TWI Master/Slave AVR Example Code

RCC 9/2006, rcraig.campbell@gmail.com

This project is an attempt to figure out a completely reliable master/slave I2C driver for the AVR microcontrollers. The Atmel application notes refer to this being possible, but only implement, rather poorly, separate master and slave examples.

The challenges in implementing a completely reliable master/slave implementation are not at first obvious, and it is not difficult to write something that works the majority of the time. The attached code works for me about 199,999 times out of 200,000, but of course that's not good enough. Atmel, when asked, indicated that they have not seen sufficient user base demand to address this issue. My latest Atmel response is given below:

> Multimaster TWI communication is a beehive of lost packages and arbitration between masters. Implementation of TWI is done by software and unfortunately we do not have any application notes for multimaster TWI at this time. We are making application notes by the customers demands combined with potential volume of the end product, and we have not got the impression that application notes for multimaster TWI are requested at a sufficient demand to defend the resources needed to implement a multimaster system as an application note.

This is a fair, if disappointing response, so I have decided to post this project as a way to expand on Atmel's resources. I suspect, but am not completely convinced, that it is possible to solve all of the problems, but there are several issues with the TWI hardware design that make things difficult:

1. After receiving a repeat start or a stop on the slave interface, the ISR and overall system interrupt latency must be very short because the TWI hardware can not hold off the following transaction. This is an issue that is reasonably well known and has been covered in user group discussion, but is not addressed by Atmel and is very poorly addressed in the application note slave code.
2. When some other master owns the bus, you have no good way to know this, so the normal thing to do is to issue a START as master and wait for the TWI hardware to do the right thing, after which you will get an interrupt. The TWI hardware does not, in this case, delay for the required 5usec of bus idle time after a STOP before issuing it's START, and this can cause problems on a loaded bus. I know of no fix for this, and I have not seen it identified by others.
3. Similar to item 2 above, if another bus master is addressing you as a slave when you issue the START, there seems to be a small window where the wrong thing can happen, resulting in the slave interface sending or receiving the wrong data. Although it seems as if the slave interface could be disabled before the START is sent, I have not been able to completely eliminate the problem this way, and dealing with a slave interface that occasionally goes away requires complete control over the software of all devices on the bus, which is almost never the case. I have tried many things to address this, and have no solution yet. I have not seen it identified by others.

In order to share the problem here, I have put together a set of AVR code using the TWI that implements a fairly general purpose and flexible I2C interface capable of master and slave transactions. This code is a conglomeration of stuff used in real products, and while I'm happy to share this and for others to play with it and use it, I have a job and may not be able to do consulting on other people's applications.

On the master side, simple register based functions can be called for the simple transaction types such as byte and word reads and writes, or the API to the ISR can be used directly using the "polled" API routines as an example.

The Slave interface implements some GIR registers (Read only Status), A/D conversion registers (Read only), Control Registers (R/W with special actions, flags, and side effects), and some RAM (R/W). These types of behaviors are implemented as examples to attempt to bribe the user community to implement

general purpose I2C interface features in AVRs and therefore pressure Atmel to help us figure out how to make it reliable.

A serial port is used to implement a very simple monitor program that allows reading and writing over the I2C bus as well as printing some values.

The code is all written in assembly for AVR Studio as I could just not deal with the C compiler issues surrounding the low level hardware and interface stuff. I understand this is not an optimal approach, but code is code and I don't care very much what I write in. Don't be too intimidated, it's not that tough. This code will run fine on an ATMega128 on the STK500 + 501 board, but may take a little porting for other parts.

Code Overview

The software for the ucontroller is contained in a group of text files listed in the table below. The ATMega128 is configured to use the internal 8 MHz clock, 2.7V power threshold, and memory protection is currently disabled since there is nothing here of significant IP value that needs to be protected.

Startup consists of stack configuration and a series of calls to initialization routines for each of the interfaces or functions. The order of initialization is important, see the code comments for details.

The ucontroller does not go into power saving or sleep modes since the few milliamps of power consumption is not important in this particular design, but stays in a small main loop that checks for memory flags set by interrupt service routines indicating that it's time to perform some monitoring, and calling a few simple routines each time through the loop for operations that benefit from fast response.

