FRONTGRADE

APPLICATION NOTE

UT32M0R500

32-bit Arm™ Cortex® M0+ Microcontroller - I2C Unit

12/6/2018
Version #: 1.0.0

Product Name	Manufacturer Part Number	SMD #	Device Type	Internal Pic Number
Arm Cortex M0+	UT32M0R500	5962-17212	I2C Unit	QS30

Table 1: Cross Reference of Applicable Products

1.0 Overview

The UT32M0R500 contains two Inter Integrated Circuit (I2C) controllers. I2C is a two-wire serial communications interface: a serial data line (SDA) and a serial clock line (SCL). I2C supports master or slave communication mode, 7 or 10-bit addressing scheme and standard, fast and plus speeds of 100 Kbps, 400 Kbps and 1 Mbps respectively. I2C applications range from LCD's to accelerometers, i.e., ADXL345. The latter is used for illustration in this application note.

To interface to a slave device, the I2C master controller follows the next steps:

- START--pull data select (SDA) low, then (SCL) low to start communication.
- DATA—8-bit data: 7 bits of address and 1-bit for read/write from master.
- ACK—1-bit for slave responding to the master after 8 bits of data.
- RESTART—master initiates more transfers without releasing the bus.
- STOP—pull SCL high, then SDA high to stop interfacing and release the bus.

Figure 1 shows the I2C master/slave interface and the 2-wire standard mode with 2.2K Ohms recommended for data rates of 100Kbps or below.

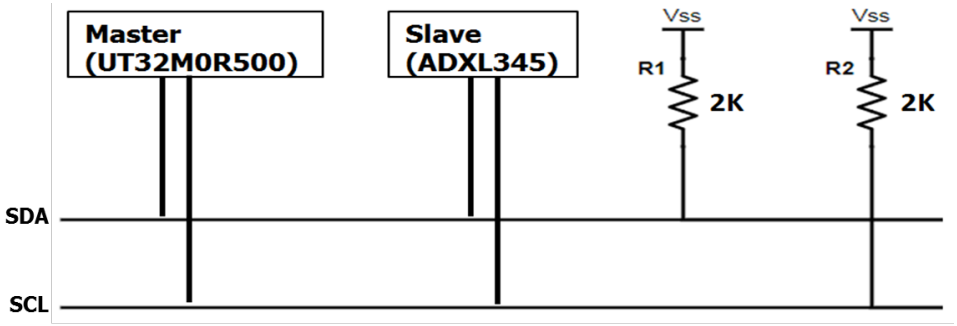

Figure 1: I2C master/slave interface

Transmission begins with the master pulling SDA low, followed by SCL low; it is the start of a new interface. After the start sequence, the next 7 bits constitute the address to enable up to 127 slave devices, followed by 1-bit for control, see Figure 2.

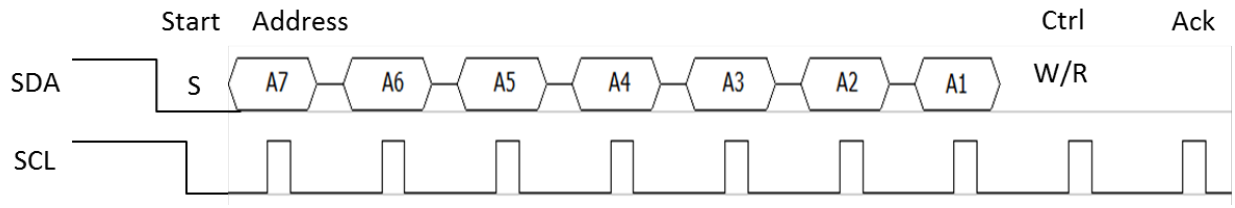

Figure 2: I2C master/slave addressing

All data transfers are 8 bits long, followed by 1-bit ack/nack, see Figure 3.

