EMBEDDED SYSTEMS

Programming the I^2C Interface

When intelligent devices need to communicate

Mitchell Kahn

The Inter-Integrated Circuit Bus ("I^2C Bus" for short) is a two-wire, synchronous, serial interface designed primarily for communication between intelligent IC devices. The I^2C bus offers several advantages over "traditional" serial interfaces such as Microwire and RS-232. Among the advanced features of I^2C are multimaster operation, automatic baud-rate adjustment, and "plug-and-play" network extensions.

Mention the I^2C bus to a group of American engineers and you'll likely get hit with an abundance of blank stares. I say American engineers because until recently the I^2C bus was primarily a European phenomenon. Within the last year, however, interest in I^2C in the United States has risen dramatically. Embedded systems designers are realizing the cost, space, and power savings afforded by robust serial interchip protocols.

The idea of serial interconnect between integrated circuits is not new. Many semiconductor vendors offer devices designed to "talk" via serial links with other processors. Current examples include Microwire (National Semiconductor), SPI (Motorola), and most recently Echelon's Neuron chips. In all cases, the goal is the same: to reduce the wiring and pincount necessary for a parallel data bus. It simply does not make economic sense to route a full-speed parallel bus to a slow peripheral.

Unfortunately for most serial-bus-capable devices, the choice of a bus protocol will dictate the CPU architecture. For example, only two CPU architectures implement an on-chip I^2C port. If your choice of architecture precludes use of these architectures, then your only option is to implement the protocol in software.

The software implementation of the I^2C protocol discussed in this article came about as a result of an implicit challenge during a staff meeting. One of our managers proposed that we hire a consultant to write a software I^2C driver for the Intel 80C186EB embedded processor. Being somewhat new to the group, I took exception (although not verbally!) to his suggestion. A weekend of intense hacking later, I presented the first prototype of the driver. My reward? I got to write a generic version of the driver for general distribution.

Design Trade-offs

Three distinct tasks are involved in implementing the I^2C protocol: watching the bus, waiting for a specific amount of time, and driving the bus. This became apparent when I flowcharted 1 byte of a typical bus transaction; see Figure 1. The time delays associated with creating the bus waveforms would normally have been relegated to the 80C186EB's on-chip timers. I could not, however, assume that the end users of my code would be able to spare a timer for the software I^2C port. I had to forego the elegance (and to some extent accuracy) of the on-chip timers for the sledgehammer approach of software timing loops. Luckily, the I^2C protocol is extremely forgiving with regard to timing accuracy. The decision to use assembly instead of a high-level language stemmed directly from the need to control program-execution time. I had neither the time nor the inclination to hand-tune high-level code.

Having made the decision to use assembly language, I faced my next problem: Could I make the code portable? Intel offers a plethora of CPU and embedded-controller architectures. Would it be possible to make the code somewhat portable between disparate assembly languages? I found my answer in the use of macros.

Mitch is a senior strategic development engineer for Intel and can be contacted at 5000 W. Chandler Blvd., Chandler, AZ 85226 or at mkahn@sedona.intel.com.

All the basic building blocks of the I^{2}C protocol (watching, waiting, and doing) can be compartmentalized into distinct macros. The algorithms that make up the I^{2}C driver are written with these macros as the framework. You don't need to understand the intricacies of the I^{2}C protocol to port these routines—you just need to know how to make your CPU watch, wait, and do.

For example, a 4.7_uS delay is a common event during a transfer. The macro *%Wait_4_7_uS* implements just such a delay by using the 8086 LOOP instruction with a couple of NOPs for tuning; see Example 1(a). Total execution time is readily calculated from instruction timing tables. The same macro is ported to the i960 architecture in Example 1(b). Although I am a neophyte when it comes to i960 programming, I had no problems porting the core macros.

Hardware Dependencies

A few words about the target hardware are in order before I discuss the code. Any implementation of the I^{2}C protocol requires two open-drain (or open-collector), bidirectional port pins for the Serial Clock (SCL) and Serial Data (SDA) lines. The code in this article was designed for the 80C186EB embedded processor, which has two open-drain ports on-chip. The two pins, P2.6 (SCL) and P2.7 (SDA), are part of a larger 8-bit port. Processors without open-drain I/O ports can easily implement I^{2}C with the addition of an external open-collector latch.

Two special-function registers, P2PIN and P2LTCH, are used to read and write the state of the port pins. The 80C186EB allows the special-function registers to be located anywhere in either memory or I/O space. For this implementation, I chose to leave the registers in I/O space, even though this limited my choice of instructions. The 80186 architecture does not provide for read-modify-write instructions in I/O space (an AND to I/O, for example); it can only load and store (IN and OUT). So why did I limit myself? Again, I had to assume the lowest common denominator for our customers when designing my code.

Building the Framework

Early on in development, I decided to partition my code macros according to physical processes involved in the I^{2}C protocol. Code not directly involved in mimicking the actions of a hardware I^{2}C port was not written as macros. For example, the code necessary to access the stack frame is not written as a macro, whereas the code needed to toggle the clock line is. This was done to isolate architecture-dependent code sequences from the more generic I^{2}C functions. Macros were also not used for "gray areas" such as the shifting of serial data, which is both architecture dependent and physical in nature. The I^{2}C functions that passed the litmus test fell into the three aforementioned categories of watching, waiting, and doing.

