Two-Wire Peripheral Expansion for the AT89C2051 Microcontroller

The attribute shared by most embedded controllers is their ability to interact with the outside world. While this fact is generally accepted, the form this I/O takes includes everything from parallel and bit-addressable digital I/O, analog I/O, as well as complex functions such as a user interface panel. Furthermore, timing related activities such as pulse width modulation, pulse accumulation, frequency measurement, and duty cycle and phase determination are often lumped together under the heading of high speed I/O. Additionally, while not necessarily I/O related, many small systems will benefit from other functional extensions involving real time clocks, interval timers, and nonvolatile memories.

Traditional interface techniques for such peripheral functions rely on using a conventional microprocessor data and address bus. While being inefficient in terms of printed circuit board real estate and requiring multiple interconnections, this standard approach offers good throughput and a choice of many established peripherals. Regardless of the benefits, this method is useless when working with very small single-chip microcontrollers that do not possess an external bus structure. What's required here is a synthesized expansion bus that does not excessively impose on the microcontroller's limited resources. That is, one that does not require too many I/O pins, firmware, or processor bandwidth.

This application note details an extensible I/O and memory expansion framework suitable for AT89C2051 embedded systems. To best illustrate the point an inordinate amount of external peripheral and memory functions will be accommodated while not intruding unnecessarily on the controller's limited I/O resources.

The Evolving Controller

A well established player in the embedded arena has long been the venerable 8051, quite possibly the epitome of an embedded controller. Although this fundamental architecture has been pressed to serve in a multitude of derivative designs, many of the scaled down versions have clearly been limited implementations. Worse, most of these have been plagued by a number of subtle and disturbing compatibility issues: almost compatible timers; SFRs, SFR bits, and I/O ports in the wrong places; missing instructions and missing functions.

Now everything has changed with the introduction of Atmel's AT89C2051. This small 20-pin circuit, unlike some of its diminutive predecessors, includes the full 8051 feature set—essentially an 80C51 in a 20-pin package. It retains all the standard SFRs and includes the full 128-bytes of internal data RAM. More importantly, the SFRs, SFR bits, and ports retain their original locations and functions. The standard processing core guarantees compatibility with many existing 8051 application programs, the multitude of established library and support functions, and most importantly, the immense accumulated 8051 knowledge base.

The retention of the 128 bytes of internal data RAM proves to be a rather significant issue. This can spell the difference between the option of developing code in a high level language such as C or being left with no other choice than working exclusively in assembler. And obvi-

0593A-B–12/97

ously, having a hardware UART opens up potential applications that were previously unattainable.

Packaging limitations reduce the AT89C2051's available on-chip I/O count to 15 pins. Although this is enough to handle a wide range of applications, there will be some that will need more.

Serial Standards

It is easy to add a lot of functionality to a very small controller such as the AT89C2051. Keeping the number of pins budgeted for the expansion bus low is desirable since most AT89C2051 based systems will benefit from preserving as much on-chip I/O as possible. In general, external peripherals are no match for the Boolean processor's bit-addressable I/O when high speed bit manipulation is required. And then there are the unique pins such as the external interrupts, the external timer controls, and the transmit and receive pins to the hardware UART. Quite obviously, a serial expansion bus is the only workable option, but this still leaves several alternatives to choose from.

Two primary categories of serial communication are defined: asynchronous and synchronous. Self-timed asynchronous communication is generally used for interfacing a microcontroller to other intelligent controllers or to a host computer. The intrinsic timing constraints make this method costly in terms of hardware or processor bandwidth (depending on the particular implementation). Although asynchronous methods are routinely applied in point-to-point or multidropped communication schemes, this necessitates some form of high level protocol. The implication is a level of processing capacity that falls well outside the domain of most dedicated peripheral circuits. Because of this, and because of the inherent timing constraints, asynchronous communication proves to be an inappropriate choice for a general purpose peripheral expansion bus.

Synchronous communication coordinates data transfer under control of a clock signal that is generated by the master controller. This clock signals when data bits are valid for both sending and receiving. Since the master controller generates the synchronizing clock, the protocol allows for a variable transfer rate. Here, the limiting factor is the maximum clock frequency. Usually, the minimum can go to DC. This can be an extremely important attribute in an embedded system where the master controller must vary the transfer rate according to varying degrees of interrupt loading and the stringency of processing multi-priority real time events.

