# Understanding and using the I2C bus

*By James M. Flynn*
*Senior Staff Engineer*
*Ericsson Inc.*

I ²C is as different from I ₂O as superscripts are from subscripts. This article describes the inter-IC control bus, a two-wire bus for providing a communication link between integrated circuits.

The Inter-IC Control (I ²C) bus is a de facto standard developed by Philips Semiconductors over a decade ago. Originally, its purpose was to connect a CPU to peripheral chips in a television, but its scope has since broadened to cover a whole range of intelligent devices. The reason for its acceptance is clear: board sizes are shrinking, thereby decreasing IC package sizes, while functionality of these programmable devices continues to increase. While this article deals with the original I ²C bus interface, the applicability of other interfaces such as ACCESS. bus, SMBus, and a host of other manufacturer serial protocol interfaces (SPIs) is evident. I'm focusing on this subject because while many of us routinely use these serial interfaces, only a limited number of tutorials describe the mechanics of implementing them. Dealing with these interfaces consumes much of my time; in fact, one of my biggest problems is simply monitoring the interface.

When it comes to monitoring various bus interfaces, there is an abundant supply of monitors for RS-232, Ethernet, and the like, but a distinct lack of tools for monitoring simple I ²C bus-like interfaces. With limited code examples, few ready-made tools, and often confusing documentation, I needed an easy-to-use, expandable tool that would allow me to monitor I ²C bus interfaces. In this article, I present a software life-cycle that I believe is well thought out, allows for easy customisation (for other serial protocols), and will give others the basis for a good workable tool.

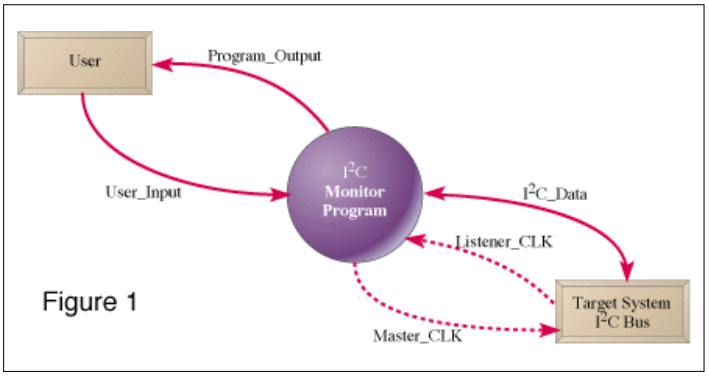

Figure 1

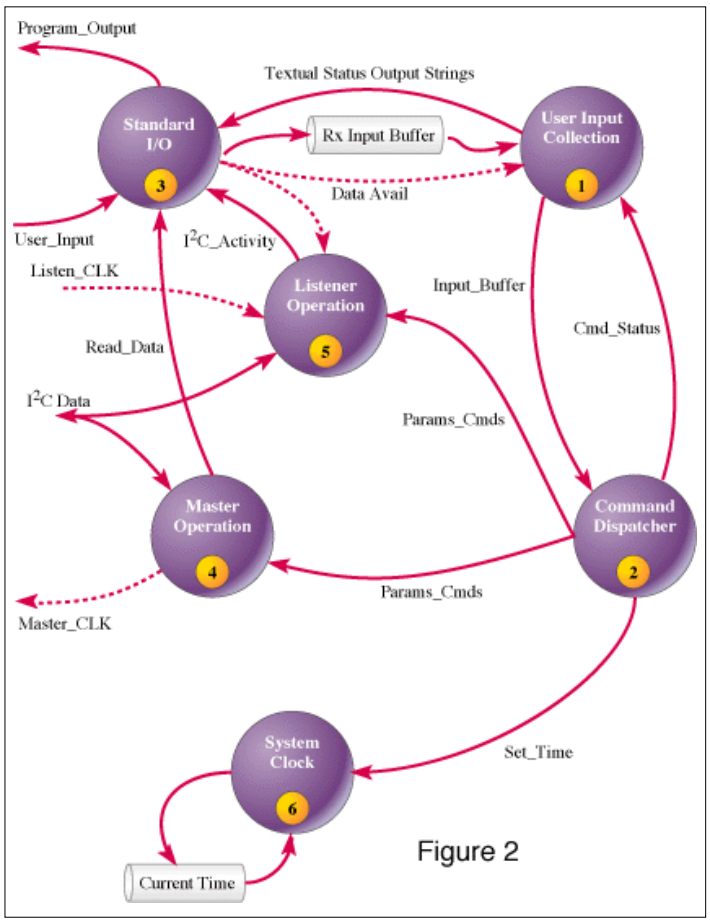

Figure 2

## History

The I ²C bus is a simple bidirectional two-wire interface that provides for efficient Inter-IC control. The bus has enjoyed such wide acceptance that Philips and other IC manufacturers now market over 150 I ²C bus-compatible devices. The function of these devices range from EEPROMs to LCD drivers. So what is the reason for this acceptance? Simply put, the device:

