APPLICATION NOTE

ABSTRACT
Starts with an overview of I^2C basics including functions of master & slave, data transfers, addressing and transfer formats and use of sub-addresses. Next, the I^2C hardware features of the 87LPC76X are described including control, data and configuration registers and Timer I. Finally, a single-master ASM programming example is presented which includes send and receive routines with error recovery.

AN464

Using the 87LPC76X microcontroller as an I^2C bus master

2000 Jan 11

Philips Semiconductors

PHILIPS

Using the 87LPC76X microcontroller as an I²C bus master

AN464

DESCRIPTION

The 87LPC76X Microcontroller offers the advantages of the 80C51 architecture in a small package and at a low cost. It combines the benefits of a high-performance microcontroller with on-board hardware supporting the Inter-Integrated Circuit (I²C) bus interface.

The I²C bus, developed and patented by Philips, allows integrated circuits to communicate directly with each other via a simple bidirectional 2-wire bus. The comprehensive family of CMOS and bipolar ICs incorporating the on-chip I²C interface offers many advantages to designers of digital control for industrial, consumer and telecommunications equipment. A typical system configuration is shown in Figure 1.

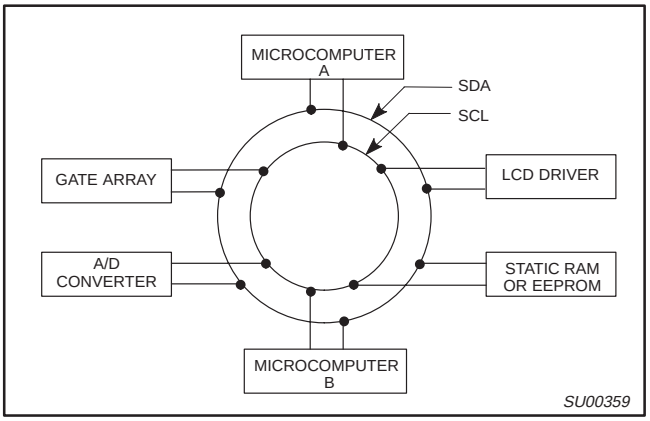

Figure 1. Typical I²C Bus Configuration

Interfacing the devices in an I²C based system is very simple because they connect directly to the two bus lines: a serial data line (SDA) and a serial clock line (SCL). System design can rapidly progress from block diagram to final schematic, as there is no need to design bus interfaces, and functional blocks on a block diagram correspond to actual ICs. A prototype system or a final product version can easily be modified or upgraded by 'clipping' or 'unclipping' ICs to or from the bus. The simplicity of designing with the I²C bus does not reduce its effectiveness; it is a reliable, multimaster bus with integrated addressing and data-transfer protocols (see Figure 2). In addition, the I²C-bus compatible ICs provide cost reduction benefits to equipment manufacturers, some of which are smaller IC packages and a minimization of PCB traces and glue logic.

The availability of microcontrollers like the 87LPC76X, with on-board I²C interface, is a very powerful tool for system designers. The integrated protocols allow systems to be completely software defined. Software development time of different products can be reduced by assembling a library of reusable software modules. In addition, the multimaster capability allows rapid testing and alignment of end-products via external connections to an assembly-line computer.

The mask programmable 87LPC76X and its EPROM version, the 87LPC76X, can operate as a master or a slave device on the I²C small area network. In addition to the efficient interface to the dedicated function ICs in the I²C family, the on-board interface facilities I/O and RAM expansion, access to EEPROM and processor-to-processor communications.

The multimaster capability of the I²C is very important but many designs do not require it. For many systems, it is sufficient that all communications between devices are initiated by a single, master processor. In this application note, use of the 87LPC76X as an I²C bus master is described. Some of the technical features of the bus and the 87LPC76X's special hardware associated with the I²C are discussed. Also included is a software example demonstrating I²C single master communications. Note that the sample routines are quite general, and therefore may be transferred easily to many applications.

The discussion of the I²C bus characteristics in this application note is by no means complete. Additional information for the I²C bus and the 87LPC76X Microcontroller can be found in the 80C51 Microcontroller databook.

THE I²C BUS

The two lines of the I²C-bus are a serial data line (SDA) and a serial clock line (SCL). Both lines are connected to a positive supply via a pull-up resistor, and remain HIGH when the bus is not busy. Each device is recognized by a unique address—whether it is a microcomputer, LCD driver, memory or keyboard interface—and can operate as either a transmitter or receiver, depending on the function of the device. A device generating a message or data is a transmitter, and a device receiving the message or data is a receiver. Obviously, a passive function like an LCD driver could only be a receiver, while a microcontroller or a memory can both transmit and receive data.

