:

```
ORL PERHIPHEN2, #1 ; Enable FPI
MOV FPIDATAL, #0AAh ; UNLOCK 1
MOV FPIDATAL, #055h ; UNLOCK 2
MOV FPIADDRL, #0 ; Move in 0
MOV FPIADDRH, #64 ; Move in page number
MOV FPICONFIG, #2 ; Trigger the page erase

Wait:
   MOV A, FPICONFIG ; Get the FPI status
   JNB ACC.7, Wait ; Jump if not ready
; The page is now erased
```

16.6.3 Mass Erase

It is possible to completely erase the Flash memory from within a program. To do so, the following steps must be performed:

1. Make sure that the FPI module is enabled
2. Write AAh to the FPIDATAL register
3. Write 55h to the FPIDATAL register
4. Write 1 to the FPICONFIG register
5. If still possible, wait for FPIIDLE to go high

Warning: At this point, the Flash should be totally erased. If running from external memory, make sure the program is copied back to its locations in Flash with write commands. Step 5 can only be performed if executing code from external SRAM.

16.7 Writing to the Flash

There are two methods to write to the Flash:

- o 8-bit double buffered
- o 16-bit double buffered

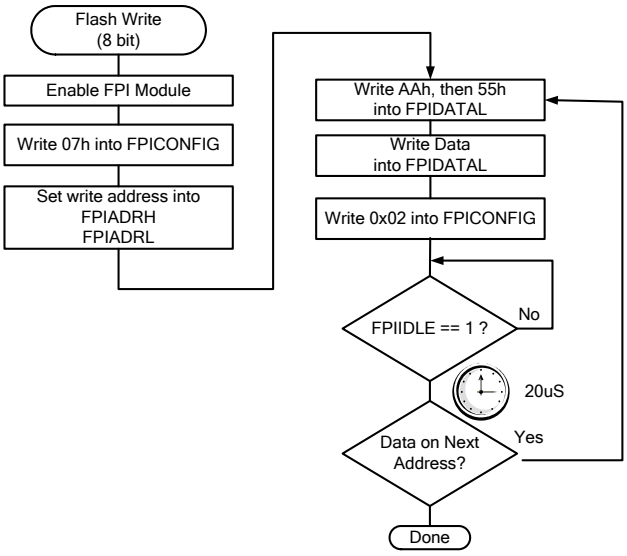
Depending on the complexity and the amount of Flash to be written, one mode may be more efficient than the other: 8-bit mode is more suited to programming a few bytes of data, while 16-bit mode is more suited to memory dumping.

16.7.1 Writing the Flash in 8-bit mode

Follow the steps below to write in 8-bit mode:

1. Make sure the FPI module is enabled
2. Write 7 to the FPICONFIG register
3. Set FPIADDRH and FPIADDRL to the appropriate addresses
4. Write AAh to the FPIDATAL register
5. Write 55h to the FPIDATAL register
6. Write data to the FPIDATAL register (this triggers the operation)
7. If complete, wait for FPIDLE to go high. If there are more bytes to be written at a different address, return to step 3. If the next address is contiguous, go to step 4 instead.

FIGURE 37: FPI FLASH BYTE WRITE ALGORITHM



Note that the address the data is written to will be automatically incremented for the next byte. As such, the address only needs to be written once per data stream (assuming that a contiguous block is written), as shown in the following example.

16.7.2 FPI Flash Write in 8-Bit Mode Example

```
//***********************************************
//* FPI Flash Write 8bit Mode Example  *
//***********************************************
#include <VRS51L3074.h>
/*
This function uses the FPI module to write a null terminated string to flash
*/
void copy_to_Flash(int address, char *str)
{
 unsigned char ready; /* Is the FPI idle? */

 PERIPHEN2 |= 1;     /* Enable FPI */

/* Upper address */
 FPIADRH = (unsigned char) (address >> 8);
/* Lower address - automatically truncates */
 FPIADRL = (unsigned char) address;
 FPICONFIG = 7;  /* Trigger the write
   in 8 bit mode */

  while(*str)   /* while not null */
    {
    FPIDATAL = 0xaa; /* 1st step unlock */
    FPIDATAL = 0x55; /* 2nd step unlock */
    FPIDATAL = (unsigned char)(*str);

    /* Wait for the buffer to be ready */
/* The operation is not finished, check for FPI_READY */
    do
       {
       ready = FPICONFIG & 0x10;
       }while(!ready);
    str++;
    }

/* Null character encountered, write an
   additional 0 to memory */
     FPIDATAL = 0xaa; /* 1st step unlock */
  FPIDATAL = 0x55; /* 2nd step unlock */
  FPIDATAL = 0;    /* End in null - this avoids
           having  to pass the string
           length */
/* The operation is finished, check for FPI_IDLE instead of FPI_READY */
    do
{
    ready = FPICONFIG & 0x20;
    }while(!ready);

  return;
}
void main(void)
{

 /*** CODE ***/
  copy_to_Flash(0x3000, "Ramtron Inc");
  copy_to_Flash(0x4000, "Microsystems connecting two worlds");

  /*** CODE ***/

  while(1);
}
```