- requires only two wires to implement and has a unique address so that a master/slave relationship can be maintained
- is 8-bit- (or, byte-) oriented and bidirectional
- supports transfer speeds of around 100kHz (original standard, or 400kHz using the most recent standard)
- allows a relatively slow and inexpensive microcontroller to implement because of the generic nature of the bus interface

As a result of its simplicity, the I ²C bus interface has also been used as the basis for a number of related protocols such as ACCESS. bus and SMBus. Its influence on other manufacturer SPIs is undeniable. Clearly, understanding this protocol and having a good tool-box of I ²C interface routines would greatly benefit any embedded software practitioner.

## An I ²C Interface Project: Overview and Considerations

Recently, I wanted to profile a system that had a rather heavily-used I ²C bus. Because no hardware support for the I ²C bus existed, I expected that the microprocessor that implemented this interface was being disproportionately burdened. I then set off to profile the I ²C bus, using my trusty oscilloscope. Obviously, the easiest technique would be to measure the START to STOP transitions. However, multiple devices were on the interface, all talking at random times. Instead of going through the oscilloscope trace bit-by-bit, I wanted to simply connect a bus monitor that would trigger at the appropriate time, so I could observe individual devices. With this in mind, I developed the following list of specific requirements:

- First, the tool should operate as a non-intrusive monitor. The I 2 C bus being monitored will then operate the same way, whether or not the monitor device is attached
- Because the I 2 C bus specifies logic levels from of -0.5V to VDDmax + 0.5V, the tool should work equally well across this range without any

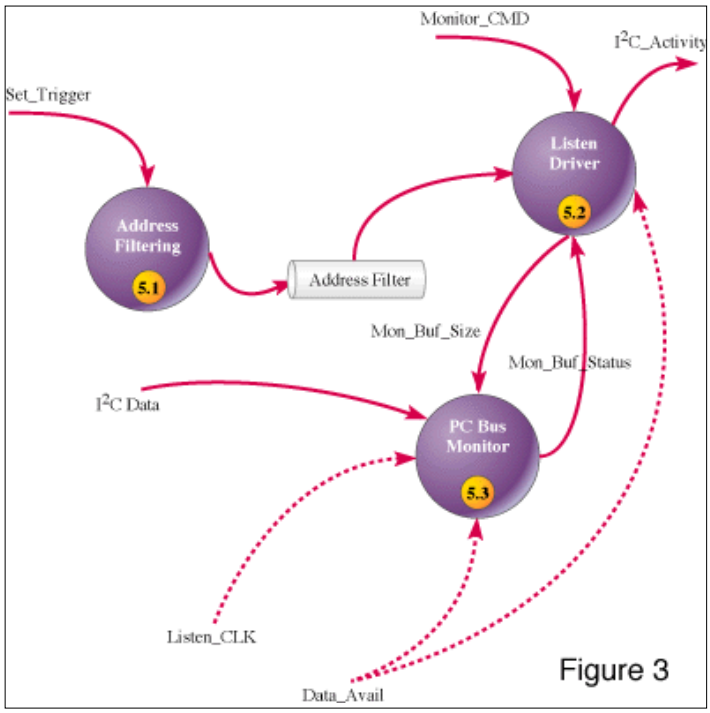

Figure 3

need for hardware reconfiguration. Whether the target system being monitored is a 3- or 5V system, the monitor need only be connected to the system's I 2 C bus

- When the program is in monitor mode, the user should be able to filter messages on the bus by slave address
- The monitor program itself should be designed so that new features and protocols can be easily added
- For completeness, the tool should be capable of operating as a master for both sending and receiving
- To limit the scope of hardware requirements, the bus speed should be limited to a standard I 2 C transfer rate of 100kHz
- Data should be time-tagged, preferably to the millisecond, as it's received from the I 2 C bus
- Terminating monitor operation at any time by providing some type of signal should be possible

**Project Requirements Model**

As with most projects, this one starts with a simple premise: essentially, the program needs to take input from the user and display the requested information. On the other side, it monitors the attached target data and clock lines. This context is depicted in **Figure 1**.