Masters and Slaves

When a data transfer takes place on the bus, a device can either be a master or a slave. The device which initiates the transfer, and generates the clock signals for this transfer, is the master. At that time any device addressed is considered a slave. It is important to note that a master could either be a transmitter or a receiver; a master microcontroller may send data to a RAM acting as a transmitter, and then interrogate the RAM for its contents acting as a receiver—in both cases performing as the master initiating the transfer. In the same manner, a slave could be both a receiver and a transmitter.

The I²C is a multimaster bus. It is possible to have, in one system, more than one device capable of initiating transfers and controlling the bus (Figure 2). A microcontroller may act as a master for one transfer, and then be the slave for another transfer, initiated by another processor on the network. The master/slave relationships on the bus are not permanent, and may change on each transfer.

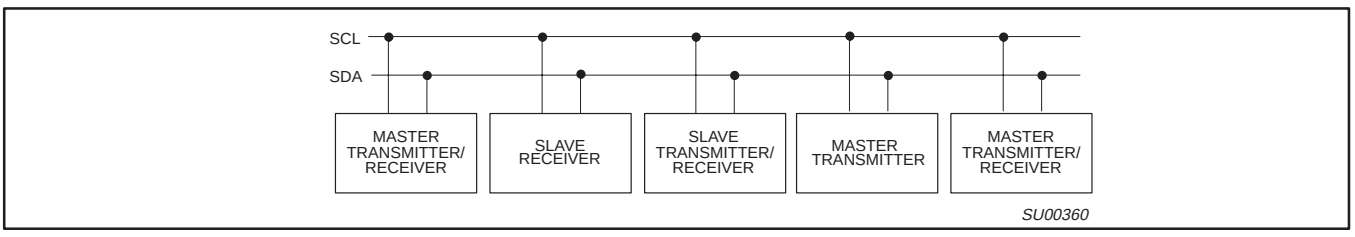

Figure 2. I²C Bus Connection

As more than one master may be connected to the bus, it is possible that two devices will try to initiate a transfer at the same time. Obviously, in order to eliminate bus collisions and communications chaos, an arbitration procedure is necessary. The I²C design has an inherent arbitration and clock synchronization procedure relying on the wired-AND connection of the devices on the bus. In a typical multimaster system, a microcontroller program should allow it to gracefully switch between master and slave modes and preserve data integrity upon loss of arbitration. In this note, a simple case is presented describing the 87LPC76X operating as a single master on the bus.

Data Transfers

One data bit is transferred during each clock pulse (see Figure 3). The data on the SDA line must remain stable during the HIGH period of the clock pulse in order to be valid. Changes in the data line at this time will be interpreted as control signals. A HIGH-to-LOW transition of the data line (SDA) while the clock signal (SCL) is HIGH indicates a Start condition, and a LOW-to-HIGH transition of the SDA while SCL is HIGH defines a Stop condition (see Figure 4). The bus is considered to be busy after the Start condition and free again at a certain time interval after the Stop condition. The Start and Stop conditions are always generated by the master.

The number of data bytes transferred between the Start and Stop condition from transmitter to receiver is not limited. Each byte, which must be eight bits long, is transferred serially with the most significant bit first, and is followed by an acknowledge bit. (see Figure 5). The clock pulse related to the acknowledge bit is generated by the master. The device that acknowledges has to pull down the SDA line during the acknowledge clock pulse, while the transmitting device releases the SDA line (HIGH) during this pulse (see Figure 6).