The "waiting" macros provide a fixed-minimum time delay. They are implemented using a simple LOOP $ delay. The LOOP instruction decrements the CX register, then branches to the target (in this case itself) if the result is nonzero. The delay is (n–1)*15+5 clocks, where n is the starting value in the CX register. All the delays were calculated assuming a 16-MHz clock rate (62.5 nanoseconds per clock). The code still works at lower CPU speeds because the I^{2}C protocol only specifies minimum timings. In fact, the delay macros are only "accurate enough," providing timings as close as I could get to the specified minimum without undue tuning.

The "watching" macros are "spin-on-bit" polling loops. These pieces of code wait for a transition on the appropriate I^{2}C line to occur before allowing execution to continue. There are two polling macros for each of the two I^{2}C signal lines; one for high-to-low transitions and one for low-to-high transitions. The

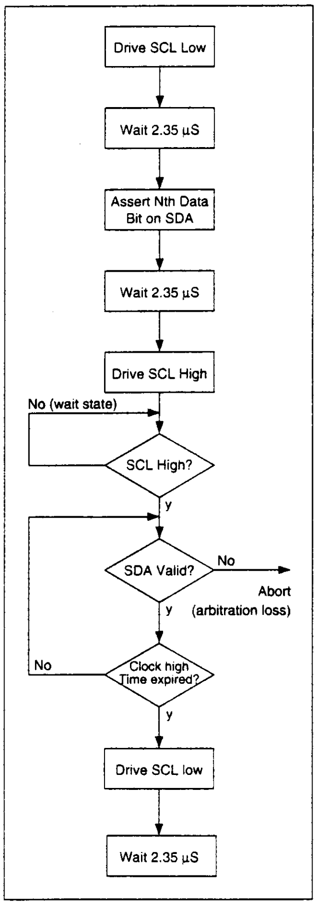


***Figure 1:** Flowchart of process for transmission of a single bit.*

(a)

```
%*DEFINE(Wait_4_7_uS)(
          mov     cx, 5              ; 4 clocks
          loop    $                  ; 4*15+5 = 65 clocks
          nop                        ; 3 clocks
          nop                        ; 3 clocks
                                     ; total = 75 clocks
                                     ; 75 * 62.5ns = 4.69uS (close enough)
          )
```

(b)

```
define(Wait_4_7_uS,'

          lda      0x17, r4          # instruction may be issued in parallel
                                     # so assume no clocks.
0b:       cmpdeco  0, r4             # compare and decrement counter in r4
          bne.t    0b                # if !=0 branch back (predict taken
                                     # branch)
                                     #
                                     # The cmpdeco and bne.t together take 3
                                     # clocks in parallel minimum.
                                     #
                                     # 0x17 (25 decimal) * 3 = 75 clocks
                                     # at 16MHz this is 4.69uS
')
```

***Example 1:** (a) 80C186 implementation of 4.7_uS wait macro; (b) 80960CA implementation of 4.7_uS wait macro.*

polling of the SCL line that gives rise to an important feature of I^{2}C: automatic, bit-by-bit baud-rate adjustment. Any device on the I^{2}C bus may hold the clock line low in order to stall the bus for more time (a *serial wait state*). The other devices on the bus are then forced to poll the SCL line until the slow device releases control of the clock.

The *%Get_SDA_Bit* macro also falls under the category of "watching." Its function is simply to return the state of the SDA line without waiting for a transition. *%Get_SDA_Bit* is used primarily to pull the serial data off the bus when the clock is valid.

The "doing" macros control the state of the clock and data lines. As with the polling macros, there are four types—one for each transition of the SCL or SDA lines. The "doing" macros are named to reflect the physical operations they perform. For example, *%Drive_SCL_Low* always drives the SCL line to a low state. *%Release_SCL_High*, on the other hand, relinquishes control of the SCL line, which may then be pulled high or driven low by another device on the bus. A read-modify-write operation is used for the bit manipulation so that the other 6 bits of Port 2 are not affected by the I^{2}C operations.

Getting on the Bus

Three procedures were created using the macro framework. I'll describe only the master transmit (Listing One, page 106) and master receive functions (Listing Two, page 108), as they represent the needs of most I^{2}C users. The slave procedure is long and intricate and will not be described here.

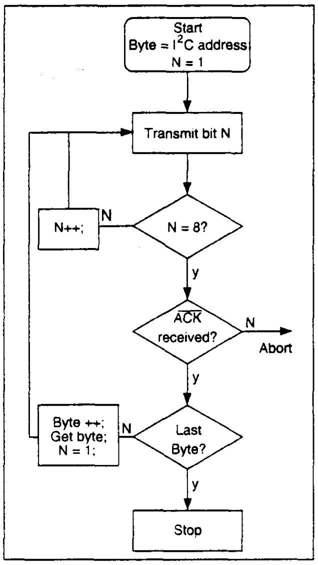


Figure 2: *Flowchart for I^{2}C transmit procedure.*

An I^{2}C master transmission proceeds as follows:

1. The master polls the bus to see if it is in use.
2. The master generates a start condition on the bus.
3. The master broadcasts the slave address and expects an acknowledge (ACK) from the addressed slave.
4. The master transmits 0 or more bytes of data, expecting an ACK following each byte.
5. The master generates a stop condition and releases the bus.