The Microwire™ and SPI (Serial Peripheral Interface) standards are examples of popular synchronous protocols used for peripheral I/O. Even though an abundance of valuable peripherals exist that conform to these standards, their usefulness is diminished by the fact that the protocol provides no built-in means of addressing individual peripherals on a shared bus. Instead, each device uses a discreet chip select that must be individually asserted by the master controller before any communication can take place. This can pose significant problems if a number of peripheral devices must be accommodated concurrently since the number of I/O pins escalates. This constraint renders Microwire and SPI effective only if the system requires a small amount of peripheral functions.

Two-Wire Protocol

When a small system must support a moderate-to-large set of external peripherals (especially if additional functions may ultimately be necessary), the Inter-Integrated Circuit (I^2C™) standard offers an answer to the I/O pin dilemma. I^2C uses just two wires for communication regardless of the number of peripherals that are supported. More than just a methodology for transporting bits and bytes, I^2C introduces the benefits of a true bus architecture to the realm of 2-wire serial communications.

The two signal lines are defined as clock (SCL) and data (SDA). Electrically, these bi-directional lines are specified as open collector and, therefore, must be supplied with pull-up resistors to the positive supply rail. Since the lines are passively pulled up, they are in the recessive state when they are not being driven. Any device on the bus is free to pull these lines low thereby asserting the dominant state. This phenomenon is utilized for a variety of bus management functions including wait state synchronization and bus arbitration.

This 2-wire bus not only carries data and control information but is also used to establish addressability in order to select a specific bus member for data transfer. Additional information can also be transferred to access specific locations within memory devices and to access special configuration and status registers within complex peripherals. Although most applications will be content using I^2C in a master/slave configuration, the protocol supports multimaster capabilities that can be used for direct processor-to-processor communication or for implementing shared memories or peripherals that can be accessed from multiple processors residing on a common bus.

Standard I^2C throughput is specified nominally at 100 kbps with some newer devices capable of sustaining a 400 kbps transfer rate. This is more than adequate for many applications. Frequently, the peripherals used will be low utilization devices such as real time clocks, nonvolatile memories, and data converters that do not require high speed access. The maximum line length specification of 10 meters opens the intriguing potential of locating various devices "where the action is." The central controller can thus orchestrate the functions of a moderately dispersed system as easily as one that is self contained.

Before examining some of the available peripherals and looking at how they can be used in a small embedded system it would be informative to summarize some I^2C fundamentals. Since information is available on the I^2C standard, the following discussion will be limited to a brief overview of some of the most basic features of the protocol. Furthermore, the scope will be restricted to a master/slave peripheral scenario.

I^2C Protocol Recap

For our simple master/slave implementation two categories of bus members will be defined: The bus master which initiates and coordinates the particular transmit or receive operation and the bus slave that carries out the requested function.

Special Conditions

The I^2C protocol establishes a number of unique line conditions that are initiated by asserting the SDA and SCL lines in specific combinations. For example, all bus operations are initiated by issuing a START condition which causes all bus members to listen for incoming data. The master accomplishes this from an idle state by first pulling SDA low and then pulling SCL low.

The conclusion of a data transfer sequence is framed with the complement of the START condition which, naturally enough, is called the STOP condition. Beginning with SCL and SDA low, the master first releases SCL and then releases SDA. You will notice that the line transitions occur just opposite to those of a START condition. The STOP condition signals that the bus has been released and indicates that all bus members may expect another transmission to start at any time.

Data Transfer

The start and stop conditions indicate special bus seize and release phases. Once bus control has been established, data is transferred in a conventional clocked fashion eight bits at a time, MSB first. Data bits are set up when SCL is low and must remain stable while SCL is high. After holding SCL high for a period of time, the master pulls SCL low before the state of SDA is allowed to change. Notice that the only time SDA is permitted to change while SCL is high is in a START or STOP condition.

The fact that the master controls the system clock does not necessarily imply that it has absolute control over the transfer rate. As noted, I^2C's wire AND characteristic allows either the master or the slave device to place either SCL or SDA in the dominant low state. This capability allows slower slaves to cope with a high speed master at either the bit or the byte level. At the bit level the data transfer can be slowed down when the slave extends the SCL low interval. The master checks the state of SCL while transferring data and will not proceed while SCL is being held low. This is the I^2C version of a wait state. Note that even though the transfer rate is variable, parameters such as setup time, hold time, and the minimum clock high and low times must not be violated.