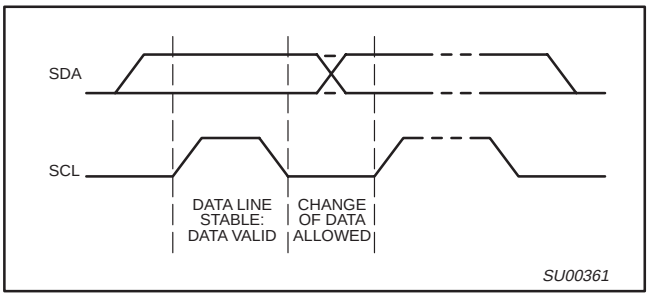

Figure 3. Bit Transfer on the I²C Bus

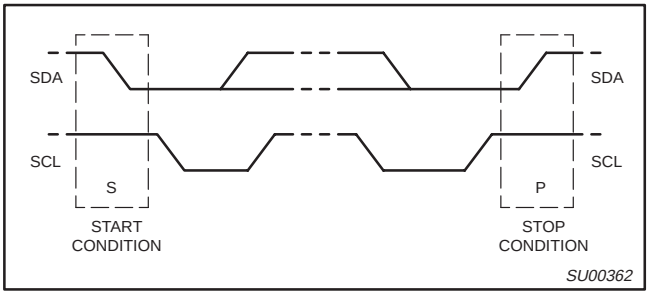

Figure 4. Start and Stop Conditions

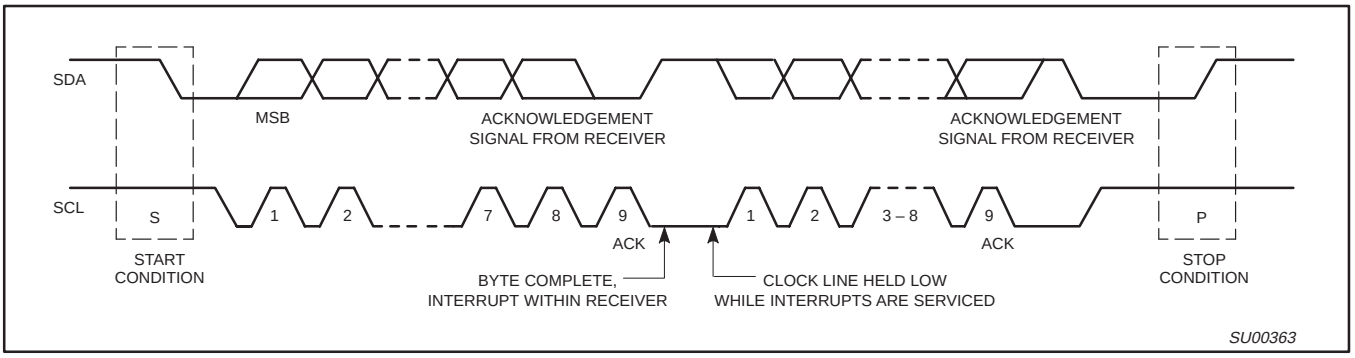

Figure 5. Data Transfer on the I²C Bus

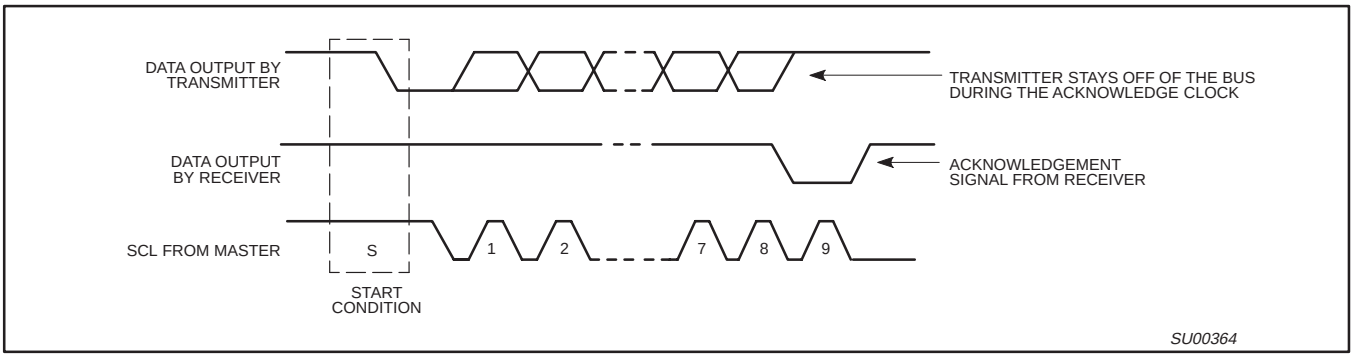

Figure 6. Acknowledge on the I²C Bus

A slave receiver must generate an acknowledge after the reception of each byte, and a master must generate one after the reception of each byte clocked out of the slave transmitter. If a receiving device cannot receive the data byte immediately, it can force the transmitter into a wait state by holding the clock line (SCL) LOW. When designing a system, it is necessary to take into account cases when acknowledge is not received. This happens, for example, when the addressed device is busy in a real time operation. In such a case the master, after an appropriate "time-out", should abort the transfer by generating a Stop condition, allowing other transfers to take place. These "other transfers" could be initiated by other masters in a multimaster system, or by this same master.