16.7.3 Writing to the Flash in 16-Bit Mode

Follow the steps below to write in 16-bit mode:

1. Make sure the FPI module is enabled
2. Write 3 to the FPICONFIG register
3. Set FPIADDRH and FPIADDRL to the appropriate addresses (remember to convert to 16-bit addressing)
4. Write AAh to the FPIDATAL register
5. Write 55h to the FPIDATAL register
6. Write data to the FPIDATAL register (this triggers the operation)
7. If complete, wait for FPI_IDLE to go high. If there are more bytes to be written at a different address, return to step 3. If the next address is contiguous, go to step 4 instead

Note that the address the data is written to will be automatically incremented for the next byte As such, the address only needs to be set once per data stream (assuming a contiguous region is written), as shown in the following example.

16.7.4 FPI Flash Write in 16-Bit Mode Example

This routine copies 512 bytes (1 page) of external SRAM to the Flash memory at address E000h + XRAM. The R0 and R1 registers contain the starting address of the page to copy.

```
//***********************************************
//* FPI Flash Write 16-bit Mode Example  *
//***********************************************

WRITE_PAGE:

  PUSH DPH0             ;PUSH THE DATA POINTER
  PUSH DPL0
  PUSH ACC            ;PUSH THE VAR. TO BE USED
  PUSH B
  MOV ACC, R2
  PUSH ACC

  MOV DPH0, R1      ;LOAD THE DATA POINTER
  MOV DPL0, R0
  MOV R2, #255      ;LOOP COUNTER (511 BYTES)
  ORL PERHIPHEN2, #1   ;ENABLE FPI MODULE
  MOV FPICONFIG, #3     ;ENABLE WRITING IN 16 BIT ;MODE
; SET THE ADDRESS MUST BE 16 BITS (ADDRESS / 2)
```

```
  CLR C                         ;CLEAR THE CARRY FLAG
  MOV A, R1
  RRC A                         ;CHECK IF THERE WILL BE A CARRY
  CLR A                         ;DOES NOT AFFECT CARRY BIT
  RRC A                         ;SETS A TO 80h IF R1 WAS ODD, OR
                                   ;KEEPS IT 0


MOV FPIADRL, A          ;SET LOWER ADDRESS
  MOV A, R1
  RR A                  ;DIVIDE ADDRESS BY 2
  ADD A, #7             ;ADDS E000H TO THE ADDRESS
                           ;(E000 / 2 = 7000)

  MOV FPIADRH, A ; SET UPPER ADDRESS

  WRITE_PAGE_LOOP:

    MOV FPIDATAL, #0AAh     ;UNLOCK STEP 1
    MOV FPIDATAL, #055h     ;UNLOCK STEP 2
    MOVX A, @DPTR
    MOV B, A
    INC DPTR                    ;NEXT BYTE

    MOVX A, @DPTR
    INC DPTR                    ;NEXT BYTE
    MOV FPIDATAH, A         ;SET THE UPPER VALUE
    MOV FPIDATAL, B             ;SET THE LOWER VALUE
;AND START THE WRITE

WRITE_PAGE_LOOP_WAIT:
    MOV A, FPICONFIG        ;CHECK TO SEE IF THE
              ;BUFFER IS READY
    ;JUMP IF FPI_READY IS NOT HIGH
JNB ACC.4                       ,WRITE_PAGE_LOOP_WAIT

DJNZ R2                         ,WRITE_PAGE_LOOP

          ;NOW WRITE THE LAST WORD (BYTE 511 AND 512)

    MOV FPIDATAL, #0AAh    ;UNLOCK STEP 1
    MOV FPIDATAL, #055h    ;UNLOCK STEP 2

    MOVX A, @DPTR
    MOV B, A
    INC DPTR                    ;NEXT BYTE

    MOVX A, @DPTR
    INC DPTR                    ;NEXT BYTE
;(not necessary)
    MOV FPIDATAH, A         ;SET THE UPPER VALUE
    MOV FPIDATAL, B             ;SET THE LOWER VALUE
;AND START THE WRITE
WRITE_PAGE_LAST_WAIT:
    MOV A, FPICONFIG        ;CHECK TO SEE IF THE
                                ;BUFFER IS READY
 JUMP IF FPI_IDLE IS NOT HIGH (LAST WORD)
JNB ACC.5                       , WRITE_PAGE_LOOP_WAIT

;RESTORE VARIABLES USED
POP B
POP ACC
    MOV R3, ACC
    POP ACC
    POP DPL0
    POP DPH0
    RET                         ;RETURN TO CALLER
```