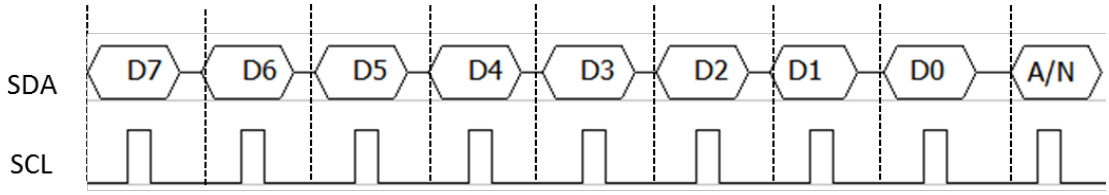

Figure 3: I2C master/slave data transfer

2.0 Application Note Layout

This application note (AN) provides a brief description of the I2C unit's memory map, configuration and programming.

3.0 I2C Unit Hardware

The I2C Unit is mapped to the memory region from 0x4000_9000 to 0x4000_9FFF for I2C0 and from 0x4000_A000 to 0x4000_AFFF for I2C1. Each unit has 46 registers. For more information on each register, refer to Chapter 16 of the UT32M0R500 Functional Manual.

3.1 I2C Unit Control Register

The Control Register **(IC_CON)** sets the slave mode (IC_SLAVE_DISABLE), bit [6], high for disable; the I2C restart

(I2C_RESTART_EN), bit [5], high to enable; 10-bit address master (I2C_10BITADDR_MASTER), bit [4], 0 for 7-bit; 10-bit address slave (I2C_10BITADDR_SLAVE), bit [3], 0 for 7-bit; speed range (SPEED), bits [2:1], 1 for standard mode; master mode (MASTER_MODE), bit [0], 1 for master mode.

Device Mode	Bit [6] and Bit [0] Value
Slave	00
Error	01
Error	10
Master	11

Table 2. I2C Device modes

3.2 I2C TAR Register

The I2C Target Address Register **(TAR)** controls addressing and target address; 10-bit address master (I2C_10BITADDR_MASTER), bit [12], 0 for 7-bit addressing; target address (IC_TAR), bits [9:0].

3.3 I2C SAR Register

The Slave Address Register **(SAR)** holds the address for I2C core operating as slave, target address (IC_SAR), bits [9:0].

3.4 I2C ISR Register

The Interrupt Status Register **(ISR)** is read-only for reading the status of the mask interrupts; bit values of 1 are for active interrupts.

3.5 I2C Interrupt Mask Register

I2C Interrupt Mask Register **(IMR)** is read/write and masks or enables all interrupts generated by the I2C controller.

3.6 I2C Raw Interrupt Status Register

I2C Interrupt Mask Register **(RISR)** is read-only for reading the status of the mask interrupts; bit values of 1 are for active interrupts.

3.7 I2C Enable Register

I2C Enable Register **(ENR),** bit [0], enables or disables all I2C operations.

3.8 I2C Status Register

I2C Status Register **(STR)** is a read-only register.

3.9 I2C Transmit FIFO Level Register

I2C Transmit FIFO Level Register **(TXFLR)** contains the number of valid data entries in the transmit FIFO.

3.10 I2C Receive FIFO Level Register

I2C Receive FIFO Level Register **(RXFLR)** contains the number of valid data entries in the receive FIFO memory. This register can be read at any time.

3.11 I2C SDA Hold Register

I2C SDA Hold Register **(IC_HOLD)** is for clock stretching; bits [23:16] extend the SDA transition (if any) whenever SCL is HIGH in the receiver in either master or slave mode; bits [15:0], control the hold time of SDA during transmit in both slave and master mode (after SCL goes from HIGH to LOW).

3.12 I2C Transmit Abort Source Register

I2C Transmit Abort Source Register **(IC_TX_ABRT_SOURCE)** is a read-only register; it holds the status of all aborted transactions in master or slave interface.

3.13 I2C Slave Data Nack Register

I2C Slave Data Nack Register **(IC_SLV_DATA_NACK)** works only when the I2C core is configured in slave mode; can generate a nack after a data byte is received by setting bit [0] high.