There are two exceptions to the "acknowledge after every byte" rule. The first occurs when a master is a receiver: it must signal an end of data to the transmitter by NOT signalling an acknowledge on the last byte that has been clocked out of the slave. The acknowledge related clock, generated by the master should still take place, but the SDA line will not be pulled down. In order to indicate that this is an active and intentional lack of acknowledgement, we shall term this special condition as a "negative acknowledge".

The second exception is that a slave will send a negative acknowledge when it can no longer accept additional data bytes. This occurs after an attempted transfer that cannot be accepted.

The bus design includes special provisions for interfacing to microprocessors which implement all of the I2C communications in software only—it is called "Slow Mode". When all of the devices on the network have built-in I2C hardware support, the Slow Mode is irrelevant.

Addressing and Transfer Formats

Each device on the bus has its own unique address. Before any data is transmitted on the bus, the master transmits on the bus the address of the slave to be accessed for this transaction. A well-behaved slave with a matching address, if it exists on the network, should of course acknowledge the master's addressing. The addressing is done by the first byte transmitted by the master after the Start condition.

An address on the network is seven bits long, appearing as the most significant bits of the address byte. The last bit is a direction (R/W) bit. A zero indicates that the master is transmitting (WRITE) and a one indicates that the master requests data (READ). A complete data transfer, comprised of an address byte indicating a WRITE and two data bytes is shown in Figure 7.

When an address is sent, each device in the system compares the first seven bits after the Start with its own address. If there is a match, the device will consider itself addressed by the master, and will send an acknowledge. The device could also determine if in this transaction it is assigned the role of a slave receiver or slave transmitter, depending on the R/W bit.

Each node of the I2C network has a unique seven bit address. The address of a microcontroller is of course fully programmable, while peripheral devices usually have fixed and programmable address portions. In addition to the "standard" addressing discussed here, the I2C bus protocol allows for "general call" addressing and interfacing to CBUS devices.

When the master is communicating with one device only, data transfers follow the format of Figure 7, where the R/W bit could indicate either direction. After completing the transfer and issuing a Stop condition, if a master would like to address some other device on the network, it could of course start another transaction, issuing a new Start.

Another way for a master to communicate with several different devices would be by using a "repeated start". After the last byte of the transaction was transferred, including its acknowledge (or negative acknowledge), the master issues another Start, followed by address byte and data—without effecting a Stop. The master may communicate with a number of different devices, combining READS and WRITES. After the last transfer takes place, the master issues a Stop and releases the bus. Possible data formats are demonstrated in Figure 8. Note that the repeated start allows for both change of a slave and a change of direction, without releasing the bus. We shall see later on that the change of direction feature can come in handy even when dealing with a single device.

In a single master system, the repeated start mechanism may be more efficient than terminating each transfer with a Stop and starting again. In a multimaster environment, the determination of which format is more efficient could be more complicated, as when a master is using repeated starts it occupies the bus for a long time and thus preventing other devices from initiating transfers.