In some cases it may be necessary for a slave to prevent the master from initiating any bus activity which might be the case if it requires additional time to process received data. A slave can accomplish this by pulling SCL low. Since the master will generate a start condition only when the bus is free (SDA and SCL high), this forces the master into a hold state until SCL and SDA are freed.

In general, each data byte transferred requires an acknowledgment. This is implemented as a bit-level function that occurs on the 9th clock pulse immediately following a data byte transfer. Subsequent to the transmittal of the 8th data bit, the transmitter releases SDA to the high state. At this point the receiver must signal the successful receipt of the data byte by pulling SDA to a logic low. This acknowledgment must be asserted by the time the master drives SCL high and must remain stable during the SCL high time. This acknowledge bit is evaluated by the transmitter in order to determine the status of the data byte transmission.

Device Addressing

Following the assertion of the start condition, a 7-bit slave address is transmitted by the master. Remember that I^2C defines all data transfers as 8-bit entities. In the case of the 7-bit slave address the 8th bit functions as the direction bit, the read/write indicator. When it is a 0 the subsequent transfer will be a write to the slave. A 1 indicates the ensuing operation will be a read from the slave. Once the address is received, all slaves compare the received address with their own. A match results in an acknowledge to the master from the selected slave device indicating it is ready to perform the requested operation.

An I^2C peripheral address is composed of two parts. The fixed part is defined by the I^2C bus committee and is assigned based on device type. The programmable part comprises the lower order bits and is selected at the slave by strapping address pins high or low. The number of available programmable bits depends on the number of pins can be made available for this function on a particular IC. This scheme allows for multiple peripherals of the same category to reside on the bus at the same time and still be uniquely identifiable.

I^2C Summary

Depending on the particular application, using the I^2C bus can get considerably more complicated than implied in the preceding description. Nonetheless, a great deal of practical functionality can be supported using just a master/slave subset of the protocol. The following is a summary of the main points just touched upon.

- A high-to-low transition of SDA while SCL is high signals a START condition.
- A low-to-high transition of SDA while SCL is high signals a STOP condition.
- ISDA must be stable during the high period of SCL while data is being transferred.
- Data is transferred MSB first, 8 bits at a time.
- Every byte transferred must be followed by an acknowledgment bit (generally).
- The I^2C bus is considered busy following a START condition.
- The I^2C bus is considered free a certain time after the STOP condition.

Figure 1 pictorially illustrates the criteria for data validity, a START and STOP condition, and a data transfer sequence.

Simple and Registered Devices

Simple I^2C devices such as parallel I/O ports contain only one register that is located at the base address of the chip. Accessing such a device involves merely addressing the chip and then performing a read or write operation.

There are a number of devices, however, that contain multiple internal locations. These include memories, real time clocks, and data converters. Memory devices contain a linear memory array whereas other devices might have multiple data registers and control and status registers located at various internal addresses. Regardless of the implementation details, it is obvious that some means of specifying the internal address is required.

Selecting such a device's internal address involves the standard sequence of establishing a START condition followed by a transmittal of the slave address with the command bit set to write. The next byte transmitted is the actual register address which effectively sets the device's internal address generator to its initial value. I^2C allows combining the initial register addressing phase and the subsequent data transfer phase into a single functional sequence. In the case of a registered write operation, the slave will already have been placed into write mode prior to the transmission of the register address. Any subsequent data transmitted to the slave will be deposited into the specified internal location. A read operation requires one additional step.

Since the slave is in write mode following the register address transmission, it must be explicitly prepared for a read operation. One way to "turn the line around" is to conclude the write sequence by setting the STOP condition and by explicitly starting a new read operation. Although this works, it does incur additional, and unnecessary, overhead. This inefficiency can be circumvented by skipping the STOP condition and, instead, immediately issuing a repeated START condition. Now, a read operation is initiated by transmitting the slave address with the command bit set to read.

Soft I^2C

To the experienced engineer the preceding discussion would have undoubtedly suggested a number of firmware-based approaches for the implementation of the I^2C protocol. This is only natural since a Boolean processor like the AT89C2051 makes such an implementation extremely efficient and straightforward. Now, it's well known that a number of popular microcontrollers provide built-in hardware support for I^2C, but how much support are you really getting?

What may not immediately be apparent from a superficial examination is that many controllers provide this hardware assistance only at the bit level. This is truly rudimentary support that consumes valuable processor silicon to little advantage. In fact, it has been the experience of many engineers that using such minimalistic hardware support can actually result in greater software complexity and the use of more program memory than a purely firmware based approach. And remember, working entirely in firmware allows you to implement however much, or little, of the I^2C protocol as is necessary or appropriate to the task at hand.