16.8 Tips on Using the FPI Interface

The following tips can be used to get the most out of the IAP features on the VRS51L3074.

- Shorter programming time can be achieved if the FPI Flash write routines are run from the 4KB external SRAM, as the circuitry that reads instructions from the Flash does not interfere with the FPI module.
- The Flash must be erased before reprogramming, and the same value should not be written more than once to the same Flash address, unless an erase cycle is performed in between writes.
- To maximize the endurance of the VRS51L3074 Flash memory, FPI Flash page erase operations should be done sparingly.
- The FPI mass erase function will erase the entire VRS51L3074 Flash memory, including code already programmed.
- IAP can be performed even if the Flash protection is enabled. It is the responsibility of the programmer not to reveal the Flash information of a secured device via the IAP.
- When write operations are performed at the boundaries of two contiguous blocks of memory, the address will automatically increment to the next byte/word after a write cycle. This can save processor cycles.
- The FPI read can be used to perform Flash memory reads, however using the MOVC instruction is more efficient.
- Make sure that the location being written to does not interfere with the program running in the Flash.

17 Crystal Consideration

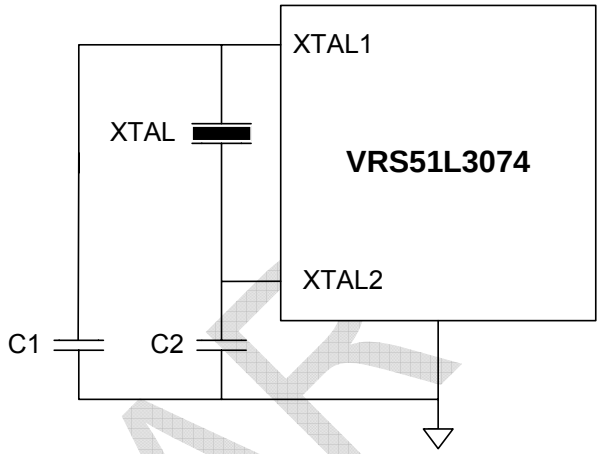
By default, the VRS51L3074 derives its clock from its internal oscillator. It is also possible to use external crystal for the VRS51L3074 clock source. The crystal connected to the VRS51L3074 oscillator input should be parallel cut type, operating in fundamental mode.

The addition of 15 to 20pF load capacitors is recommended. See the following figure for a connection diagram.

Note: Oscillator circuits may differ with different crystals or ceramic resonators in higher oscillation frequency. Crystals or ceramic resonator characteristics may also vary from one manufacturer to another.

The user should review the technical literature associated with specific crystal or ceramic resonator s or contact the manufacturer to select the appropriate values for the external components.

FIGURE 38: VRS51L3074 EXTERNAL CRYSTAL OSCILLATOR CONFIGURATION



18 Operating Conditions

18.1 Absolute Maximum Ratings

Parameter	Min.	Max.	Unit	Notes
Supply voltage input (VDD – VSS)	0	3.6	V	
I/O input voltage all except P4.6 & P4.7	-0.5V	5.5V	V	
I/O input voltage P4.6 & P4.7 only	VCC-0.5	VCC+0.5	V	
Maximum I/O current (sink/source) QFP-64 package		100	mA	Preliminary
Storage temperature	-55	125	ºC	