3.14 I2C SDA Setup Register

When the I2C core is in slave mode, the SDA Setup Register **(IC_SDA_SETUP)** controls the amount of clock stretching delay; the minimum value is 2.

3.15 I2C Ack General Call Register

I2C Ack General Call Register **(IC_ACK_GENERAL_CALL)** works only when the I2C core is configured in slave mode; When an I2C general call address, it can generate ack/nack after a data byte is received by setting bit [0] high.

3.16 I2C Enable Status Register

I2C Enable Status Register **(IC_ENABLE_STATUS)** reports hardware status from I2c core going from enable to disable.

3.17 I2C SS/FS Spike Suppression Limit Register

I2C Suppression Register bits [7:0] store the longest spike duration measured in clock cycles.

4.0 I2C Unit Initialization

The SCL counter value is derived by the following equation:

$$fSCL = \frac{fCLK(system\ clock)}{Duty\ Cycle * SCL\ Counter(SCL\_CNT)}$$

$$SCL\_CNT = \frac{fCLK(system\ clock)}{Duty\ Cycle * fSCL}$$

$$SCL\_CNT = \frac{50000000}{2 * 100000} = 250$$

Code 1 initializes the I2C 0 master control unit. It sets the clock frequency to 100 KHz by setting the counter value to 250 for standard mode, selects addressing to 7-bit and device master mode. For specifics on the API's, refer to www.frontgrade.com/HiRel.

```
 // Enables a device specific interrupt in the NVIC interrupt controller.
NVIC_EnableIRQ (I2C0_IRQn);

 // Set I2C Init structure defualts:
//SS_SCL_HCNT=250
//SS_SCL_LCNT=250
// SPEED=1, standard mode;
//addressing=0 for 7-bit addressing,; Device mode=1 for Master mode
// TFT=0x3, Tx FIFO threshold
// RFT=0x1, Rx FIFO threshold
I2C_StructInit (&I2C_InitStruct);
I2C_Init (I2C0, &I2C_InitStruct);

I2C0->INTR_MASK=0; // Mask all I2C 0  interrupts
I2C_IntConfig (I2C0, I2C_IM_RX_FULL, ENABLE);// Enable RX FIFO full interrupt
```

Code 1: I2C Initialization

5.0 I2C Unit Programming

Section 3.0 presented some of the basic configurations for the I2C core. The following sections show programming examples by making use of Frontgrade API's for the UT32RM0R500.

5.1 I2C Write Single Data

The API provides a function for sending one byte of data to the slave device. The function in Code 2 references the I2C structure and for a single write, sends one byte of address and byte of data to the specific slave device.

```
// MCP4725 DAC 0, write one byte
//uint8_t I2C_TransferData (I2C_TypeDef *I2Cx, uint16_t Address,
//   uint32_t NumTxBytes, uint8_t *TxData,
//   uint32_t NumRxBytes, uint8_t *RxData, uint32_t BusyWaitCycles)
I2C_TransferData (I2C0, 0x64, 1, fifoPut, 0, fifoGet, 0);
```

Code 2: Single byte I2C write

Data exchange between master and slave begins with the start sequence; then, the master sends the slave 7-bit address with bit [0], equals 0 for write. If the slave acknowledges, the master sends one or multiple bytes of data. When the interface is completed, the master sends the stop sequence, see Figure 4.