Most of the processing takes place on the next level. **Figure 2** depicts the major data processes and what I expect their data flows to be.

The standard I/O process will be responsible for handling all of the details associated with communicating to the user. This process will probably include initiali**sation**, as well as input and output routines. This process will probably also contain some interrupt-processing activities associated with the I/O activity.

A user input collection process will assimilate the incoming data into a single input buffer. Upon detection of an end-of-command value, the process will pass this input buffer to the command dispatcher process. The principal activity of the user input process will be to allow the user to edit input data before it is passed off for processing.

Once the command dispatcher process obtains the input buffer, the process checks the buffer for validity and then dispatches the appropriate command or action. After the command action is taken, the resultant status is returned to the user input process so that any necessary error messages can be displayed.

Because one of the requirements for this project is to time-tag incoming information, I've implemented a system clock process. This process not only maintains the current system time, but should also provide the functionality necessary to set and display time.

The listener process is what I'd actually intended to be the major part of this project. This process will require three pieces of information from the user: when to start monitoring; whether any particular slave address is being monitored; and whether listen operation should be terminated. **Figure 3** depicts this process.

Master operation is concerned with either reading from or writing to a slave device. In this mode, the program will be responsible for driving both the I ²C clock and data lines. See **Figure 4** for its context diagram.

Definitions for the various data flows are presented in **Table 1**. In this table, italicised definitions describe some physical entity, while non-italicised definitions represent composite functionality.

**I ²C bus Requirements**

In addition to the software requirements I've outlined, there are also the physical states and associated protocols. **Figure 5** is a summary of the various states that an I ²C bus can take. The two most important states in this interface are the high-to-low transition on the SDA while SCL is high-a start Condition-and a low-to-high transition of SDA while SCL is high-a stop condition. These conditions gate all activity on the I ²C bus. Once a start condition is detected, the bus is considered to be "busy," so no new transactions can be initiated.

While the start and stop conditions gate the data flow, a ninth data bit provides an acknowledge/no-acknowledge status. These data confirmation bits are sent during a master-generated ninth clock cycle. Generation of the data from this clock cycle is up to the slave device being addressed. An ACK (acknowledge) is generated by the slave device pulling the SDA line to a logic 0 during this clock cycle. If for some reason the slave device doesn't pull the line low during this clock cycle, the Master interprets the condition as a NOACK (no acknowledge). Each byte transferred on the I ²C bus is eight bits long so this ninth bit is the last data we would expect to see on the bus. A similar technique is used to control the pace at which data is transferred. In this situation, a slow device can throttle the I ²C bus speed by holding SCL in a low state after all the data and acknowledge bits have been received. Once a transmitting master sees this condition, it enters a wait state until the SCL goes high again.

Device addressing is principal to any discussion of the I ²C bus. All devices that connect to the I ²C bus have a unique address. These addresses are either seven