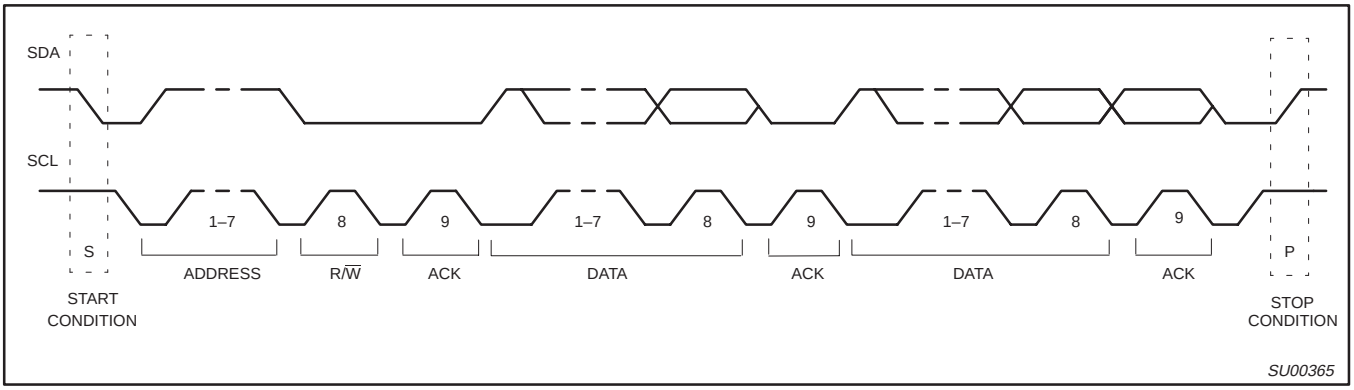

Figure 7. A Complete Data Transfer on the I2C-Bus

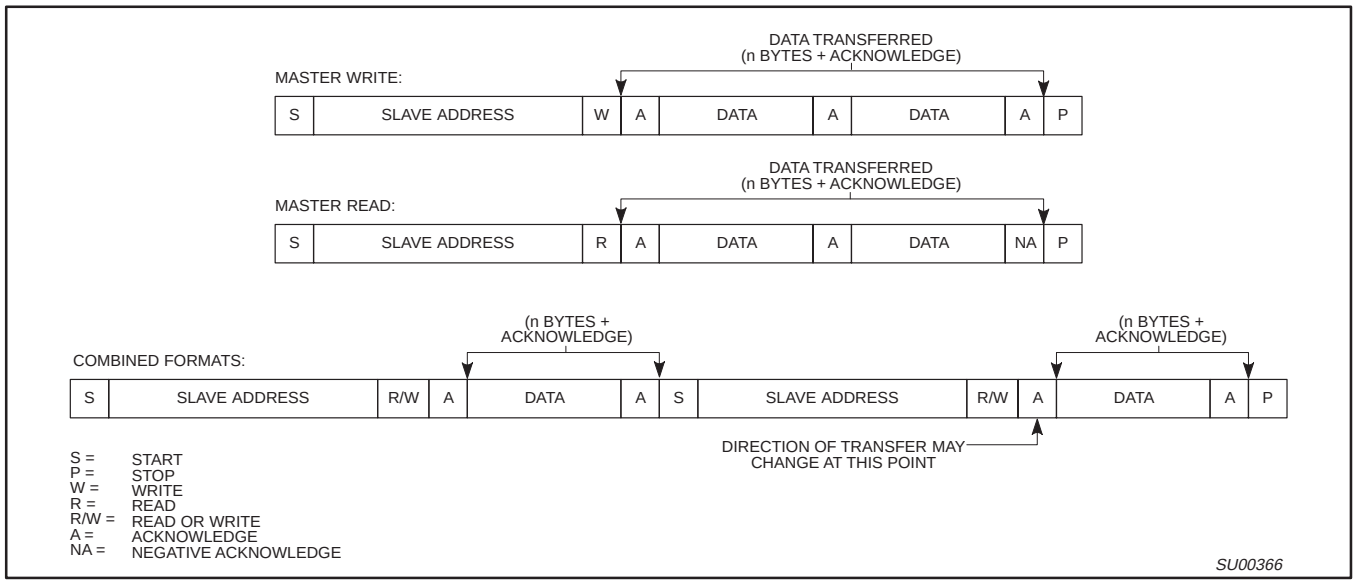

Figure 8. I²C Data Formats

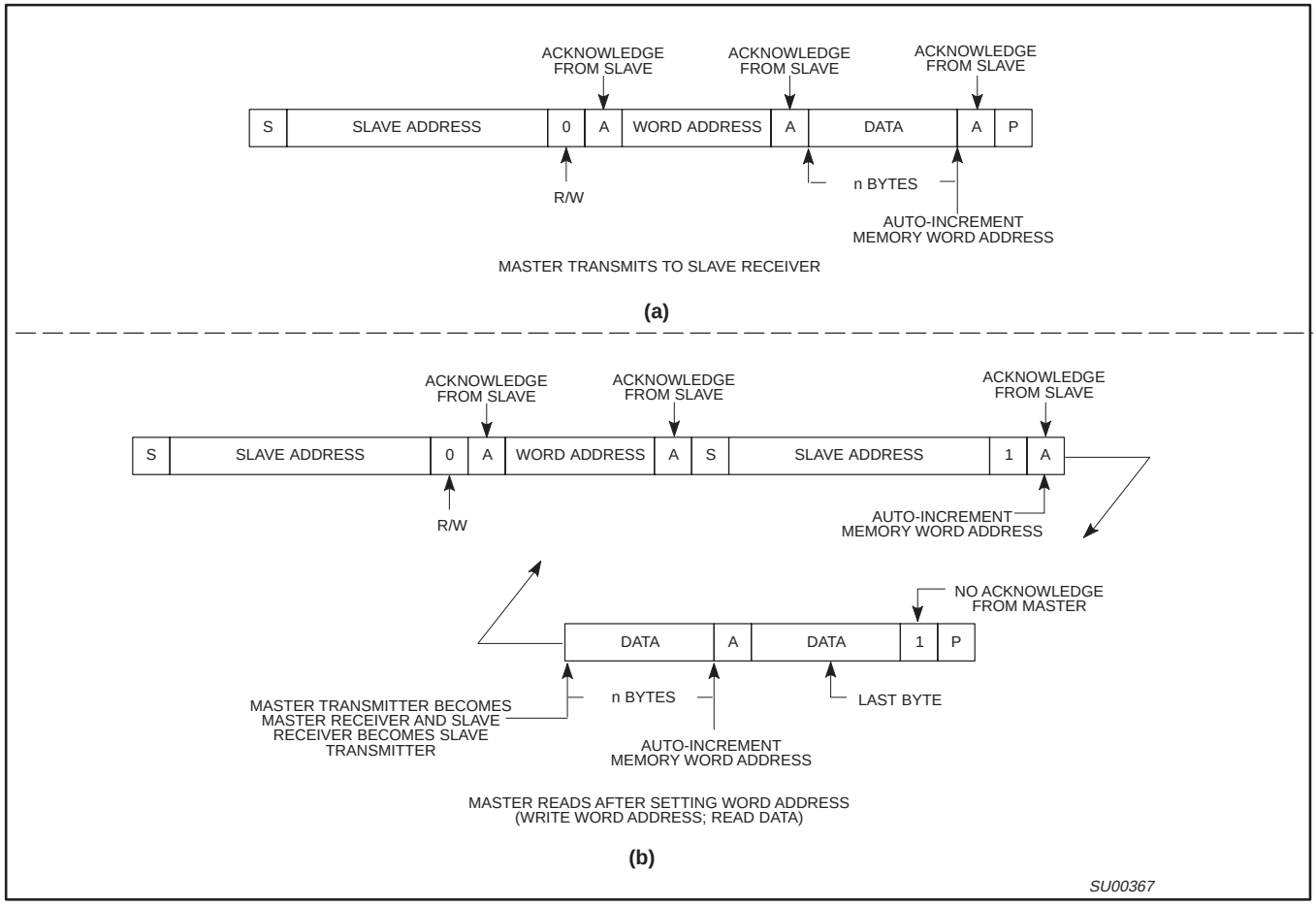

Figure 9. I²C Sub-Address Usage

Use of Sub-Addresses

For some ICs on the I²C bus, the device address alone is not sufficient for effective communications, and a mechanism for addressing the internals of the device is necessary. A typical example when we want to access a specific word inside the device is addressing memories, or a sequence of memory locations starting at a specific internal address.

A typical I²C memory device like the PCF8570 RAM contains a built-in word address register that is incremented automatically after each data byte which is a read or written data byte. When a master communicates with the PCF8570 it must send a sub-address in the byte following the slave address byte. This sub-address is the internal address of the word the master wants to access for a single byte transfer, or the beginning of a sequence of locations for a multi-byte transfer. A sub-address is an 8-bit byte, unlike the device address, it does not contain a direction (R/W) bit, and like any byte transferred on the bus it must be followed by an acknowledge.

A memory write cycle is shown in Figure 9(a). The Start is followed by a slave byte with the direction bit set to WRITE, a sub-address byte, a number of data bytes and a Stop signal. The sub-address is loaded into the word address memory, and the data bytes which follow will be written one after the other starting with the sub-address location, as the register is incremented automatically.

The memory read cycle (see Figure 9(b)) commences in a similar manner, with the master sending a slave address with the direction bit set to WRITE with a following sub-address. Then, in order to reverse the direction of the transfer, the master issues a repeated Start followed again by the memory device address, but this time with the direction bit set to READ. The data bytes starting at the internal sub-address will be clocked out of the device, each followed by a master-generated acknowledge. The last byte of the read cycle will be followed by a negative acknowledge, signalling the end of transfer. The cycle is terminated by a Stop signal.

87LPC76X I²C HARDWARE

The on-chip I²C bus hardware support of the 87LPC76X allows operation on the bus at full speed, and simplifies the software needed for effective communications on the network. The hardware activates and monitors the SDA and SCL lines, performs the necessary arbitration and framing errors checks, and takes care of clock stretching and synchronization. The hardware support includes a bus time-out timer, called Timer I. The hardware is synchronized to the software either through polled loops or interrupts.

Two of the port 0 pins are multi-functional. When the I²C is active, the pin associated with P0.0 functions as SCL, and the pin associated with P0.1 functions as SDA. These pins have an open drain output.