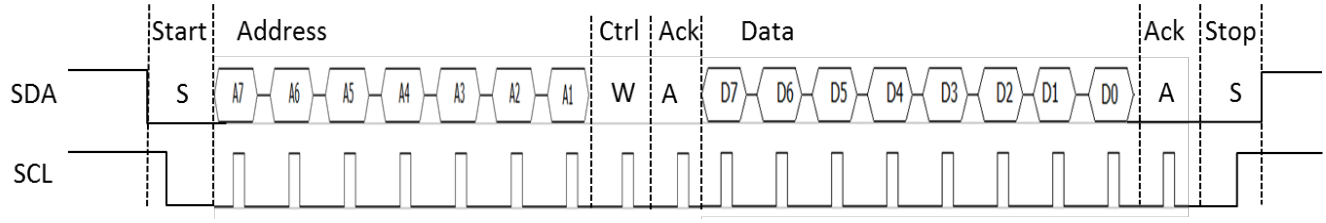

Figure 4: Master and Slave Write Interface

Figure 5 shows the Oscilloscope timing diagram for sending data between I2C master and MCP4725 slave at 100 KHz with 50% duty cycle.

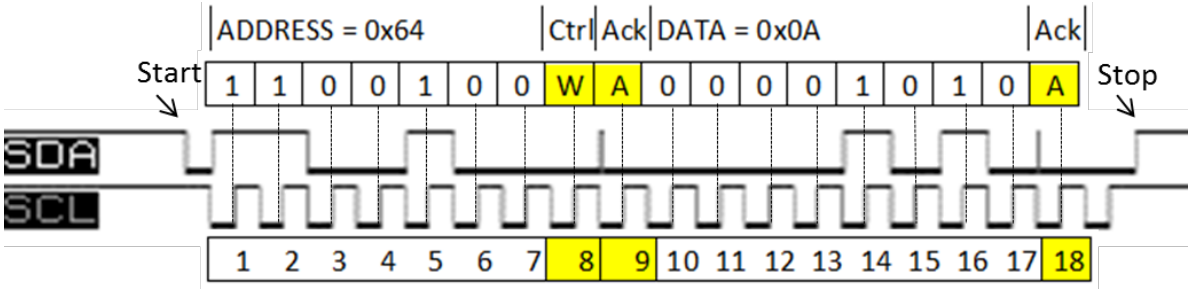

Figure 5: I2C Write Interface Timing Diagram

5.1.1 I2C Read Single Data

The API provides a function for receiving data from the specific slave device. The function in Code 3 references the I2C structure and if the flag RX_FIFO_NOT_EMPTY is set, it reads the data.

```
// ADXL345 accelerometer, read one byte
//uint8_t I2C_TransferData (I2C_TypeDef *I2Cx, uint16_t Address,
//   uint32_t NumTxBytes, uint8_t *TxData,
//   uint32_t NumRxBytes, uint8_t *RxData,
//   uint32_t BusyWaitCycles)
I2C_TransferData (I2C0, 0x1D, 0, fifoPut, 1, fifoGet, 0);
```

Code 3: I2C read byte(s)

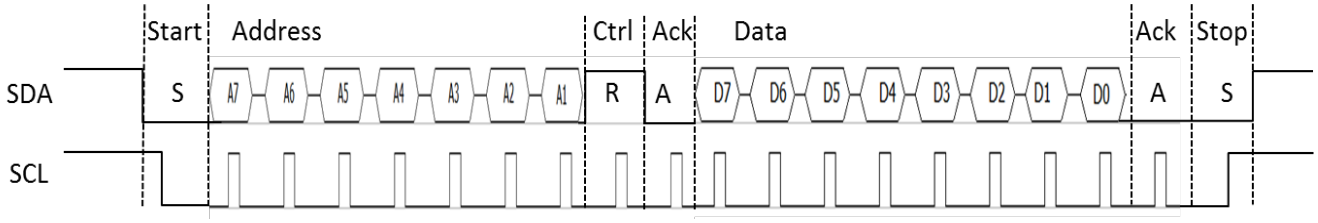

Figure 6: Master and Slave Read Interface

Figure 7 shows the Oscilloscope timing diagram for receiving data between I2C master and ADXL345 slave at 100 KHz with 50% duty cycle.