The stack frame for the master transmit procedure, I2CXA.A86, includes a far pointer to the message for transmission, the byte count for the message, and the slave address. Far pointers and far procedure calls are used in all the procedures. No attempt was made to conform to a specific high-level language calling convention, although such a conversion would be trivial. The procedures save only the state of the modified segment registers.

The master transmit procedure performs error checking on the passed parameters before attempting to send the message. The maximum message length is set at 64 Kbytes by the segmentation of the 80186 memory space. This restriction could be removed by including code to handle segment boundaries. The transmit procedure also checks the direction bit in the slave address to ensure that a reception was not erroneously indicated. Errors are reported back to the calling procedure through the AX register. (The exact code is in Listing One.)

The first step in sending a message is getting on the I^{2}C bus. The macro *%Check_For_Bus_Free* simply polls the bus to determine if any transactions are in progress. If so, the transmit procedure aborts with the appropriate error code. If the bus is free, a start condition is generated. The start condition is defined as a high-to-low transition of SDA with SCL high followed by a 4.7_uS pause. These waveforms are easily generated with the *%Drive_SDA_Low* and *%Wait_4_7_uS* macros.

All communication on the I^{2}C bus between the stop and start conditions, including addressing and data, takes place as an 8-bit data value followed by an acknowledge bit. This lead to the natural nested loop structure for the body of the procedure; see Figure 2.

The inner loop is responsible for transmitting the 8 bits of each data byte. Each transmitted bit generates the appropriate data (SDA) and clock (SCL) waveforms while checking for both serial wait states and potential bus collisions. A bus collision occurs when two masters attempt to gain control of the bus simultaneously. The I^{2}C protocol handles collisions with the simple rule: "He who transmits the first 0 on the SDA line wins the bus." To ensure that we (the master transmit procedure) own the bus, the SDA line is checked whenever transmitting a 1. If a 0 is present, then a collision has occurred (because another master is pulling the line low), and the transfer must be aborted.

Three distinct tasks are involved in implementing the I^{2}C protocol: watching the bus, waiting for a specific amount of time, and driving the bus

Control is turned over to the outer loop after the 8 bits of data (or address) have been transmitted. The outer loop immediately checks for an acknowledge from the addressed slave. The transfer is aborted if an acknowledge is not received. At the end of the ACK bit the message length counter is decremented. Control is returned to the inner loop if more data remains, otherwise a stop condition is generated and the master transmit procedure terminates.

Registers are used for intermediate result storage throughout the body of the procedure. For example, the AH register is used to hold the current value (either address or data) being shifted onto the SDA line. This eliminates the need for local data storage within the procedure.

On the Receiving End

The steps involved in an I^{2}C master receive transaction are almost identical to those in transmission:

1. The master polls the bus to see if it is in use.
2. The master generates a start condi-

tion on the bus.

3. The master broadcasts the slave address and expects an ACK from the addressed slave.
4. The master receives 0 or more bytes of data and sends an ACK to the slave after each byte. The master signals the last byte by not sending an ACK.
5. The master generates a stop condition and releases the bus.

A far pointer to the receive buffer is passed on the stack to the master receive procedure. The remainder of the parameters—slave address and message count—are identical between the two procedures. The received message length is fixed at 64 Kbytes, again because of segmentation. The error-checking, bus-availability sensing, and start-condition generation sections of the receive procedure are lifted verbatim from the transmit code.

The structure of the receive procedure differs slightly once the start condition has been generated; see Figure 3. The slave address is transmitted using one iteration of the transmit procedure's outer loop. Control is passed to the receive loop once the slave acknowledges its address.

The receive loop structure is patterned after that of the transmit procedure. The inner loop controls the clocking of the SCL line and the shifting of the serial data off the SDA line into the CPU. Eight iterations of the inner loop are performed to receive each byte. The outer loop stores the received byte in the buffer, decrements the byte count, then sends an ACK to the slave. The last data byte is signalled by not sending an ACK.

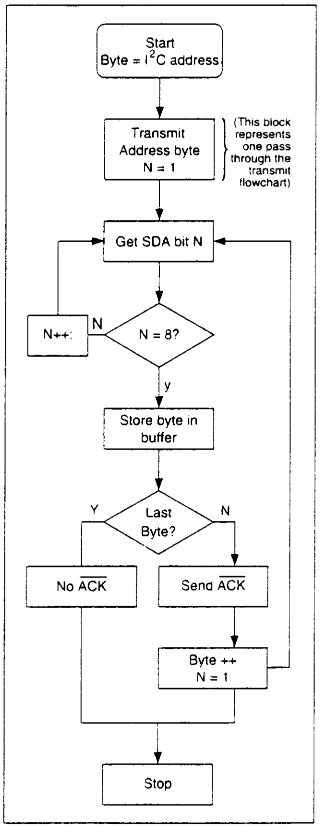


Figure 3: *Flowchart for I2C receive procedure.*

Using the Procedures

Listing Three (page 110) shows a short program that uses both the master transmit and master receive procedures. The call to procedure I2C_XMIT displays the word "bUS-" on a four-character, seven-segment display controlled by the SAA1064 I²C compatible display driver. The time of day is read from the PCF8583 real-time clock by the call to procedure I2C_RECV.