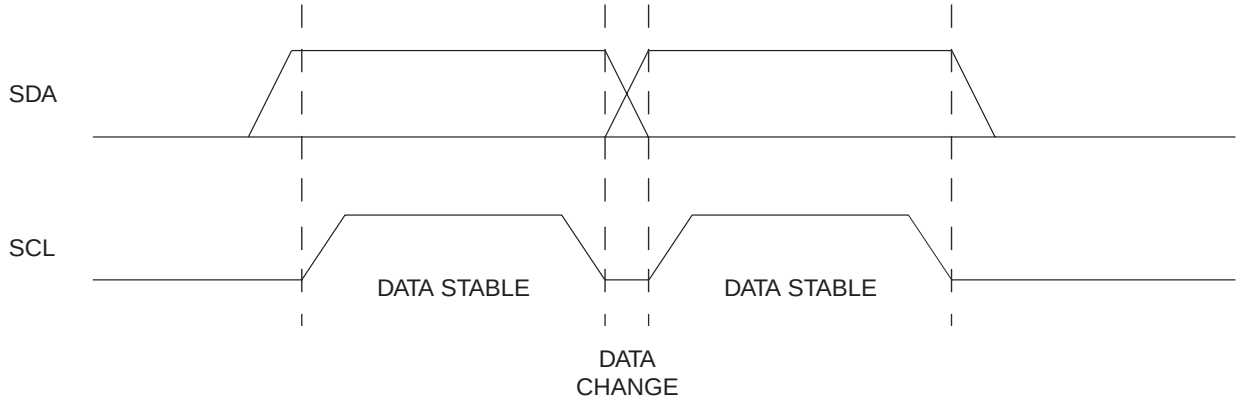
When considering a firmware-based I^2C driver it's important to realize that the intrinsic advantage of a synchronous protocol is its ability to vary the data transfer rate in accordance with the prevailing conditions. Asynchronous communication is far more problematic due to the inherent timing constraints and is better left to a hardware UART for all but the most trivial protocols. Given the choice, a hardware UART and firmware I^2C makes a lot of sense from a number of perspectives. Additionally, the AT89C2051's two-level priority interrupt structure, hardware UART, two external interrupts, and two 16-bit timer/counters with interrupt capability let you structure a system in a manner consistent with modern design practices. Using these resources, it is possible to implement all time-critical functions such as task scheduling, communications, system timing, and real-time event processing as interrupt service routines. This can result in a greatly simplified application program.

A Generic I^2C Driver

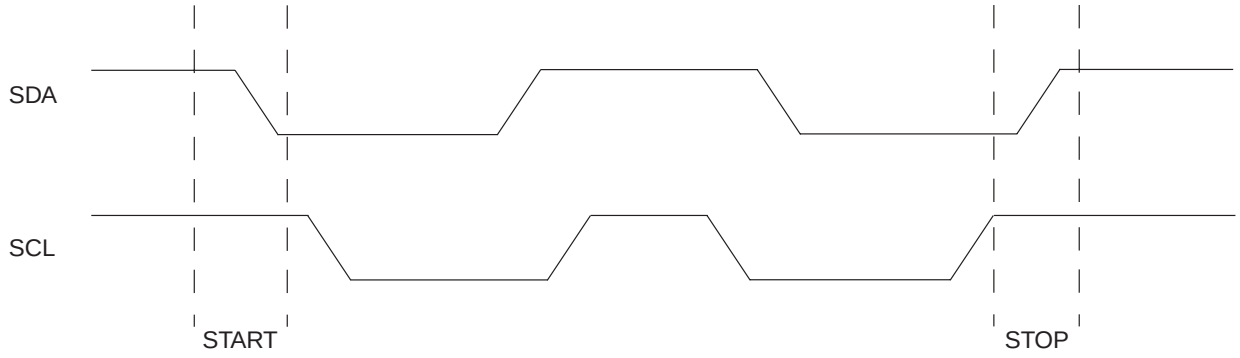
Although the basic functions of an I^2C driver can be partitioned in a seemingly endless number of ways, the approach adopted greatly limits the possible permutations without being overly restrictive. The driver module contains four user callable entry points for reading and writing to both simple and registered devices. In order to confine the number of variations, and to conserve code space, the registered I^2C support routines operate a byte at a time and, therefore, do not support streaming at the driver level.