18.2 Nominal operating conditions

TABLE 179: OPERATING CONDITIONS

Symbol	Description	Min.	Typ.	Max.	Unit	Remarks
TA	Operating temperature	-40		+85	ºC	
VCCV	Supply voltage	3.1	3.3	3.6	V	
Fextosc 40	Ext. Oscillator Frequency	0	-	40	MHz	
FextCY	Ext. Crystal frequency	4		40	MHz	
		32		100	KHz	
Internal Oscillator Operating frequency		39.7	40	40.3	MHz	
Internal Oscillator temperature stability			+/-2		%	0 to +70ºC
Internal Oscillator temperature stability			+/-3		%	-40 to +85ºC
FRAM data retention		45	-	-	Years	
FRAM byte Write		1.1			uS	
FRAM Byte Read		0.4			uS	
Flash Endurance(Erase / Write cycles)		20K			Cycles	
Flash Data retention		100			Years	Unser room temperature
Flash Page Erase duration		20			ms	
Flash Byte/Word programming time		20			uS	

18.3 DC Characteristics

VCC = 3.3V, Temp = 25ºC, No load on I/Os

TABLE 180: DC CHARACTERISTICS

Symbol	Parameter	Valid	Min.	Typ	Max.	Unit	Test Conditions
VIL1	Input Low Voltage	Port 0,1,2,3,4,5,6	-0.35		0.80	V	VCC=3.3V
VIL2	Input Low Voltage	RESET, XTAL1	-0.35		0.80	V	VCC=3.3V
VIH1	Input High Voltage	Port 0,1,2,3,4,5,6	2.0		5.5	V	VCC=3.3V
VI H2	Input High Voltage	RES, XTAL1	2.0		5.5	V	VCC=3.3V
VOL1	Output Low Voltage	Port 0,1,2,3,4,5,6,ALE			0.4	V	IOL = Rated I/O max current
VOH2	Output High Voltage	Port 0,1,2,3,4,5,6,ALE	2.4V	Vcc – 0.3V		V	Max Rated I/O Current
ILI	Input Leakage Current	Port 0,1,2,3,4			10	uA	(+/-)
R RES	Reset Equivalent Pull-up Resistance	RES	74	104	177	Kohm	
C-10	Pin Capacitance				10	pF	Freq=1 MHz, Ta=25°C
ICC	Supply Current	VDD	17		32	mA	Active mode, 40MHz (Int. Oscillator)
			7.9		12	mA	Active mode, 10MHz (Int. Oscillator)
			6.2		8.5	mA	Active mode 5 MHz (Ext. Crystal)
			3.6		11	mA	Idle mode, oscillator running 40MHz
				1.1		mA	OSC stop mode, 32kHz Crystal osc mode