Please note that interrupts must be disabled during the execution of both procedures. An interruption at an inopportune time (when the master is not in control of the clock) could cause the bus to hang. If you need to service interrupts periodically, then enable them only when the clock is driven low.

These procedures have been tested on a wide array of I²C devices ranging from serial EEPROMs to voice synthesizers. No compatibility problems have been seen to date.

Enhancing the Code

I've kicked around many ideas for enhancing the I²C procedures. You could, for example, replace the timing loops with timed interrupts. That way, the CPU could perform useful work during the pauses. Along the same lines, the pauses could be scheduled using a real-time kernel, again improving CPU throughput. Finally, you could add a high-level language calling structure.

The use of timed interrupts adds an order of magnitude to the complexity of the code, but would be worth it for high-performance, real-time systems.

All the basic building blocks of the I²C protocol (watching, waiting, and doing) can be compartmentalized into distinct macros

Conclusion

I²C is not the only game in town when in comes to serial protocols. Hopefully, some of the techniques presented here will carry over into the development of other "simulated" serial protocols, such as those targeted at the home-automation market. Who knows, maybe someday a snippet of my code may find its way into a truly intelligent dishwasher. I'll be waiting....

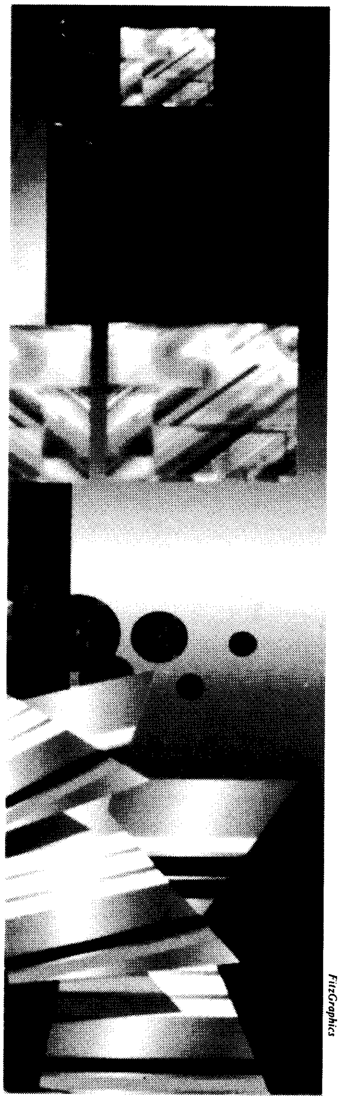
References

I²C Bus Specification, Philips Corporation (undated).

DDJ

Philips Semiconductors
a North American Philips Company
811 E. Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088-3409

Exploring I²C



Serial data buses are a well-proven tool in embedded systems. When you are communicating with slow peripheral devices, serial buses are often often more convenient and less expensive than parallel buses. Additionally, a serial interface featuring a UART or similar intermediary chip can also serve to isolate the CPU from noise and line glitches that might bring down the house if they were to occur on the processor bus. Peripherals can usually be controlled over a much greater distance by a serial bus. The serial approach offers greater resilience and noise immunity.

The price you pay for the convenience is a slower transmission rate and, possibly, the need for added interface circuitry at higher voltages. Many peripheral devices, however, are not in constant communication with the CPU and are not greatly affected by a slower bus. On the hardware side, any added interface circuitry required for serial-bus support is frequently compensated for by the resulting simplicity and tighter pinout of the serial peripherals.

The choice of bus and protocol depends at least as much on the system's software as it does on the manufacturer's data sheets.

CHOOSING THE PROPER ROUTE

Having decided that a serial bus makes sense for your application, your next task is to select the most appropriate bus and protocol. Here, as with rapid transit, your choice should be determined by your destination. Contrary to what some people may tell you, the choice of bus and protocol depends at least as much on the nature of the system's software as it does on the manufacturer's data sheets.