Figure 1. Two-Wire Protocol Timing

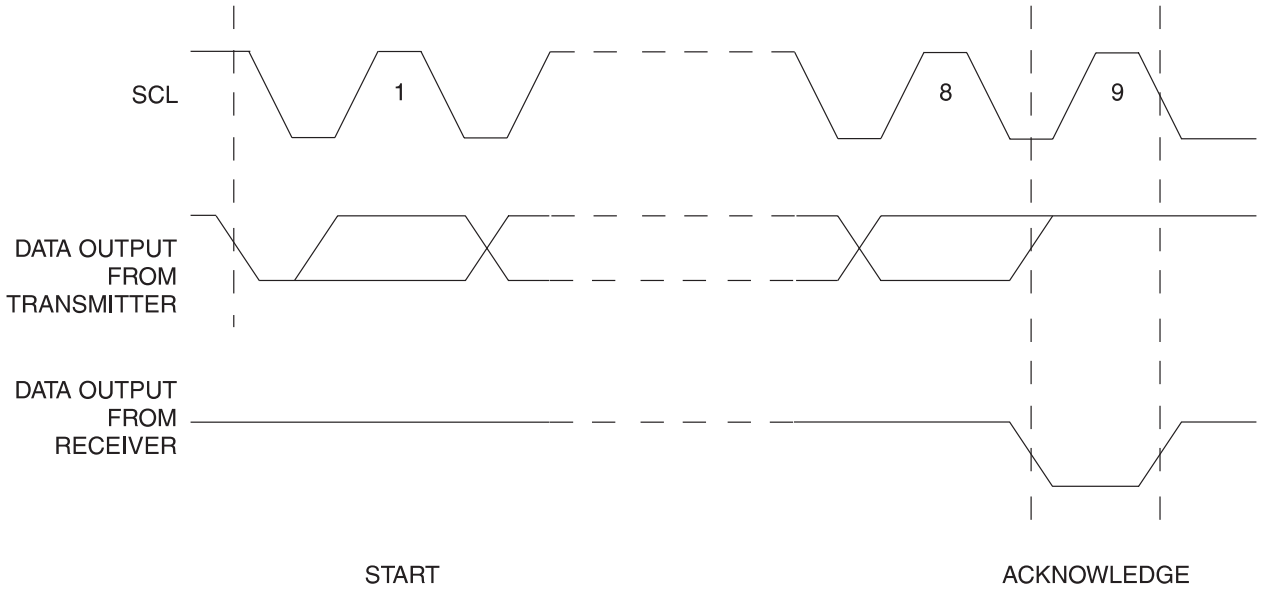
Data Validity



Start and Stop Definition



Acknowledge Response from Receiver



Although allowing the driver to directly handle multi-byte transfers would result in significantly increased throughput, speed often is not an issue. Instead, stream I/O is left to the device-specific second level functions on a need-to-do basis. In fact, a convincing case can be made for placing this type of functionality away from the driver. After all, in order for the driver to be truly generic it should, by definition, possess no device specific characteristics. As a specific example, consider I^2C EEPROMs. These come in a wide variety of memory densities with differently sized pages and different internal configurations. They all seem to possess minor peculiarities that become especially evident when performing sequential write operations. Obviously a driver specifically designed to deal effectively with a particular device becomes completely dysfunctional when used with a different one.

The assembly language support module is available on the Atmel Web Site or BBS. To conserve program memory all branching is performed using absolute address mode instructions which is adequate to fully navigate the 2K program space of the AT89C2051. Should it be desirable to operate this driver in a larger device simply substitute the absolute branch instructions with the corresponding long versions.

The module begins with a set of MACROs that establish several low level functions. These include a rudimentary bit delay, SCL control and synchronization, and I^2C START and STOP conditions. These are all implemented as instruction MACROs since their small size does not justify the overhead of a function call. Here, Bit_Delay consumes processor cycles to provide a short delay required to meet the basic 100-kHz I^2C timing parameters on a 12-MHz AT89C2051. Set_SCL releases the SCL line and synchronizes with slave devices that may be asserting clock-stretching wait states. Clr_SCL simply pulls the SCL line to a logic low while Emit_Clock invokes Set_SCL and Clr_SCL in sequence. Finally the Start and Stop MACROs embody the I^2C START and STOP conditions.