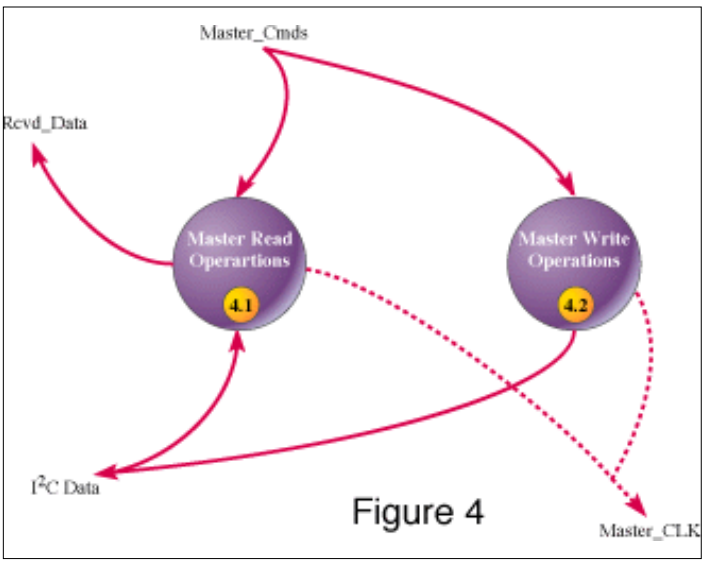

Figure 4

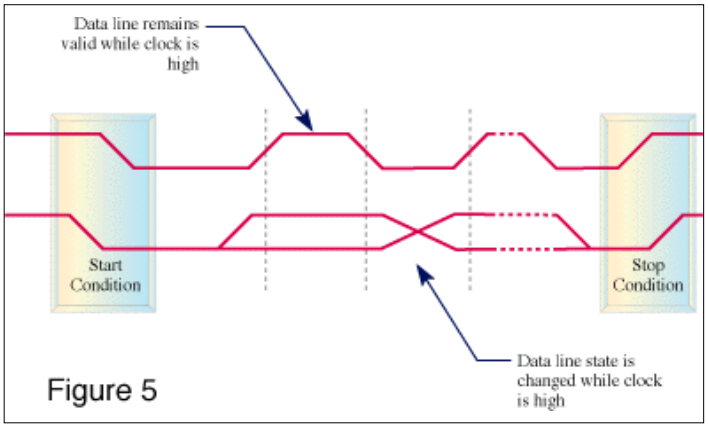

Figure 5

or 10 bits long. The first seven bits of address are always transferred along with a read or write bit immediately after the start condition. The address is used to identify the slave address being called, as well as specifying if this request is for reading or writing. In the case of an extended address (10-bit address mode), the first seven bits usually address a unique class of device, while the second byte contains the address bits needed to uniquely identify the device.

Data transfers fit into three basic formats. The first, a simple master-to-slave transfer, begins with the master generating a start condition. The master then sends the desired 7-bit slave address with the read/write bit cleared and begins the acknowledgment clock cycle. Once the ACK condition is observed, the cycle of writing data and reading ACK bits continues until the receiver either issues a NOACK or the transmitter completes the transfer and issues a STOP condition.

The second format, master-slave read, is similar to the previous format, except that the slave device immediately begins placing data onto the bus after the acknowledgment clock.

The last category is a combined format. Often this format is used when a device requires some initial configuration information in order to continue. Again, this format is similar to the previous two except that instead of issuing a stop condition, the transmitter issues a second start condition, followed by the slave address. You will often see this command used when accessing devices like serial EEPROMs. The first sequence is used to supply the device with the address that should be read, while the second start and clock sequence are used to clock the data back.

A sequence like this is also used to determine if a device is accessible. Because all I²C bus devices respond during the acknowledge clock cycle, if the device doesn't respond (a NOACK condition), it isn't currently accessible.

## Microcontroller Considerations

Many fine microcontrollers that could easily implement this project are on the market, and some even have I²C hardware support built right in. However, as I'm a software person first and foremost, I intended this to be a software project. With this prejudice in mind, I chose to use an 87C51FA microcontroller. This 8051 derivative is one of the more common microcontrollers available with an increased RAM capacity-I was never able to reduce the program's RAM usage to below 128 bytes. Because the 87C51FA afforded 256 bytes of internal RAM plus 8K of user-programmable EPROM, I figured it would satisfy my immediate needs, as well as any future enhancements. In addition to its expansive RAM, the MCU also provides a bidirectional input/output port that makes it ideally suited for implementing an I²C bus.

The I²C bus specification calls for connecting I²C devices via open drain outputs. With this configuration, the bus will have both SDA and SCL at a logic-high level when the bus is idle. Using the 87C51FA, I was able to use Port 0 (see Figure 6) to implement the functionality I've described. This port is a true open drain output (as opposed to the quasi-bidirectional ports 1, 2, and 3 that incorporate an internal pull-up on their input). Because it was a requirement for the I²C bus monitoring device to use the target system's VDD, Port 0 is the only port that could be used due to those internal pull-ups. With this microcontroller selection, I satisfied the requirement that the tool be non-intrusive.

Because the logic levels specified for the I²C bus vary greatly, it was important that the microcontroller running at 5VDC would be able to monitor an I²C bus that was operating at, say, 3VDC. From the data sheets, I determined that the input voltage for a logic low was between -0.5V and 0.9V, while a logic high was between 1.9V and 5.5V. With these specifications, this microcontroller should easily be able to satisfy the wide logic level range of the I²C bus and thereby meet my requirements for operating voltages. It is true that this hardware design decision is probably marginal, but my intent is to develop software, so please bear with me.