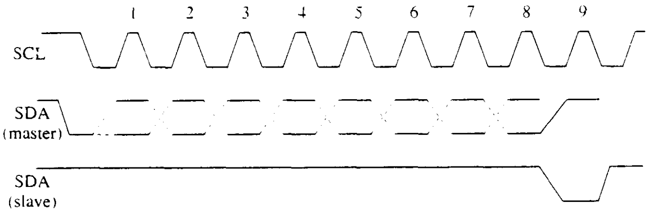
Consider, for example, the serial-peripheral interface (SPI) and multidrop serial buses. Both buses are popular, but each exhibits severly constrained performance in large networks. SPI, as embodied in the Motorola 6800 family, was designed primarily for one-on-one exchanges between two devices. Similarly, the multidrop approach used in various 8051 family members as well as in the 68HC11 and various UART chips finds its broadest expression in RS485/422 half-duplex transmissions. Multidrop has no deterministic arbitration scheme between multiple masters, leaving it mainly suitable for single-master multiple-slave situations. *(For more on multidrop, see Jack Woehr's article, "Multidrop Processing,"* Embedded Systems Programming, *March 1990, pp 58-67—ed.)* A different approach is to use a three-wire protocol called MicroWire, available from National Semiconductor in Santa Clara, Calif., which is fine for use with addressable peripherals, but requires an individual chip select for each device ad-

Exploring I²C

dressed. The added wiring offers no advantage to developers, and the bus offers nothing towards achieving multiple-mastering capabilities.

One of the more versatile options available to developers is the I²C bus promulgated by Philips/Signetics in Sunnyvale, Calif. I²C allows you to set up a multiple-master, multiple-slave communications bus with conflict arbitration, using only twisted-pair wiring to connect the processors and peripherals. Philips/Signetics has moved to support this protocol (which is quite popular in Europe) with a large assortment of interesting doodads, and is actively encouraging other manufacturers to join in the fun. If your next design features a microprocessor that supports I²C or you are prepared to implement I²C in software using a PIA as this article illustrates, your reward could be a decreased chip count and lower power consumption—along with a comfortable distributed-programming model for peripheral devices.

Figure 1
Generation of acknowledge.



Open-collector configuration means that the output stage can only pull the node to ground.

I²C is more flexible than the protocols noted above, since only two wires are required to service a large network of addressable masters and addressable slaves. A third wire may be added if interrupt service is required, though Philips/Signetics microprocessors featuring I²C support feature on-chip circuitry and are capable of interrupting the processor upon receipt of a valid address.

HOW I²C WORKS

The I²C bus consists of two lines: serial clock (SCL) and serial data (SDA). The beauty of the I²C bus is that each of these lines is bidirectional. Bidirectional means that everything on the bus is equal, unlike most other serial-peripheral busses such as SPI or MicroWire, which have dedicated inputs and outputs. Each I²C transaction line (SCL and SDA) is an open collector of output and input. The pullup resistor is external.

Open-collector (actually, they are CMOS, so "open drain" is more appropriate) configuration means that the output stage can only pull the node to ground. A passive resistor pulls the node high, which means that any number of open collector outputs can be connected together with no deliterious results, because it is impossible to pull more current through the resistor than any one output will produce. Tying outputs together will produce disastrous results if the same procedure is tried with standard TTL outputs. If some of the outputs go high and some are low, the current is unlimited and the logic level of the output will be in an indeterminate state. Tying open-collector outputs together is also known as "wire ORing" because if either A or B goes low, so does the single-output line.

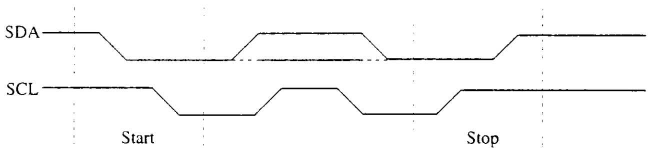
The I²C bus speed is specified at a maximum SCL rate of 100kHz SCL, which, admittedly, is not blazingly fast. The speed limit stems from the meager ability of a pullup resistor to source current to a long distributed line of peripherals. The 10-microsecond period allows plenty of time to charge the parasitic capacitance of the wires. (The maximum specified wire capacitance is 400 pF.)

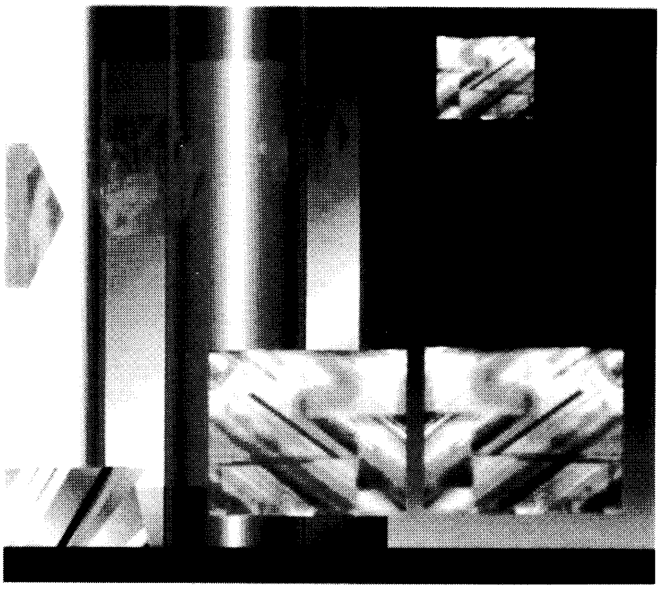
PUTTING IT TOGETHER

Although I²C supports multiple-master operation, here we use single-master, single-slave transactions to keep the example code simple. The master, as you might imagine, is defined as the unit that initiates the data transfer and generates the SCL signal. (In a multimaster system, each master would be responsible for generating its own SCL signal.) In our example, based strongly on the design of one of our company's single-board computers, the processor doesn't directly support I²C. Instead, we've implemented the I²C bus using a couple of the pins on an 8255 peripheral I/O chip. Consequently, the bulk of the example application code is simple setup and housekeeping routines. *(Steven R. Wheeler's example application listing was a bit too long to run in this issue. Interested readers may download it from the library 12 of CLMFORUM on CompuServe or from the* Embedded Systems Programming *bulletin board service at (415) 905-2689—ed.)*