Next are two general purpose subroutines for transmitting and receiving bytes of data over the I^2C bus. These primitives are invoked by the public routines and are responsible for transporting data bytes while providing error checking and synchronization with some help from the previously defined MACROs. Xmit_Byte evaluates the acknowledge from the slave whereas Rec_Byte does not handle the acknowledge generation. When receiving, the acknowledgment is properly a function of the specific operation being performed and must be handled by the calling function. These subroutines communicate their completion status back to the higher functions through the carry flag.

Finally, the four public entry points appear that are accessible from the main application program. These include the transmit and receive routines for both registered and simple devices. These perform the requisite initial bus synchronization and make use of the previously defined subroutines and MACROs to orchestrate the requested operation. Status information, either from the called subroutine or generated locally, is conveyed to the application to indicate the completion status of the requested operation. Should a problem occur, it is up to the caller to sort it out. This makes sense since the appropriate response is often dependent on the type of device that is being accessed. For instance, fault status may simply mean that the device is not present or not responding. In the case of an EPROM it could indicate that a programming cycle is in progress. Obviously, these situations have quite different implications and should be handled differently.

Populating the Bus

Having defined the rudiments of the I^2C bus and now being in possession of a set of generic drivers, it's time to look at a typical peripheral set suitable for inclusion in a small embedded system. For all intents and purposes, the I^2C bus can be viewed as a scaled down version of a parallel bus such as used with a conventional microprocessor. As such, it provides a vehicle for the very same types of activities you'd perform using a standard bus structure.

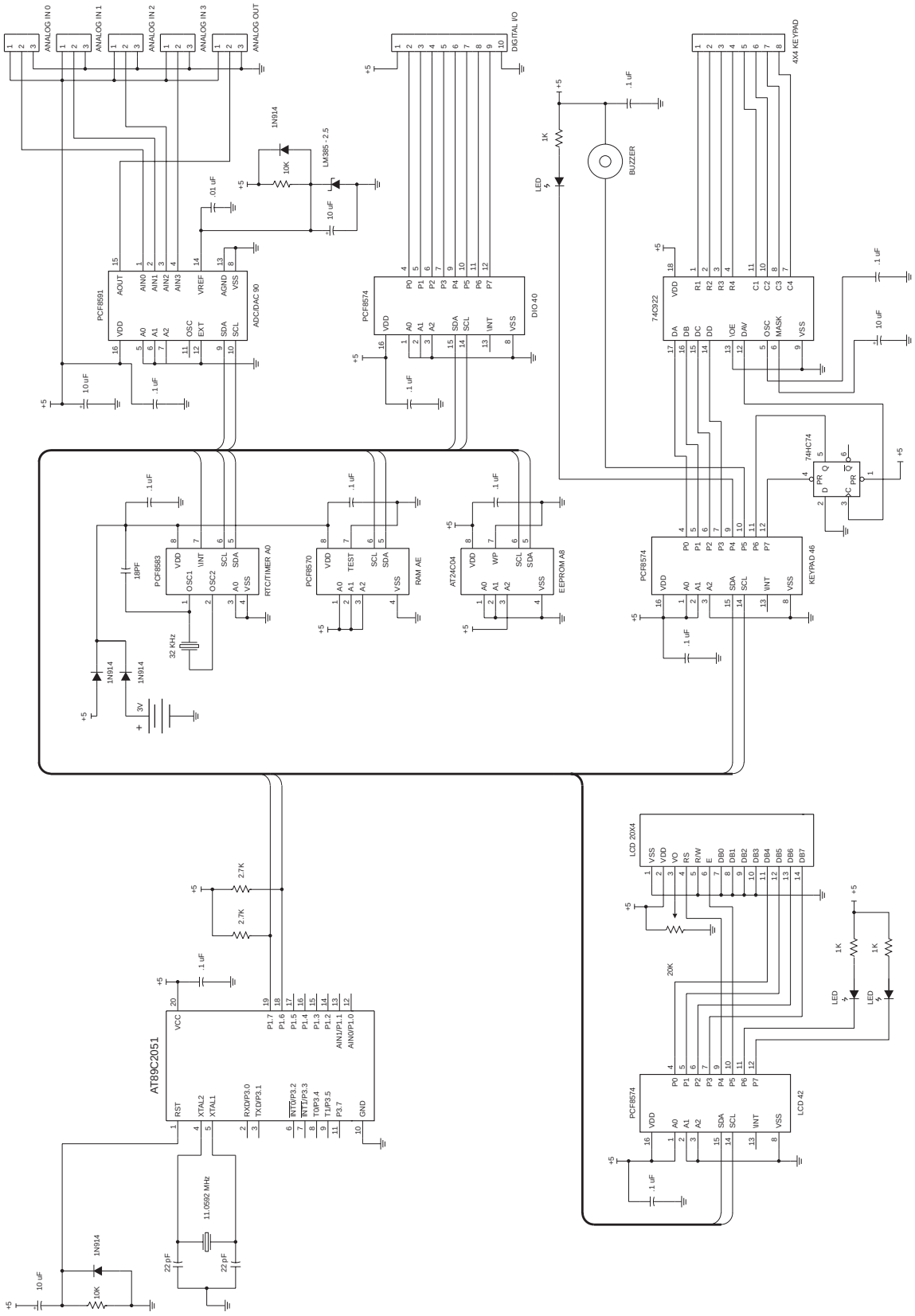
Figure 2 shows how an AT89C2051 based system could populate its 2-wire bus with some standard peripheral functions. Included are 8 bi-directional I/O points (PCF8574), 4 channels of 8-bit analog inputs and a single channel 8-bit analog output (PCF8591), a real time clock/calendar/timer with 256 bytes of nonvolatile RAM (PCF8583), 512 bytes of EEPROM (AT24C04), and 128 bytes of RAM (PCF8570). This represents a respectable function set applicable for many embedded applications.