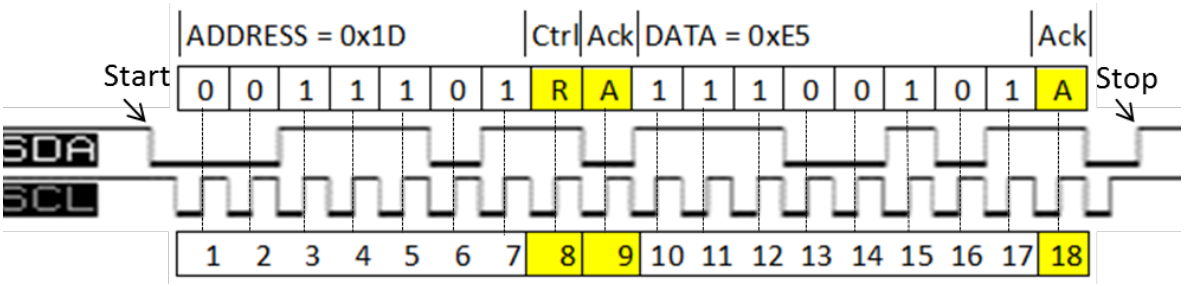

Figure 7: I2C Read Interface Timing Diagram

5.1.2 I2C Interrupt

All I2C interrupts are share to one interrupt (IRQ), which for I2C 0 is mapped to number 18 in the Interrupt Vector Table. The address of interrupt 18 in the Interrupt Vector Table is mapped to the I2C0_IRQHandler, which is the interrupt service routine (ISR) for all I2C 0 interrupts. In the ISR, software must check for which interrupt happened.

```
 // Enable RX FIFO full interrupt
I2C_IntConfig (I2C0, I2C_IM_RX_FULL, ENABLE);

//Enables a device specific interrupt in the NVIC interrupt controller
NVIC_EnableIRQ (I2C0_IRQn);
```

Code 4: I2C RX FIFO full Interrupt settings

Putting it all together, Code 5shows the main subroutine for sending and receiving I2C data in an endless loop. Code 6 shows The I2C0_IRQHandler, which is the interrupt service routine for handling the particular I2C interrupt.

```
int main (void){
        // Initialization and settings from previous sections go here.
        for(;;)
        {
                if(SysTickExpired)
                {
                SysTickExpired=0;

                do{
                  I2C0->DATA_CMD = I2C_DC_CMD;
                }while((I2C0->TXFLR & I2C_TXFLR) < I2C_TX_FIFO_DEPTH);
                }

                if(RxFifoFull)
                {
                        RxFifoFull=0;
                        do{
                                *fifoGetPtr++ = I2C0->DATA_CMD;
                        }while((I2C0->RXFLR & I2C_RXFLR) != 0);

                 }
}
void SysTick_Handler(void)
 {
    SysTickExpired=1;
}
```

Code 5: Sample program for ADXL345 Accelerometer

```
void I2C0_IRQHandler (void)
{
        // Check to see if the interrupt is SPI RX FIFO full
        if (I2C_GetIntStatus (I2C0, I2C_INT_RX_FULL)){
                // Clear the flag
                I2C_ClearInt (I2C0, I2C_INT_RX_FULL);

                RxFifoFull=1;

                // set GPIO 45
                GPIO_WriteOutputDataBit (GPIO2, GPIO2_PIN_GPIO45_OUTPUT, SET);
        }
        //GPIO_WriteOutputDataBit (GPIO2, GPIO2_PIN_GPIO45_OUTPUT, SET);
        __ASM volatile ("nop");
        // clear GPIO 45
        GPIO_WriteOutputDataBit (GPIO2, GPIO2_PIN_GPIO45_OUTPUT, RESET);

}
```

Code 6: Sample program I2C RX FIFO full interrupt

6.0 Summary and Conclusion

With speeds of up to 1 Mbps and only two wires, the I2C makes for a fast interface and more integrated functionality while at the same time reducing the number of I/O pins.

For more information about our UT32M0R500 microcontroller and other products, please visit our website: www.frontgrade.com/HiRel

7.0 Revision History

Date	Revision #	Author	Change Description	Page #
12/06/18	1.0.0	JA	Initial Release	