By definition, a slave can be any processor or peripheral that responds to the master. Slaves all have unique, 7-bit addresses that are based on the device type and the wiring of address pins on the chip. All I²C peripherals have the top nibble of an address built in. For the PCF8574 I/O-port expanders we're us-

Figure 2
Start and stop conditions.





Exploring I^{2}C

ing as examples, the address is 0100*xxx*. The *xxx* indicates the address selected by the state of the three address pins on the peripheral.

I^{2}C serial transactions are always eight bits of data from the transmitter followed by a ninth ACK bit from the receiver. The first step in any I^{2}C data transfer is to send the address of the slave on the SDA line. This act might seem confusing, since we seem to be mixing 7-bit addresses with 8-bit data. In practice, it's quite easy to work with: addresses are always seven bits long, and the eighth bit is used to determine whether the operation is a read or a write. For example, upon transmitting 01000001 to the PCF8574, the slave, assuming it exists on the bus and is strapped to address 000, will respond with a low on the SDA line after the master has finished with its last (eighth) data bit. The master leaves the line high. If it doesn't find a slave with address 10000, the data line will remain high and a failed communication attempt can be detected.

If a slave is connected, it begins putting data on the SDA line as soon as it has detected that the eighth bit is set (which is a read request). The SDA line is driven to the data level when the SCL line is low. Data is read when SCL is high, so SDA must not change when SCL is high. This protocol leads to a simple definition of the start of an I^{2}C transaction—SDA goes from high to low when the clock is high.

The end of a transaction is equally simple to detect: SDA goes from low to high when SCL is high. This cycle leaves SDA and SCL in the high state, which is necessary if any other open-collector I^{2}C peripheral wants access to the bus. Figure 2 illustrates the start and stop conditions of an I^{2}C bus transaction.

ADDITIONAL DESIGN ROUTES

As you've seen, the I^{2}C protocol is easy to work with and relatively simple to implement, even if you're not using a processor that directly implements it. If you're not planning to use Philips/Signetics microprocessors with onboard I^{2}C support (such as the 68070 or various members of the 8051 family), you can still use the wide variety of available peripheral chips.

The number of integrated circuits using the I^{2}C serial bus is increasing all the time. Application-oriented integrated circuits that support I^{2}C include a voice sythesizer, a transcoder for IR remote control, several digital tuning circuits for computer-controlled television, several audio processors, PLL frequency synthesizers, tone generators, and frequency synthesizers. General-purpose integrated circuits using I^{2}C include LCD drivers, digital-to-analog converters, SRAMs, EEPROMs, and a RAM clock/calender.

I^{2}C is very popular in Europe, where Philips has been aggressively marketing this flexible method of extending peripheral support to control projects, and it is currently catching fire on this side of the Atlantic. It seems reasonable to expect that, given the burden of printed-wire requirements for embedded systems based on increasingly wider chip buses, more and more designers seeking economy of means will be attracted to the economy of I^{2}C.

Steven Sarns is the president of Vesta Technology in Wheat Ridge, Colo. He is a member of Mensa, Intertel, and the Michigan Society of Professional Engineers. Sarns is also a founding member of the Denver chapter of the Forth Interest Group.

Jack Woehr is a senior project manager at Vesta Technology Inc. in Wheat Ridge, Colo. He is a Chapter Coordinator for the Forth Interest Group and is currently a member of the X3J14 Technical committee for ANS Forth. He can be reached by E-mail as jax@well.sf.ca.us or as VESTA on GEnie.

BY MARK GARDNER

Bit-Banging Serial Ports

They say that necessity is the mother of invention, and it certainly seems to be the case in embedded systems work. No sooner do you accomplish the impossible in one project than your boss or customer asks you to do it again, only faster and cheaper this time. Even when you're working with low-cost microcontrollers, there's still that incentive to make things cheaper through magic software.

Performing miracles through software trickery is a skill that all embedded developers must cultivate. An opportunity for me to practice such tricks came in the form of a project using the Signetics 8x751 microcontroller. The 8x751 is an 8051 derivative that has no internal serial port—no attachment of SBUF shift registers to RxD and TxD, no diversion of timers to baud rate pacing, no serial interrupts. But the chip is low-priced and offers a small-footprint, and hence is desirable in many applications. Where the price or size outweighs the need for a simple serial port, one must be built out of firmware by appropriately controlling a single bit in a port. The practice is affectionately known as "bit-banging."