A reliance on the previously defined I^2C drivers serves to conceal the protocol details from the calling functions and allows the application program to conceptually deal with the peripherals strictly as basic I/O devices. Access to these peripherals is considerably slower than with a standard parallel bus but, since the electrical interface consumes only two I/O pins, the controller retains most of its fast on-chip bit-addressable I/O for general use.

Console I/O

The 2-wire peripheral set also provides the functions of a user I/O interface panel. This includes 20 x 4 LCD, 4 x 4 matrix keypad, an audible beeper, and several indicator LEDs. This subsection is supported by two PCF8574 I^2C port expanders along with some additional support ICs. The LCD interfaces use one of the PCF8574s and operates in 4-bit mode as an output-only device. Although the PCF8574 can be operated bidirectionally, this would offer little advantage here and would unnecessarily complicate the code. The LCD interface requires only 6 I/O lines: a 4-bit data bus, a register select line (RS), and an enable line (E). The LCD's read/write line (RW) is hardwired to ground to permanently enable the write function.

Figure 2. Expanded AT89C2051 System



The keyboard circuit is a simple but effective implementation based on a 74C922 4 X 4 matrix self-scan IC and a flip-flop. Again, a PCF8574 serves as the interface port. Briefly, the 74C922 continuously scans the keyboard looking for a key closure. When a closure is detected, it is debounced and, if valid, the 74C922 places the corresponding binary key code on its data lines while asserting data available (DAV). Since DAV is only driven while a key is actually being depressed, the associated flip-flop records the event. This scheme works because, although the 74C922 may stop asserting DAV, its data lines continue to emit the key code until a new key stroke is detected and validated.

Second Level Drivers

Armed with a considerable set of peripheral devices and a set of low level I^2C drivers, a set of device-specific driver modules is now required to fully utilize the peripheral set. For clarity, these functions are written in C and are presented in the code file located on the Atmel Web Site or BBS.

Basically, two levels of support are provided that include basic drivers for the peripheral chips themselves and higher level functions that offer system-level services. Basic functions include rudimentary support for the PCF8574 parallel I/O port, PCF8591 analog converter, PCF8583 real time clock/calendar/timer with RAM, 24C04 EEPROM, and PCF8570 RAM. These are limited in scope and should be self explanatory.

LCD Gyrations and Library Hooks

When using a compiled language, it can be advantageous to implement user I/O as extensions to the standard library functions. Doing so, immediately presents a set of built-in capabilities that include character I/O, stream I/O, formatted I/O, and number conversion functions. The most effective way to hook into the libraries is to replace the device specific input/output function that falls at the end of the call chain. To this end, replacements are provided for PutChar and GetKey that furnish the low-level device interface to the 20 X 4 LCD and 4 X 4 keypad while retaining the familiar C language interface. With these in place, all standard library functions that utilize console I/O will operate using the local user I/O devices.

Most character mode LCDs are based on the HD44780 LSI. This LSI contains two internal registers defined as the command register and the data register. The command register receives initialization and set up information as well as functional commands to clear the LCD, set the cursor, select the cursor appearance, etc. The data register is the destination of all displayable data.