The last hardware requirement I'd specified was that we be able to operate at a 100kHz clock speed. Because the average execution period for an 8051 instruction is 12 cycles (though it does vary among instructions), a microcontroller using a 11.0592MHz clock can expect to average 1ms per instruction. It appeared, then, that there would be about 10 instructions worth of execution time during a normal I²C clock period. Because I²C transfers tend to be bursty (in the page size of the device being addressed), I felt that I'd have little trouble supporting this transfer rate. To monitor one of the new "Super" I²C bus devices (400kHz), it would be necessary to employ one of the faster 8051s that run at 40MHz clock speeds.

One important note about using this part is that it must never attempt to access external memory, due to Port 0 being multiplexed with the low order eight bits of address/data. The port pull-up field-effect transistor (FET) depicted in **Figure 6** is only used when external memory accesses are made. At all other times, this FET is turned off-so it's important to not accidentally enable this FET while monitoring an I²C bus.

Finally, the most practical reasons I chose to use this device are the availability and cost-it's easy to find and is generally less than $50 in single-part quantities.

## Tools and Development Environment

While I doubt that anyone reading this article would need to be sold on the benefits of using C, one must overcome a number of

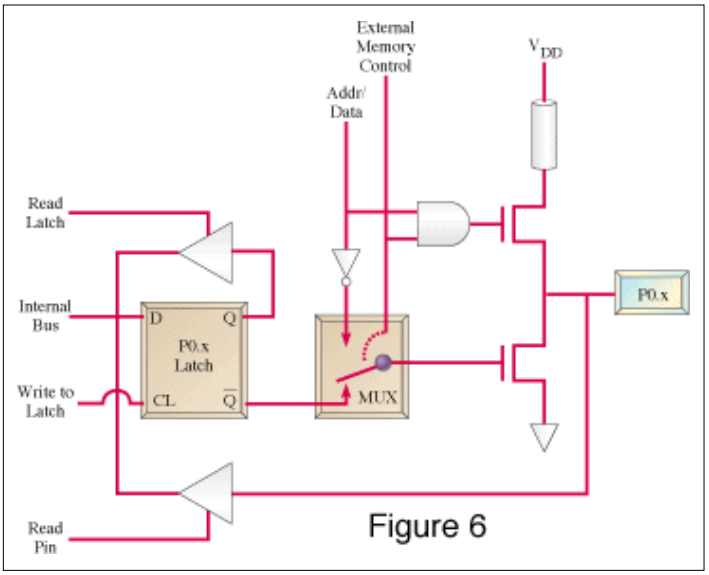

Figure 6

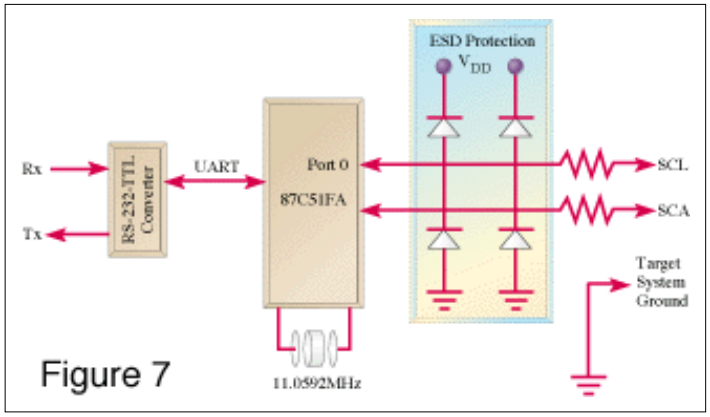

Figure 7

obstacles due to the microcontroller's limited RAM. Because C is a stack-intensive language, many of today's 8051 compilers contain data overlay techniques that minimise the RAM requirements to better utilise the available RAM. In fact, most 8051 C renditions pass parameter arguments in designated memory locations to reduce or eliminate the stack requirements, because of this limited RAM. The bottom line is memory. This key area must be researched when choosing a development environment for this family of microcontrollers.