The approach I'll describe here has the advantages of being simple and fast. There is no transmit state-machine, no special provision for start and stop bits, and it takes less than two dozen machine cycles for each bit. It has a further advantage that the data doesn't need to be specially organized for transmitting. That is, the bits that are adjacent in the transmit data stream don't need to be adjacent when they are stored in memory. This solution is for a transmitter only, but I have used a similar procedure for receiving.

The shift (or rotate) operation is the first thing that comes to mind when you're designing code to provide a serial data output.

My project was required to operate at 9600 baud. This rate gives a per-bit time of 104 microseconds, or 104 cycles if you're using a 12-MHz part. The application in question had plenty of other activities as well as a serial port (such as reading a serial analog-to-digital converter, performing averages, and so on), so it was imperative that the serial port handling take an absolute minimum of time. Since I chose to execute in a fixed-time loop (to avoid interrupt overhead), it was also a goal that the code take a fixed amount of time regardless of the current transmit state.

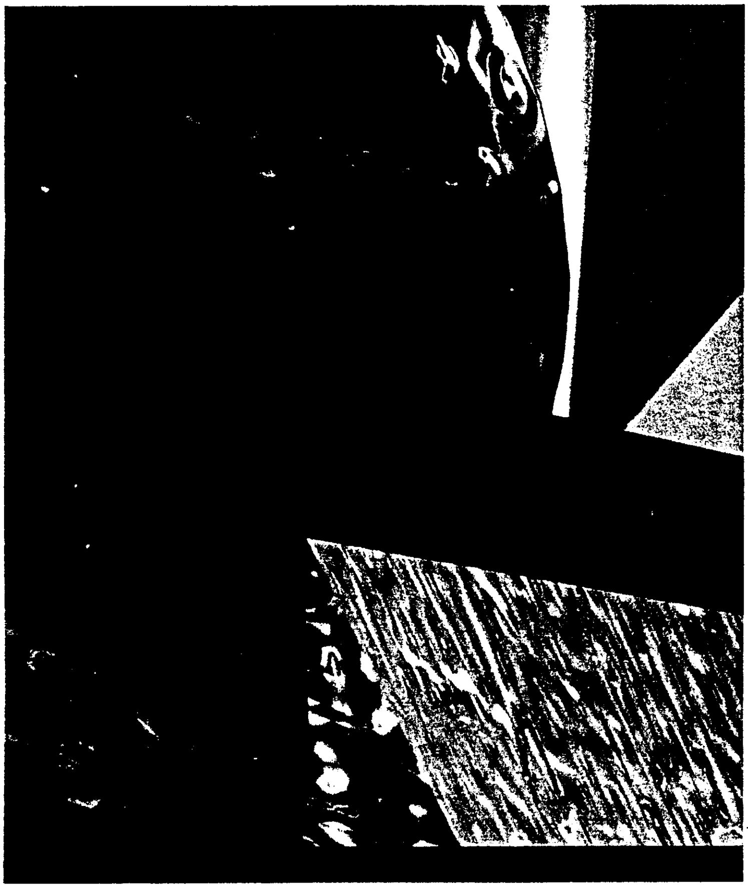
THE STRUCTURE POINTER SOLUTION

Generally, the shift (or rotate) operation is the first thing that comes to mind when you're designing code to provide a serial data output—the format of the data suggests such a scheme. With this approach, however, special states and a counter are needed to provide the start and stop bits and to sequence through the set of bytes to be transmitted.

The method presented here provides an array of structures (in the code or PROM space) that defines the transmit sequence bit by bit and uses a pointer to this array as the only controlling element. This means that only two bytes of scarce internal RAM is used.

The structures are referenced consecutively. Each gives the source of a bit to be transmitted and a flag to indicate whether the pointer should be increased to point to a new bit. The transmission is terminated by having a structure that refers to an "idle" bit and does not increase the pointer. Transmission is initiated by changing the pointer to point to the first structure. Start and stop bits are not distinguished from data bits. The bit update portion of the code is constant-time, and the pointer update can be easily padded if necessary to achieve this part of the goal.

Franklin's C51 compiler was used for the work described here. The 8x751 does not support external RAM, so the small model is used. (If the transmit data resided in external RAM, the algorithm could be applied, but would be expected to take a little longer to execute.)



THE DECLARATIONS

The structure that provides individual bit definitions is:

```
// transmit bit-reference structure
struct BR {
        unsigned char index ;
        unsigned char mask ;
        unsigned char bump ;
} ;
```

No memory is allocated by this definition—it is essentially a `typedef`. The actual allocation and initialization are provided by the definition (in a header file, send_seq.h, in this case) of the `BitRef` array:

```
code struct BR BitRef[41] = { ... } ;
```

where the details will be given in a moment. The pointer is defined as:

```
// pointer to BitRef structure array
data struct BR code *BR_ptr ;
```

In Franklin's C51, the declaration tokens are interpreted as follows. In the struct `BR` declaration, the token `code` assigns the `BitRef` array to program memory (which is then accessed with the `movc` instruction). In the `*BR_ptr` declaration, the token `code` implies that `BR_ptr` is exclusively a pointer to the program space, so it requires only two bytes to be completely defined. The token `data` causes the compiler to store the pointer value in internal RAM. (Since I was using the small model, this would have been the default storage anyway.)