File Name	Size	Description
A2d.asm		A/D initialization and ISR
Alarms.asm		Just a little check routine that updates status registers
Ctrl-regs.asm		Definition, initialization, and update/response for CTRL0 and CTRL1
TWI_MS_main.asm		Vectors, startup, and main loop including serial port monitor
I2C-ms-irq.asm		Definitions, initialization, and ISR for the I^2C interface
I2C-polled.asm		API's for simple byte and word reads and writes over I^2C
I2C-regs.asm		Variables, Definitions, and initialization for the slave registers
Pinout.asm		Definitions and initialization for the I/O pins
SCP-things.asm		Just a little stuff to call, more status update
Timers.asm		Initialization and ISRs for the 1 sec and 10 msec timers
Usart.asm		Initialization and interface routines for the debug serial port

Table 1: Source Code Files

1.1 Interrupts

There are 4 active interrupts used: 1 second timer, 10ms timer, I^2C interface, and A/D converter. All other functions are handled with periodic polling, some of which are based on the timer interrupts.

The interrupt routines are intended to be as small and simple as possible, and no nesting of interrupts is supported. There are no I^2C transactions allowed within interrupt service routines, and I/O pins should not be modified either, as this would lead to the need to lock all such transactions outside of the ISRs, which is not done.

1.1.1 10 ms and 1 Second Timer ISRs

Timer/Counter 1 is used in CTC mode for the 1 second interrupt, and timer/counter 3 is used in the same way for the 10ms interrupt. Nothing special here, output compare register A is used in both cases for both terminal count cycling and to generate the output compare interrupt. Initialization and ISR code is in the timers.asm file.

The 1 second irq is used to determine the test check interval, implementing a simple memory counter each second and comparing to the Test_Chk_Interval constant, setting the flag when it's reached.

The 10ms irq is used to directly set the flag to indicate that it's time to so vmon_check (so this is done 100 times/second), [note that this flag is not looked at in the main loop, so it does nothing now], and to increment a wait counter used by routines outside of ISRs for implementing delays (not nestable as implemented). This approach minimizes irq and response latency.

1.1.2 A/D Converter ISR

The example uses 4 channels of single ended A/D conversion for the first 4 A/D inputs. The converter is initialized to use the external 2.56V reference and the conversion clock is set to a high divider (128) since we don't need conversions to be done all that often. The settings result in 1 conversion every 208 usec, so each of the 4 monitored values are updated 1,201 times per second, much faster than needed for 100 Hz monitoring.

Each time the ISR is entered, it reads the value of the completed conversion, places the most significant 8 bits directly into the memory location used for the I^2C slave registers, updates the input mux selection, and starts the next conversion. Any function can access the latest conversion values by reading the I^2C memory locations. The ISR and initialization routine is part of the a2d.asm file.

NOTE: In case you have not yet learned this the hard way, the internal reference is very loosely toleranced and can not be used for absolute voltage measurements. If you are using the A/D to monitor supply voltages or something where you care about the actual voltage level, implement a better reference.

1.1.3 I^2C Interface ISR

The I^2C, or TWI, interface ISR is the most complicated section of code in the program, as it is required to implement both a continuously available slave interface so that other master devices on the I2C bus can access all of the status and control registers, and a master interface so that the ucontroller can deal with all of it's own peripheral chips and other bus devices. This is a difficult set of code to understand, primarily due to the somewhat convoluted TWI interface, so if necessary start by reading the ATMega128 datasheet TWI section several times. Despite extensive time and effort and many emails with Atmel support and the AVR user forum, there is still a very small time window in this code that may result in a device addr nack or even read data error when it is accessed as a slave. The frequency of error in software version 01 was measured at about 1:200,000 accesses.

The ISR is broken into 2 case statements modeled after the Atmel drivers for master and slave operation. In general the slave interface has priority over the master, with the master dropping the bus and restarting on a conflict. The slave case statement is a complete implementation of the slave device behavior, supporting both reads and writes as allowed to all of the slave interface registers, and setting flags to indicate further processing is required as needed. The master case statement is similar, implementing a complete transaction as defined by the buffer contents and byte count variables. To address some rare cases that do occur in multi-master environments, it is necessary for the master to loop to the slave and vice-versa. This is due to the inability to reliably set the master/slave flags prior to the ISR being entered in all cases, but the details here and the cases where unexpected things happen are difficult to understand. If you set breakpoints in the default cases at the end of the ISRs for master and slave, you will get some hits in busy multi-master operation.

Note that the master ISR does not currently support autonomous repeat start transactions, I didn't need them (this was developed for a system that does not share devices), but it can be added with a second buffer, byte count, and flag.

1.1.4 I/O Pins

Although this application has no particular need for I/O pins, they were used in some of the code that came along for the ride, and an explanation of the approach is included here.