To implement the code associated with this article, I chose Archimedes Software's C compiler. The IDS-51/251 Development Suite is a state-of-the-art 8051 development suite, that while targeted for a Windows environment, still maintains its command-line support. I chose this compiler specifically because:

- It's an ANSI C-compliant implementation for the 8051 family and includes all the associated C run-time libraries
- It includes a tightly coupled assembler for easy assembly/C function interaction and development.
- It incorporates a linker that will manage memory overlaying and optimises data memory overlays.
- It includes an 8051 simulator (SimCASE-51) that provides source level simulation of your program prior to any hardware being available or implemented

Using this suite of tools, I implement the majority of the program in C and resorted to assembly only when speed was critical (for example, when monitoring the I $^2$C bus). Once I'd coded the program, I was able to perform much of my testing using the bundled SimCASE-51 simulator.

Because I use DOS for most of my project development, I usually use makefiles tailored to the Microsoft NMAKE utility. In the code for this article (located on the Web at www.embedded.com/code.htm), you will find such a makefile. The DOS environment allows me to use the development tools that I prefer. Even though I do prefer DOS, the integrated development environment (IDE) supplied by the IDS-51/251 Development Suite is comprehensive. I guess the bottom line is that once you grow accustomed to an environment, you'll tend to stick with it.

## Project Architecture

A set of architectural diagrams is now required to describe how the project is to be implemented. I'll provide architectural diagrams for both hardware and software (though the hardware block diagram is admittedly brief).

Hardware Architecture. The representative block diagram, simple though it is, contains two points worth noting: because we'll be attaching to the target system's I $^2$C bus, ESD protection is a must; and we need to have a good target system ground connection. The resultant block diagram for the I $^2$C monitor is provided in **Figure 7**.

Software Architecture. A software architecture diagram is often overlooked in many projects, which amazes me because no one would think of developing hardware without first coming up with a block diagram. The software block diagram (SBD) for this project is depicted in **Figure 8**.

I settled on four distinct functional areas in the SBD. The first and most obvious is the user interface, which obtains the user's input, parses it, and dispatches the appropriate function.

Command processing is primarily a function of either the master or listener operating modes. The routines in the command processing sub-system interact with the routines that interface directly with the hardware. Because listener operation requires monitoring the master-slave communications that may take place at any speed up to 100kHz, I planned to implement the monitor function as a stand-alone assembly module. Lastly, I grouped all the miscellaneous routines under the inclusive category of library routines.

## The Implementation: Header files

Three header files were implemented for this program. The first file, system.h, contains hardware definitions and constants. The second, i2c.h, contains project specific typedefs, constants, and defines. Lastly, ascii.h contains some of the various ASCII codes used in the program.

## Program files

As is shown in the software block diagram, the I $^2$C program is made up of six core C files and one assembly file. The makefile manages the compilation and linking of these files. While Archimedes provides an excellent "total" solution, I still prefer a command-line environment for development work, so I implemented my usual makefile system. This makefile system is composed of the MAKEFILE, RULES.NMK, and HEX.NMK.

The main program for the I $^2$C project is i2cmain.c. I generally classify this file as "user interface" because most of the user input is processed there, and most execution takes place there as well. After the C start-up code has completed, execution is passed to the main function in this file. Note the manner in which functionality can be added to