The `index` entry in each structure allows the serial bit to be selected from an array of bytes called `transmit[4]` in my case. The transmit array can, if desired, be set up to literally overlay all of the internal memory, so that the maximum "random access" can be achieved. This was not necessary in my case.

The physical port pin to be exercised is defined:

```
/*     transmit is on P3.3 */
sbit TransBit = 0xB3 ;
```

THE STRUCTURE INITIALIZATION

Each bit to be transmitted is defined by an index and mask. These are initialized in the `BitRef` structure so that characters can be formed as desired in the output bit stream. The index is the offset within the transmit array. The initialization in my case, for a sequence of 40 bits making up four characters, was:

```
code struct BR BitRef[41] = {
// index   mask      bump    comment

   3 , b01000000 , 1,   //  0 start bit

   1 , b00000001 , 1,   // D6
   1 , b00000010 , 1,   // D7
   1 , b00000100 , 1,   // D8
   1 , b00001000 , 1,   // D9
   1 , b00010000 , 1,   // D10
   1 , b00100000 , 1,   // D11

   3 , b10000000 , 1,   //  1 fixed
   3 , b10000000 , 1,   //  1 fixed
   3 , b10000000 , 1,   //  1 stop bit

   3 , b01000000 , 1,   //  0 start bit
   ...
   ...
   ...

   3 , b01000000 , 1,   //  0 fixed
   3 , b10000000 , 1,   //  1 stop bit
   3 , b10000000 , 0    //  1 idle bit
} ;
```

(The "masks" are given in binary notation. [*See "A Binary Upgrade for C," pp. 60-62.—Ed.*] Because of my assembler and hardware background, this notation is natural for me in bit mask references.)

The "bump" is a flag that continues the transmission. When it finally reaches 0, the serial output sequence will stop.

The "index" refers, as mentioned, to the element of "transmit" in which the bit resides. Some initialization code has guaranteed that the upper two bits of `transmit[3]` will be 10, so that they can be referred to for start and stop bits and for any fixed-value bits that happen to be in the data stream (in my case, the fixed bits are used to indicate data byte order).

The "bump" is a flag that continues the transmission. When it is finally 0, the serial output sequence will stop.

THE CODE

The code fragment that accomplishes the transmission is:

```
(a) TransBit =
  (bit)( transmit[ BR_ptr->index ]
    & BR_ptr->mask ) ;
(b) if ( BR_ptr->bump )
  BR_ptr++ ;
```

The program sequence for section (a) looks like this:

```
BR_ptr->index
  -- looks up current index, then used in
transmit[index]
  -- to get byte with desired bit,
     then ANDed with mask
BR_ptr->mask
  -- to get zero/nonzero value, which
(bit)(value & value)
  -- is then cast to a bit for output
TransBit = bit
  -- to port pin, the ultimate goal.
```

The pointer is increased in (b), depending on the value of `BR_ptr->bump`. As indicated earlier, this is *always* one except in the last structure, so the serial transmission always proceeds to the defined end. The statement:

```
BR_ptr = &BitRef[40] ;
```

in initialization will keep the transmitter off during startup, and:

```
BR_ptr = BitRef ;
```

is used to initiate a transmission sequence.

Bit-Banging Serial Ports

The previous transmitting code compiles, with only a little manual assistance, to:

```
; TransBit = (bit)( transmit[
  BR_ptr->index ] & BR_ptr->mask ) ;
    MOV     DPL.BR_ptr+01H
    MOV     DPH.BR_ptr
    CLR     A
    MOVC    A.@A+DPTR
    ADD     A.#transmit
    MOV     R0.A
    MOV     A.@R0
    MOV     R7.A
    INC     DPTR
    CLR     A
    MOVC    A.@A+DPTR
    ANL     A.R7
    ADD     A.#0FFH
    MOV     TransBit.C
; if ( BR_ptr->bump )
    INC     DPTR
    CLR     A
    MOVC    A.@A+DPTR
    JZ      ?C0011
;      BR_ptr++ ;
    MOV     A.#03H
    ADD     A.BR_ptr+01H
    MOV     BR_ptr+01H.A
    CLR     A
    ADDC    A.BR_ptr
    MOV     BR_ptr.A
?C0011:
```

The assembly language code reveals that the mechanism is pretty efficient. This method is in use in one of my clients' products and has proved effective.

BIT-BANGING WORKS

This bit-banging solution serves to provide serial transmission in an embedded system that has no hardware specifically dedicated to the function. Although alternate and more traditional solutions would have worked, the need for speed encouraged development of a code-pointer-based solution that works fast enough in this case and takes up only two internal RAM bytes for operation. I hope that this presentation will prove to be useful for you.

Mark Gardner is a consultant based in Acton, CA. He has been designing hardware and writing firmware for embedded systems for over 15 years. He has an MS in electronic engineering from the University of Illinois.

For more information, contact:

Philips Semiconductors
811 E. Arques Avenue
P.O. Box 3409
Sunnyvale, CA 94088-3409
(408) 991-3552