18.4 VRS51L3074 Timings Parameters

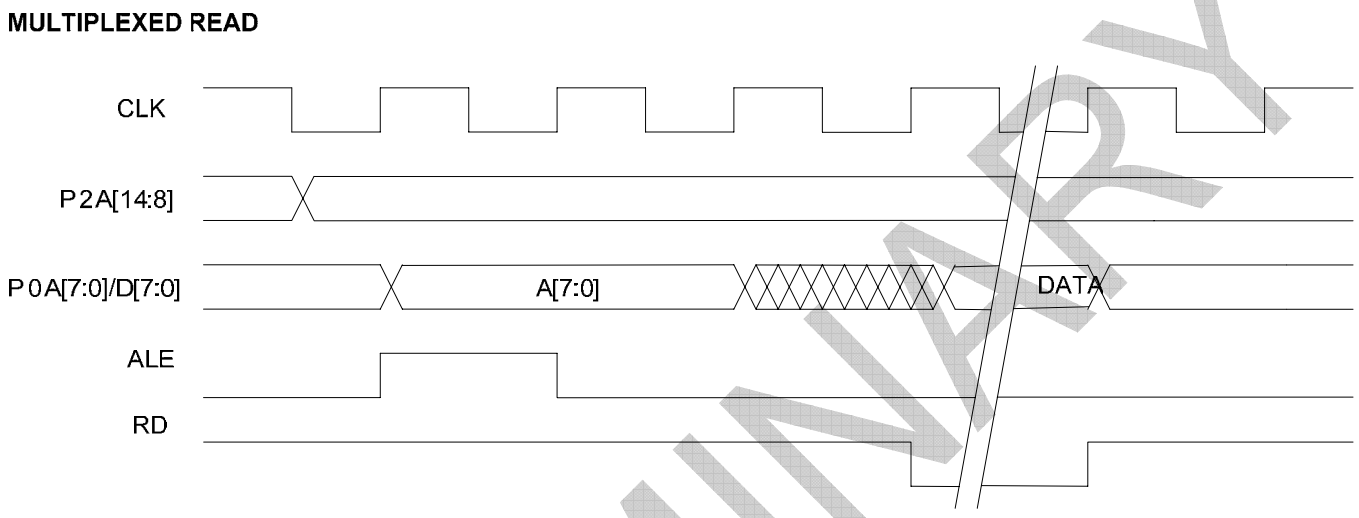
TABLE 181: AC CHARACTERISTICS

Symbol	Parameter	Variable Fosc			Unit
		Min.	Typ	Max.	
	ALE Pulse Width				nS
	Address Valid to ALE Low				nS
	Address Hold after ALE Low				nS
	ALE Low to Valid Instruction In				nS
	ALE Low to #PSEN low				nS
	#PSEN Pulse Width				nS
	#PSEN Low to Valid Instruction In				nS
	Instruction Hold after #PSEN				nS
	Instruction Float after #PSEN				nS
	Address to Valid Instruction In				nS
	#PSEN Low to Address Float				nS
	#RD Pulse Width				nS
	#WR Pulse Width				nS
	#RD Low to Valid Data In				nS
	Data Hold after #RD				nS
	Data Float after #RD				nS
	ALE Low to Valid Data In				nS
	Address to Valid Data In				nS
	ALE low to #WR High or #RD Low				nS
	Address Valid to #WR or #RD Low				nS
	Data Valid to #WR High				nS
	Data Valid to #WR Transition				nS
	Data Hold after #WR				nS
	#RD Low to Address Float				nS
	#W R or #RD High to ALE High				nS
	Clock Fall Time				nS
	Clock Low Time				nS
	Clock Rise Time				nS
	Clock High Time				nS
	Clock Period				nS

18.5 Data Memory Read Cycle Timing – Multiplexed Mode

The following diagram shows the timing of a multiplexed external data memory read cycle.

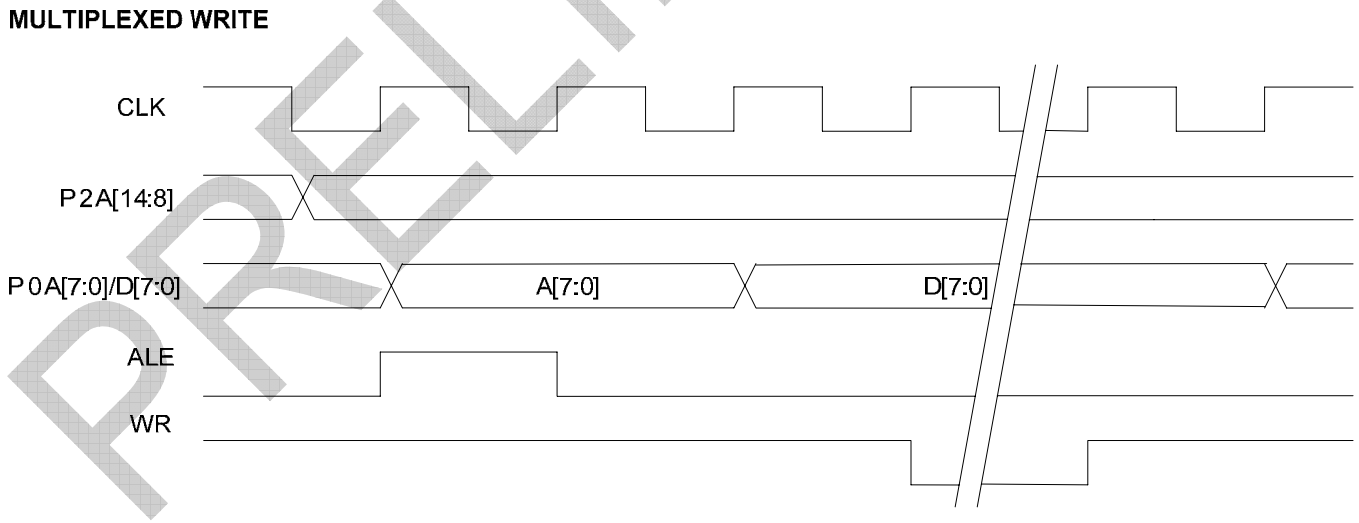
FIGURE 39: DATA MEMORY READ CYCLE TIMING



18.6 Data Memory Write cycle Timing – Multiplexed mode

The following diagram shows the timing of a multiplexed external data memory write cycle.