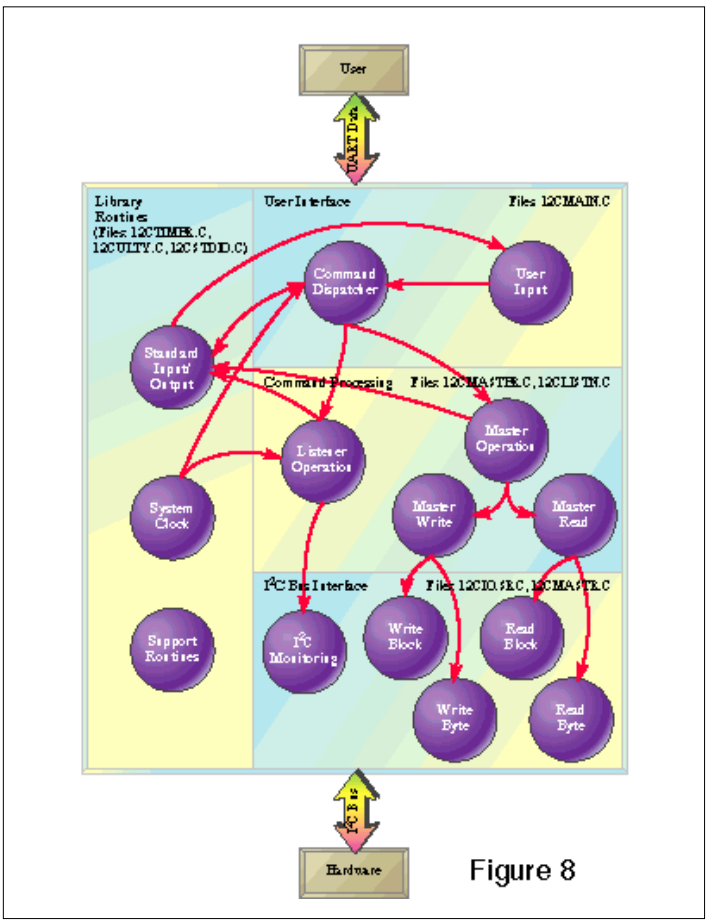

Figure 8

| Description | Defined in | Composed of/Definition |
|---|---|---|
| Program_ Output | Figure 1 | Cmd_Status \| I2C_Activity \| Rcvd_Data |
| User_Input | Figure 1 | Valid input |
| Listener_Clk | Figure 1 | I2C clock, which is driven by the target system |
| Master_Clk | Figure 1 | I2C clock, which is driven by the I2C Monitor Program |
| I2C_Data | Figure 1 | Data obtained from the I2C data line during the high-clock period |
| Params_Cmds | Figure 2 | Master_Cmds \| Set_Trigger \| Monitor_Cmd |
| I2C_Activity | Figure 2 | Received I2C data that is obtained while "listening" to the target's I2C bus |
| Rcvd_Data | Figure 2 | I2C data that is returned to us while we're operating in master mode |
| Data_Avail | Figure 2 | Boolean value (TRUE/FALSE) indicating that data has been supplied by the user; usually for the purpose of aborting operations in progress |
| Status_Output | Figure 2 | |
| Set_Time | Figure 2 | Desired time in ms (unsigned long) |
| Cmd_Status | Figure 2 | Returned-error status of a dispatched command |
| Input_Buffer | Figure 2 | The buffer of data that the user supplied the I2C program as a command |
| Monitor_Cmd | Figure 3 | Command to begin monitoring the I2C bus (GO) |
| Mon_Buf_Size | Figure 3 | A buffer address and size for where the monitor should place obtained data |
| Mon_Buf_Status | Figure 3 | This is the caller-supplied buffer (containing data), plus the read status |
| Set_Trigger | Figure 3 | This is the slave address of I2C device that we want to monitor exclusively (see Slave_Addr below) |
| Master_Cmds | Figure 4 | Slave_Buf_Cnt \| Slave_Addr \| Addr_Data_Cnt |
| Slave_Buf_Cnt | N/A | In master mode, this is the buffer-and its size-where received data from the slave device should be stored |
| Addr_Data_Cnt | N/A | Multi-byte write operations specify both the slave address to write to, the address of the data and the count of bytes to write |
| Slave_Addr | N/A | The 7-bit address assigned to the slave we wish to communicate with (or watch) |

Table 1

the file. All dispatch commands are contained in a table format (the "commands" array) and specify the string used to invoke a command, the function name to call, and the mode in which each command is valid. After the command is entered (and terminated with a carriage return), the dispatcher is called. This function extracts the first sub-string from the buffer and attempts to match it to a command in the dispatch table. If a matching command is found and the program is operating in a valid mode, the command is executed. If no command is found or it is in an invalid mode, then the function treats the command as if it were an intrinsic command (valid in all modes). As you can see, the program's functionality can be enhanced by simply adding the appropriate command to the "commands" table. In the current implementation, the program allows only three modes of operation: command mode, listen mode, and master mode.

The i2cmastr.c file contains the code for the master read and write operation. Essentially, the file is a set of stand-alone routines that implement master I $^2$C bus operation. The delay() function is used to ensure that the clock duty cycle is valid for the I $^2$C bus (remember, the duty cycle must be at least 10ms long). The i2c_read and i2c_write routines are used to read and write data, while the other routines drive the I $^2$C clock and data lines.