Because I/O pins can be inputs, driven outputs, open collector outputs, or a combination of these things, it was challenging to come up with an abstraction using simple .equ statements that prevents the interface details from being imbedded in the code for every access. The solution implemented assigns a symbolic name to each interface pin based on it's function, with a _port, _bit, and _mask extension for each bit.

To determine the state of an input pin, the _port can be read and the bit tested with the _bit or _mask values. Polarity is not handled in this scheme, the code needs to deal with it directly. The DDR for inputs are initialized to 0, and the output port registers are initialized to 1 in most cases to enable the weak pullup. For input pins, the _port is set to the appropriate PIN register.

Driven outputs are set by reading the _port, setting or clearing the pin with the _bit or _mask values, and writing the result back to the _port. Again, polarity is the responsibility of the code. For driven outputs, the DDR is set to 1, and the _port is set to the appropriate PORT register. This means that reads of the _port will give the driven value, not the actual pin state, but they should be the same.

Open collector outputs are handled in the same way as driven outputs, except that the register DDR bit is used to enable the low drive. The PORT register bit is initialized and stays 0 so that when the DDR bit is set to 1, the output goes low. The _port is set to the appropriate DDR register, and the polarity is inverted. External pull-ups must be supplied. This sounds a little messy, but the code is pretty clean.

All of the pin definitions and initialization code is in the pinout.asm file.

1.1.5 Watchdog

The watchdog reset is enabled and set at a timeout period of approximately 110ms. The code assumes that the configuration fuses are set for safety level 1, requiring a timed instruction sequence to disable or change the WD timeout.

The WD is reset each time through the main loop, once per USART transmitted character (necessary for printing long strings), continuously in the USART input loop, and in the 10ms ISR only when the polled wait routine is active. The placement of WDR instructions is limited and avoided in any loop that could result in the ucontroller spinning forever in order to make the watch dog function as useful as possible.

The main loop is the normal case for resetting the WD, and the 110ms timeout is about 7800 times longer than the average loop time, and more than 100 times the known worst case loop time. Note that events that spend time outputting characters on the serial port will generate additional resets.

The USART transmit routine clears the WD once for each character, about every 1 ms at 9600 baud. This is necessary primarily to cover the long startup message that is printed. It is also cleared in the USART receive routine every time through the polling loop, which could cause a hang except for the timeout that is implemented in this routine (4 test_check intervals, about 20 seconds) to prevent this.

The only other routine that can take longer than the watchdog period is the polled wait routine (Wait_x10ms), which is used to delay during lamp test and a few other places. To ensure that an infinite loop is avoided here, the watchdog is reset in the 10ms ISR during polled waiting. The waiting routine is limited to an 8 bit counter which will wrap around as long as interrupts are active, so the maximum time is limited to 2.5 seconds.

There are multiple places in the code that spin waiting on an event that may not occur if there is some failure or problem. A primary example of this is the I2C driver, which includes several polling loops without timeouts. Rather than dealing with each of these cases separately, the watchdog timer is relied upon to address these issues. The reset resulting from a watchdog timeout will result in complete re-initialization of the hardware and peripherals. The effectiveness of this approach for I2C bus hangs is difficult to completely verify, and is complicated by the reset and startup behavior of any hotswap buffers in the system (which will isolate the backplane bus if it is the source of a hang). Testing has shown this to work well so far.

1.1.6 I2C Interface and Driver

As covered above in the TWI ISR section, the slave interface is completely contained within the interrupt service routine. The master interface is also nearly completely in the ISR, but for historic code reasons and simplicity, a number of routines that were previously implemented with polling code have had their API ported to the interrupt scheme. These routines, located in the I2C-polled.asm file, provide simple register based ways to read and write bytes and words across the bus.

There is a tiny hole in the bus locking in the case of a read that requires an address (or control word) to be written before the read is done. This transaction is a write followed by a read, and in the mult-master system it is possible for a different master to slip in during the few instructions between the read and write, access the targeted device, and change the address or control word. As long as the multiple masters do not access the same device, there is no issue, and for this reason direct access to the on-board peripherals by other masters is not supported. It's not a really big problem to deal with this, but it is not necessary so it has not been done.

When a control register is written by another master, the driver saves the old value, updates the register's memory location, and sets a flag to indicate that the new value needs to be checked for possible action. This checking is done every time through the main loop, typically every 15 usec or so, with the code in the ctrl-regs.asm file.