Two of the five 87LPC76X interrupt sources may be used for I²C support. The I²C interrupt is enabled by the EI2 flag of the interrupt enable register, and its service routine should start at address 023h. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates a new data bit on the bus, or a special condition occurs: Start, Stop or arbitration loss. The interrupt is induced by the ATN flag—see below for the conditions for setting this flag. The Timer I overflow interrupt is enabled by the ETI flag, and the service routine starts at 01Bh.

The I²C port is controlled through three special function registers: I²C Control (I2CON), I²C Configuration (I2CFG), and I²C Data (I2DAT). The register addresses are shown in Table 1.

Although the following discussion of the hardware and register details is not complete, it should give a better understanding of the programming examples.

Timer I

In I²C applications, Timer I is dedicated to the port timing generation and bus monitoring. In non-I²C applications, it is available for use as a fixed time base.

In its port timing generation function, Timer I is used to generate SCL, the I²C clock. Timer I is clocked once per machine cycle (osc/12), so that the toggle rate of SCL will be some multiple of that rate. Because the 87LPC76X can be run over a wide range of oscillator frequencies, it is necessary to adjust SCL for the part's oscillator frequency. This allows the I²C bus to be used at its highest transfer rates independent of the oscillator frequency. SCL is adjusted by writing to two bits (CT0 and CT1) in the I2CFG special function register (see Table 2). The inverse of the values in CT0 and CT1 are loaded into the least significant two bit locations of Timer I every time the fourth bit of the timer is toggled. (A value is actually loaded into the least significant three bits, the third bit being 0 unless both CT0 and CT1 are programmed high and in that case the third bit is 1). SCL is then toggled every time the fourth bit of Timer I is toggled. For example: if CT1 = 0 and CT0 = 1 then the least significant three bits of Timer I would be preloaded with 2 (010 binary). Timer I would then count 3, 4, 5, 6, 7, 8 (6 counts or machine cycles). On 8, the fourth bit of Timer I will toggle, SCL will toggle and the 3 least significant bits will again be preloaded with the value 2 (010).

Table 1. I²C Special Function Register Addresses

REGISTER			BIT ADDRESS							
Name	**Symbol**	**Address**	**MSB**							**LSB**
I²C Control	I2CON	D8	DF	DE	DD	DC	DB	DA	D9	D8
I²C Data	I2DAT	D9	–	–	–	–	–	–	–	–
I²C Configuration	I2CFG	C8	CF	CE	CD	CC	CB	CA	C9	C8

Table 2. CT1, CT0 Values

CT1, CT0	Min Time Count (Machine Cycles)	CPU Clock Max (for 100 kHz I²C)	Timeout Period (Machine Cycles)
1 0	7	8.4 MHz	1023
0 1	6	7.2 MHz	1022
0 0	5	6.0 MHz	1021
1 1	4	4.8 MHz	1020

For the bus monitoring function, Timer I is used as a "watchdog timer" for bus hang-ups. It creates an interrupt when the SCL line stays in one state for an extended period of time while the bus is active (between a Start condition and a following Stop condition). SCL "stuck low" indicates a faulty master or slave. SCL "stuck high" may mean a faulty device, or that noise induced unto the I²C caused all masters to withdraw from the I²C arbitration.

The time-out interval of Timer I is fixed (cannot be set): it carries out and interrupts (if enabled) when about 1024 machine cycles have elapsed since a change on SCL within a frame. In other words, whenever I²C is active and Timer I is enabled, the falling edge of SCL will reset Timer I. If SCL is not toggled low for 1024 machine cycles, Timer I will overflow and cause an interrupt. (Note: we wrote "about 1024 machine cycles" although for the sake of accuracy—this number is affected by the setting of the CT0 and CT1 bits mentioned above and may vary by up to three machine cycles) The exact number of cycles for a time-out is not critical; what is important is that it indicates SCL is stuck.

In addition to the interrupt, upon Timer I overflow the I²C port hardware is reset. This is useful for multiple master systems in situations where a bus fault might cause the bus to hang-up due to a lack of software response. When this happens, SCL will be released, and I²C operation between other devices can continue.

I2CON Register

The I²C control register (I2CON) can be written to (see Figure 10). When writing to the I2CON register, one should use bit masks as demonstrated in the example program. Trying to clear or set the bits in the register using the bit addressing capabilities of the 87LPC76X may lead to undesirable results. The reason is that a command like CLR reads the register, sets the bit and writes it back, and the write-back may affect other bits.

I2CFG Register

The configuration register (I2CFG) is a read/write register (see Figure 11).

I2DAT Register

The I²C data register (I2DAT) is a read/write register, where the MSB represents the data received or data to be sent. The other seven bits are read as 0 (see Figure 12).