The Putchar function begins by first checking the input argument for a newline character. If the character is a newline, the cursor is advanced to the first position of the next line. If the cursor is on the last line it wraps back to the first. The cursor location is mirrored using a global variable that is updated by any operation that modifies the location of the cursor. If the input argument is any character other than a newline it is written directly to the LCD.

Special support functions are provided for writing to the LCD's data and command registers. Both DataWr and CommandWr dismember their input argument prior to dispatching it to the LCD in nibble mode. The DataWr function additionally interrogates the global variable Cursor in order to determine if it must take corrective action to maintain the visual output in a sequential fashion. That is, it determines if logical-to-physical cursor translation is required at points where discontinuity would occur.

Additional functions are furnished for direct cursor positioning with cursor fixup (PositionCursor), selecting an invisible; underline; or blinking cursor (SelectCursor), and clearing the LCD and homing the cursor (ClearLcd). An initialization function places the LCD into 4-bit mode and sets various operational parameters to their default values (InitLcd).

Whereas Putchar handles console output, Getkey is responsible for console input. While Getkey will wait indefinitely for a key to be pressed, few embedded designs will tolerate the suspension of all processing while waiting for a key stroke. The CheckKey function handles this situation more reasonably. Here, the function will return null if no data is available, otherwise the corresponding ASCII code is returned.

Basically, if a keystroke is available, CheckKey falls through and issues an acknowledge to clear the data-available flip-flop. While generating this pulse, the beeper is briefly enabled resulting in an audible key click that offers feedback that is particularly useful when a membrane keypad is used. InitKey is used to clear the keypad interface logic on power-on and can also be used to flush unneeded keystrokes from the interface logic.

Console support is rounded out with the inclusion of a function that emits beep sounds of variable length.

Driver Test Drive

Having presented a formidable array of 2-wire peripherals, it would be enlightening to verify that they behave as expected when accessed from a typical application program. Admittedly, most real embedded systems wouldn't use all these peripherals simultaneously in a given application. I^2C, however, is quite amenable to supporting a multiplicity of peripheral combinations without burdening the processor's I/O pin budget. A system requiring a single EEPROM is served equally well as one that needs an array of analog data converters or a mix of functions.

Although obviously a contrived design, the program shown in the code file on the Atmel Web Site or BBS, is intended to illustrate how the existing support code can be used to access the peripherals. The main module takes control immediately following the low-level start up code and first

performs the requisite device and variable initialization. Following this, the program enters into an infinite loop where the I^2C peripherals are accessed in a continual manner.

The initial block implements a buffered keypad that scrolls data onto the keypad field of the LCD panel. To make this somewhat mundane function a bit more attractive, keystrokes are scrolled in from right to left. The enter key causes the program to evaluate the entry for a legal numeric input in the range of 0 to 255. If it falls within these bounds it is dispatched to the analog output channel and is emitted as a voltage.

Next, the digital data port is set to an incrementing pattern. Since the port is bi-directional, it can be read as well. A read of this digital port is performed and the retrieved data is converted to binary notation and displayed in the digital data field of the LCD. In a similar manner, the four analog channels are read and converted to decimal ASCII and placed in the LCD's analog data field. Finally, the BCD date and time is read from the RTC and is translated to decimal ASCII notation and is shown in the LCD's RTC field.

This example not only shows that the peripherals are operating properly but also illustrates how easily the LCD can be effectively manipulated now that the support services are encapsulated with clearly defined input parameters. All of a sudden the tiny AT89C2051 takes on some of the attributes, and amenities, you would normally expect from a much larger computing engine.

The Adapting Controller

This application note has presented peripheral expansion techniques that, while using just two I/O pins, can provide a wide array of useful functions. If your needs are specialized, it is possible to support non I^2C serial devices by utilizing the already defined SDA and SCL lines. Half duplex Microwire peripherals will only require one additional pin whereas those that need independent transmit and receive pins will require the addition of two more. In many cases, complex functions such as multi-channel, high-resolution data converters can be picked up at the expense of just one extra pin.

Using the techniques shown, the AT89C2051 can be equipped with a peripheral set commensurate with its respectable computational capabilities. I^2C offers the inherent extensibility that can be instrumental in allowing a basic design to take on new capabilities and new features as additional requirements develop.

All the code for this application note can be found on the Atmel Web Site or BBS.