1.1.7 Alarm Interface

The state of the alarm interface inputs are monitored and updated in the GIR registers every time through the main loop as an example. No other action is currently taken based on their state.

1.1.8 RS-232 Debug

The UART0 serial port is configured as a 9600 baud, 8 bit no parity debug console that can be used to access various status information on the board and provide for a little control. The available command set is printed on the port at startup. If a valid command character is detected on the serial interface, the software enters a state where it waits for correct command completion and all other non-irq processing stops. A timeout is implemented to protect against an indefinite hang, so if 4 or more Test_Chk_Intervals have passed without the command being completed, the processor resets.

This code is spread out through the various files, with the port and character I/O routines in uart.asm, the monitor command case statement at the bottom of the main loop, and the various status routines in the appropriate files.

The current command set is described by the monitor startup message below:

I2C Monitor Version 01 for Freaks

Write Byte to Device: w <dev addr> <data>

Write Byte to Device w/ Register Adr: W <dev addr> <reg addr> <data>

Read Byte from Device: r <dev addr>

Read Byte from Device w/ Register Adr: R <dev addr> <reg addr>

Print GIR and CTRL Registers: s

Set Control Registers: S <msb> <lsb>

Print AD: p

Microcontroller I2C Register Map

The microcontroller I^2C slave interface implements several commands to allow other bus masters through the I2C interface access to microcontroller data, and to allow the microcontroller to be reprogrammed (not in this code set). These commands are detailed below.

All commands follow the typical I^2C protocol, with the device address and R/W bit, followed by the command byte (writes only) and one or more data bytes. The command byte must be written as part of a write command, and can be thought of as a register address. Multi-byte read operations will auto-increment the register address, so all of the status information can be read in a single read operation following a write to set the register address to the GIR. Writes to data bytes that are read only are not acknowledged past the register address byte. If bit 13 of the control register, auto-increment register address, is set to 1, consecutive read operations will increment the register address.

Address	R/W	Description
0x00	Read Only	GIR0 (LSB)
0x01	Read Only	GIR1 (MSB)
0x02	Read Only	A/D 0
0x03	Read Only	A/D 1
0x04	Read Only	A/D 2
0x05	Read Only	A/D 3
0x06	Read Only	Software Version
0x07	R/W	CTRL0 (LSB)
0x08	R/W	CTRL1 (MSB)
0x09 – 0x19	R/W	R/W RAM Data Block (16 Bytes)

Table 2: Microcontroller I^2C Slave Register Map

1.1.8.1 Read General Information Register (GIR)

The General Information Register is two bytes, and indicates the presence and good/not good status of things. The example code does not do much of anything with this, but I did leave the WD Reset bit in place.

Bit	Description
0, 1	
2, 3	
4, 5	
6, 7	
8, 9	
10	
11, 12	
13, 14	
15	Watchdog Reset Occurred

Table 3: General Information Register (GIR) Description

1.1.8.2 Read A/D 0-3

All A/D values are a single byte, with a value of 255 representing the nominal internal reference voltage of 2.56V, and units of (nominal reverence voltage)/255.

1.1.8.3 Read Software Version

This byte indicates the software version.

1.1.8.4 Read/Write Control Register

The control register allows other I2C bus masters to set features and control various things. In this code, only the slave read address auto-increment, implemented in the slave ISR, actually works, but the other bits, implemented in the CTRL0 and CTRL1_Change routines in the ctrl-regs.asm file have been left but commented out as a simple example of a control register that can implement IO pin or other action.

Bit	Description
0, 1	Alarm Outputs, bit 0 = SOP_M, bit 1 = SOP_B, set = 1 for both, default = 1, 0
2, 3	SCP Power/Clock Enable Primary and Secondary, Enable Power and Clock = 1, default = 1, 0
4, 5	Not Used
6, 7	SCP Power Cycle Primary and Secondary, Cycle off for 1 sec = 1, auto cleared
8, 9	Power Supply Inhibit Primary and Secondary, Inhibit = 1, default = 0, 0
10, 11	Fan Speed Force to Full Input and Exhaust, Full Speed = 1, default 0, 0
12	System Reset Drive line low for 1 sec = 1, auto cleared
13	Auto-Increment Register Address Increment after each read (wrapping) = 1, default = 0
14	Software Re-Program Mode Enable Enter Re-Program Mode = 1, default = 0
15	Re-Program Data Ack Flag 16 bytes + good cksum received and good = 1 Read Only, cleared on read, set after cksum write

Table 4: Control Register Bit Definitions