The i2clstn.c file contains the code that implements the driver portion of listen mode. This mode actually consists of the functionality required to monitor an individual I $^2$C bus address, as well as to display the received data to the user. When a complete I $^2$C bus transaction has been received, the data is time-tagged and displayed to the user. The i2cio.src assembly file completes listen mode and is implemented in assembly language for speed purposes. The i2c_listen routine collects information one byte at a time and stores it in the user-supplied buffer, interspersed with control information (START, STOP, ACK, and NOACK conditions). In terms of storing received information, it is assumed that the data buffer will be twice as large as an I $^2$C page. (A page is the number of bytes a device can accept sequentially before its internal address register wraps around.) The read8 routine collects data from the I $^2$C bus and detects the START, STOP, ACK and NOACK conditions. Because execution in listen mode primarily takes place in these two routines, I allow for their execution to be interrupted by the reception of any UART data. In this way, the user will be able to abort program operation if so desired.

Rounding out the program are what I referred to as the library routines. The i2cstdio.c file implements the details of receiving or writing data to the onboard UART. I implemented routines for initialisation, putchar(), puts(), getchar(), and kbhit(). The kbhit() routine simply returns a true or false condition, depending on whether any data is in the input buffer. The output routine's putchar() and puts() are wait I/O routines, so they will not return until all the requested data has been output. One difference between my puts() and the standard C routine is that it doesn't output a carriage return/line-feed at the end of the string. The timer support file, i2ctimer.c, provides the commands necessary to set and/or display the current system time. I decided to maintain the clock in 1ms increments using one of the microcontroller's internal timers (I tried to keep the interrupt routine as small and quick as possible). Because this interrupt can occur at any time, I minimised the interrupt's impact while monitoring the I $^2$C bus-by resetting the timer immediately after a start condition was found. Because a 100kHz clock rate allows only about 10 instructions of execution, I figured the interrupt detection and vectoring would take nine clock cycles while the interrupt service routine itself would take an additional 32 clock cycles-a total interrupt overhead of about 41ms. With the clock interrupt occurring every millisecond, I expect to get about 100 clock cycles or 11 bytes of I $^2$C data between clock interrupts. Because I allowed a page size of only eight bytes, I expect to be able to avoid clock interrupt by ensuring that interrupts don't occur while a sequence of bytes is being received. Of course if you have a lot of back-to-back traffic, you might miss some while the last received data is being sent to the user. To ensure that this clock interrupt

is avoided during reception, the routine rst_timer_isr was created. This routine is called immediately following the detection of a start condition. Lastly, the i2cutlty.c file contains support routines used to parse through the input buffer or format output data.

**Testing**

After completing the code, I simulated its operation with the Archimedes simulator. I wanted to test overall program operation, I $^2$C listen operation, and I $^2$C master operation. Because of the difficulty associated with simulating an I/O port that is constantly switching between input and output states, I chose to test I $^2$C master operation as a separate, stand-alone program.

Overall program operational testing was very straightforward. Testing the command-line input and output, intrinsic commands and dispatch commands involved verifying that the variables and actions were handled correctly.

Most of my effort went towards I $^2$C listener operation. In listen mode, I needed to handle the master-slave data exchanges for both combined and single/sequential byte transfers. I did this by creating the simulation file LISTEN.SIG. This file implements both single and sequential byte transfers in the function mxs7(). This function takes three parameters: a 7-bit slave address, the read/write bit state, and the number of bytes in the exchange. To test the combined format, I used the function comb7(). Like mxs7(), the comb7() function expects a slave address, read/write status, and number of bytes in the message. Note that if the total number of bytes specified in the third parameter exceeds a page size, the program will emit an error message stating that the page size was exceeded. In spite of the limitations encountered during simulation, it proved to be a great benefit in verifying overall operation.

**Major Influence**

The I $^2$C bus protocol is far from new, but it continues to endure. More important than its endurance, though, is the fact that it continues to be a major influence in the development of new serial protocols. I hope that others can avoid the frustrations associated with implementing these protocols and in the process find this software useful. If you should need to monitor a fast I $^2$C bus, then upgrade the hardware to one of the faster 8051 variants. If you want to add another serial protocol such as ACCESS.bus or SMBus, enhance the program by adding some new dispatch functions. Whatever the specifics of your situation, I hope you find this article and the I $^2$C bus approach useful.

**Bibliography**

The I $^2$C Bus and How to Use It (Including Specifications). Philips Semiconductor, 1996, www.semiconductors.philips.com/acrobat/3114.pdf.

www.geocities.com/siliconvalley. This site is a good resource for all kinds of information.