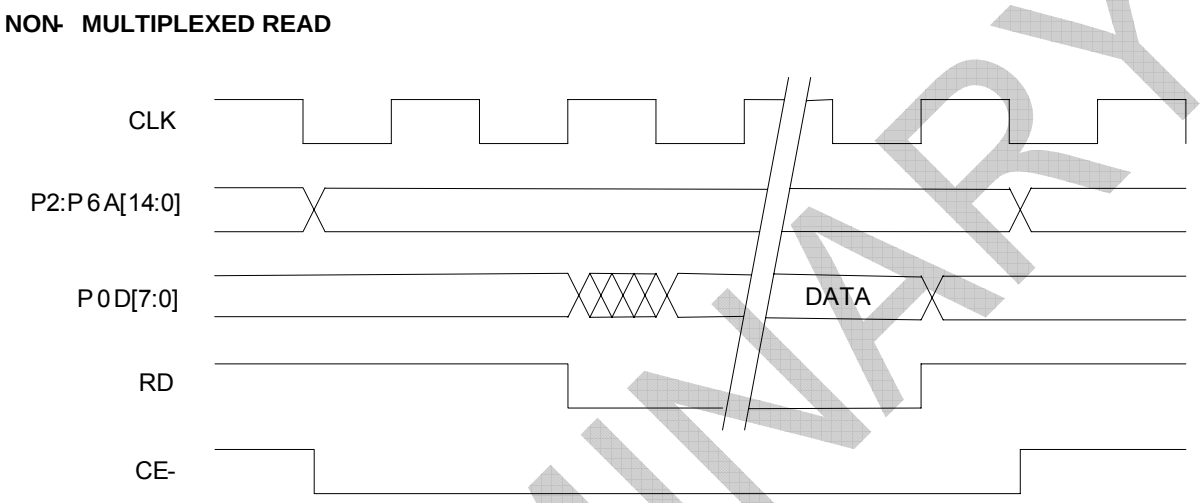
FIGURE 40: DATA MEMORY WRITE CYCLE TIMING



18.7 Data Memory Read cycle timing – Non-Multiplexed Mode

The following diagram shows the timing of a non-multiplexed external data memory read cycle.

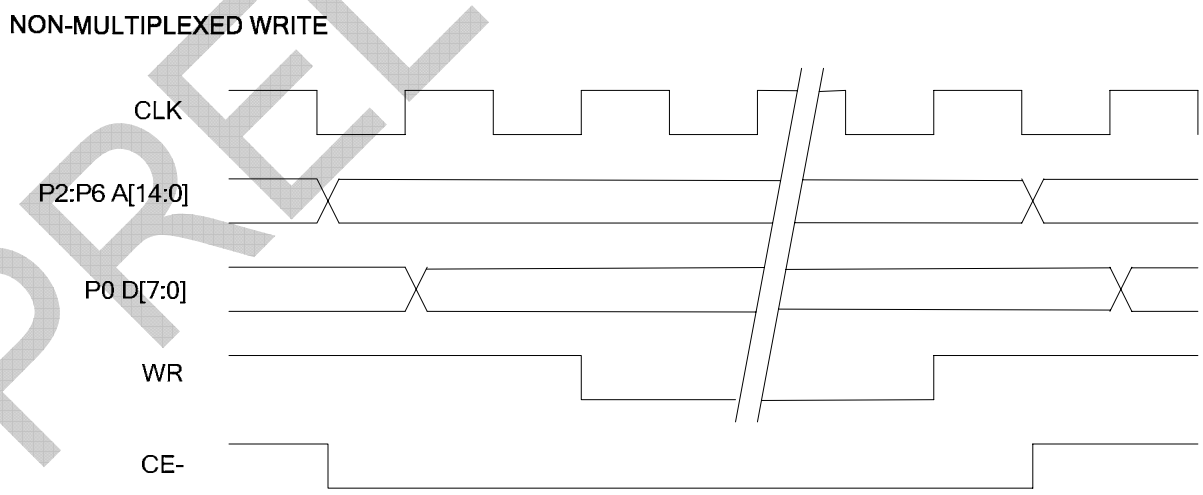
FIGURE 41: DATA MEMORY READ CYCLE TIMING



18.8 Data Memory Write Cycle Timing – Non-Multiplexed Mode

The following diagram shows the timing of a non-multiplexed external data memory write cycle.

FIGURE 42: DATA MEMORY WRITE CYCLE TIMING



18.9 Timing Requirement of the External Clock

The following diagram shows the timing of an external clock driving the VRS51L3074 input.

FIGURE 43: TIMING REQUIREMENT OF EXTERNAL CLOCK (VSS= 0.0V IS ASSUMED)

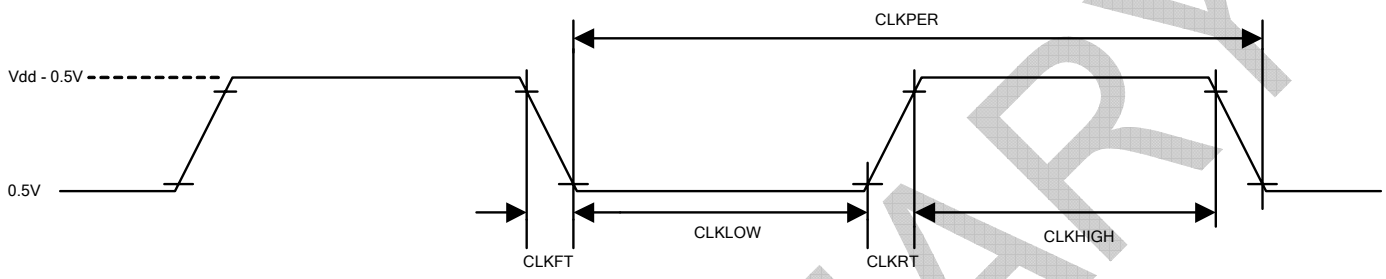


TABLE 182: EXTERNAL CLOCK TIMING REQUIREMENTS

Symbol	Parameter		Variable Fosc			Unit
			Min.	Typ	Max.	
CLKPER	Ext. clock period		25			nS
CLKLOW	Ext. clock low duration					nS
CLKHIGH	Ext. clock high duration					nS
CLKFT	Ext. clock fall time					nS
CLKRT	Ext. clock rise time					nS

18.10 VRS51L3074 QFP-64 Packages

FIGURE 44: VRS51L3074 QFP-64 PACKAGE DRAWINGS

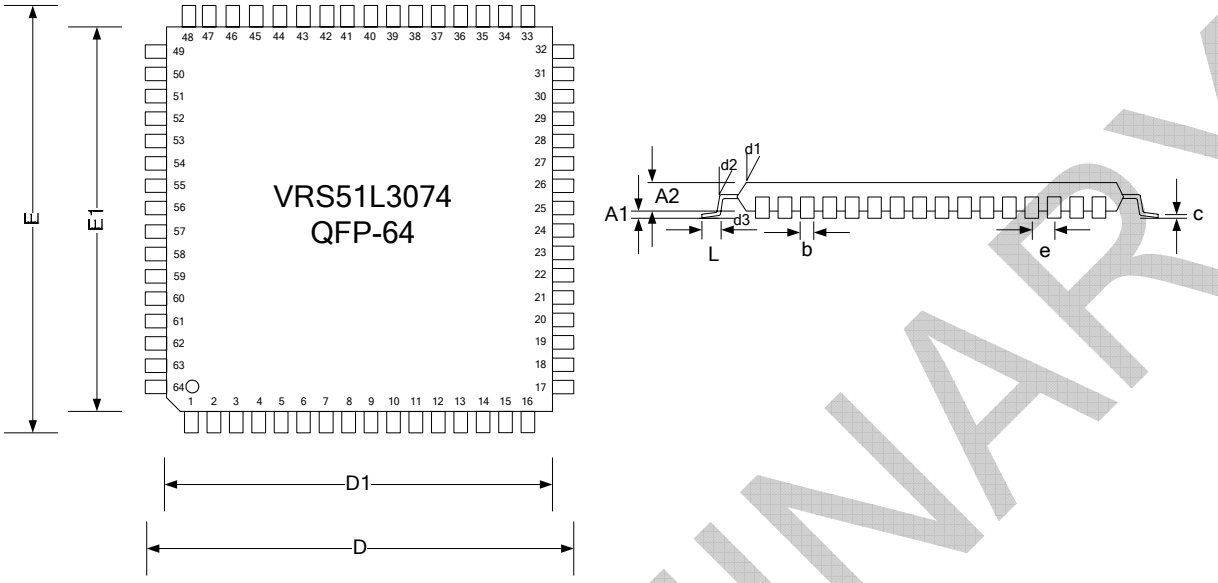
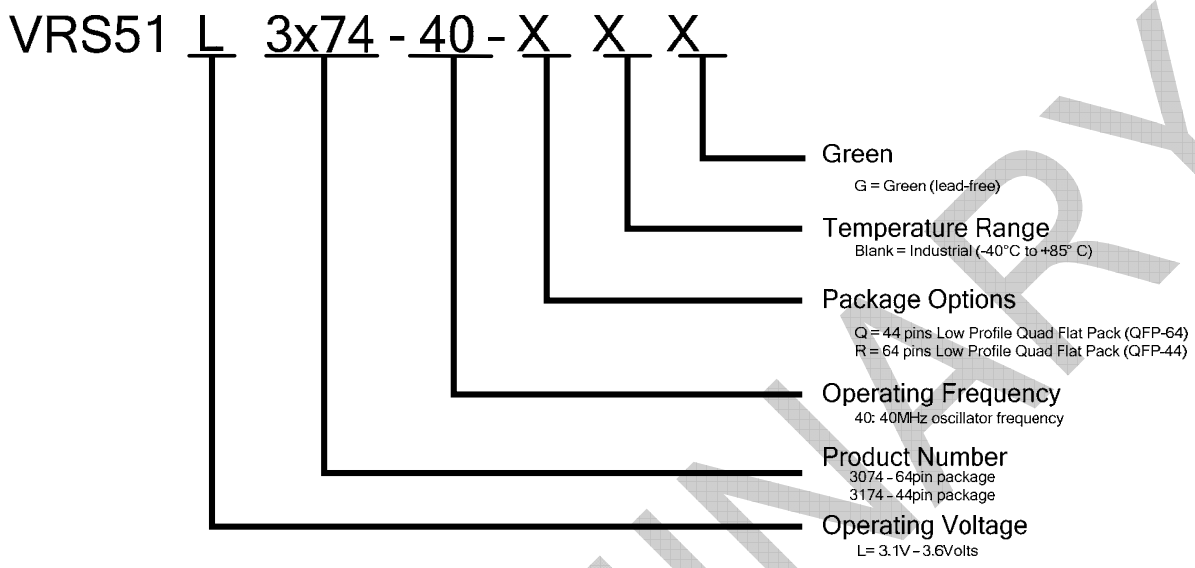


Table 183: Dimensions of QFP-64 Packages

Symbol	Description	Dimension (mm)	Tolerance (mm, º) / Notes
D	Footprint	17.2	+/- 0.25
D1	Body size	14	+/- 0.10
E	Footprint	17.2	+/- 0.25
E1	Body size	14	+/- 0.10
A1	Stand-off	0.25	Max
A2	Body thickness	2.00	
L	Lead Length	0.88	+0.15 / -0.10
b	Lead width	0.35	+/- 0.05
c	L/C thickness	0.17	Max
e	Lead pitch	0.8	
d1	Body edge angle	10º	
d2	Lead angle	6º	+/- 4º
d3	Lead angle	0º to 7º	

19 Ordering Information

19.1 Device Number Structure



19.2 VRS51L3074 Ordering Options

TABLE 184: VRS51L3074 PART NUMBERING

Device Number	Flash Size	FRAM Size	SRAM Size	Package Option	Voltage	Temperature	Frequency
VRS51L3074-40-QG	64KB	8KB	4352	QFP-64	3.1V to 3.6V	-40°C to +85°C	40MHz

* Contact Ramtron for product availability

Errata:

Engineering samples of the VRS51L3074 have an operating voltage of 3.1 to 3.6V instead of 3.0 to 3.6V. Readback of the content in the THx/TLx and RCAPxH/RCAPxL timer registers will return to 0x00 unless the corresponding timer is running or, for the timers 0 and 1, the timer gating bit is set.

Disclaimers

Right to make change - Ramtron reserves the right to make changes to its products - including circuitry, software and services - without notice at any time. Customers should obtain the most current and relevant information before placing orders.
Use in applications - Ramtron assumes no responsibility or liability for the use of any of its products, and conveys no license or title under any patent, copyright or mask work right to these products and makes no representations or warranties that these products are free from patent, copyright or mask work right infringement unless otherwise specified. Customers are responsible for product design and applications using Ramtron parts. Ramtron assumes no liability for applications assistance or customer product design.
Life support – Ramtron products are not designed for use in life support systems or devices. Ramtron customers using or selling Ramtron's products for use in such applications do so at their own risk and agree to fully indemnify Ramtron for any damages resulting from such applications.

I²C is a trademark of Koninklijke Philips Electronics NV.