Transmit Active State

The transmit active state—Xmit Active—is an internal state in the I²C interface that is affected by the I²C registers as explained above. The I²C interface will only drive the SDA line low when Xmit Active is set. Xmit Active is set by writing the I2DAT register, or by writing I2CON with XSTR = 1 or XSTP = 1. The ARL bit will be set to 1 only when Xmit Active is set—in such a case Xmit Active will be automatically reset upon arbitration loss. Xmit Active is cleared by writing 1 to CXA at I2CON register or by reading the I2DAT register.

I2CON READ

RDAT	ATN	DRDY	ARL	STR	STP	MASTER	——

RDAT	Received DATa bit. The value of SDA latched by the rising edge of SCL. Its contents is identical to RDAT in the I2DAT register. Reading the received data here allows doing so without clearing DRDY and releasing SCL.
ATN	An "ATteNtion" flag, set when any one of DRDY, ARL, STR or STP is set. This flag allows a single bit testing for terminating "wait loops", indicating a meaningful event on the bus. This flag also activates the I²C interrupt request.
DRDY	Data ReaDY flag. Set by a rising edge of SCL when I²C is active, except at an idle slave. This flag is cleared by reading or writing the I2DAT register, or by writing a 1 to CDR (at the same address, when I2CON is written).
ARL	ARbitration Loss flag. Indicates that this device lost arbitration while trying to take control of the bus.
STR	STaRt flag. Set when a Start condition is detected, except at an idle slave.
STP	SToP flag. Set when a Stop condition is detected, except at an idle slave.
MASTER	This flag is set when the controller is a bus master (or a potential master, prior to arbitration).

I2CON WRITE

CXA	IDLE	CDR	CARL	CSTR	CSTP	XSTR	XSTP

CXA	"Clear Xmit Active". Writing a 1 to CXA clears the internal transmit-active state.
IDLE	Setting this bit will cause a slave to enter idle mode and ignore the I²C bus until the next Start is detected. If the software sets the MASTRQ flag, the device may stop idling by turning into a master.
CDR	Clear Data Ready. Clears the DRDY flag.
CARL	Clear Arbitration Lost. Clears the ARL flag.
CSTR	Clear STaRt. Clears the STR flag.
CSTP	Clear STop. Clears the STP flag.
XSTR	"Xmit repeated STaRt". Writing a 1 to this bit causes the hardware to issue a Repeated Start signal. A side effect will be setting the internal Xmit Active state. This should be used only when the device is a master.
XSTP	"Xmit STop". Issues a Stop condition. The Xmit active state is set.

SU00368

Figure 10. I2CON Register

SLAVEN	MASTRQ	CLRTI	TIRUN	—	—	CT1	CT0

SLAVEN Writing a 1 to this flag enables the slave functions of the I²C interface.

MASTRQ Request control of the bus as a master.

CLRTI Clear the Timer I interrupt flag. This bit is always read as 0.

TIRUN Writing a 1 will let Timer I run. When I²C is active, it will run only inside frames, and will be cleared by SCL transitions, Start and Stop. Writing a 0 will stop and clear the timer.

CT1, CT0 These bits should be programmed according to the frequency of the crystal oscillator used in the hardware. They determine the minimum high and low times for SCL, and are used to optimized performance at different oscillator speeds.

SU00369

Figure 11. I2CFG Register

I2DAT READ	RDAT	—	—	—	—	—	—	—

RDAT Received DATa bit, captured from SDA every rising edge of SCL. Reading I2CAT clears DRDY and the Xmit Active state. If it is necessary to read the data without affecting the flags, it can be read out of RDAT in the I2CON register.

I2DAT WRITE	XDAT	—	—	—	—	—	—	—

XDAT Xmit DATa bit. Writing XDAT determines the data for the next bit to be transmitted on the I²C bus. Writing I2DAT also clears DRDY and sets the Xmit Active state.

SU00370

Figure 12. I2DAT Register

PROGRAMMING EXAMPLE

The listing demonstrates communications routines for the 87LPC76X as an I²C bus master in a single-master system.

The single-master system is less complicated than a multimaster environment. The programmer does not have to worry about switching between master and slave roles, or the consequences of an arbitration loss.

The I²C interrupt is not used, and therefore disabled. There is no need for frame Start interrupts, as this processor is the only bus master and all data transfers are initiated by it when the appropriate routines are called by the application. No one else generates frame Starts which could be an interrupt source in a multimaster system. Within the frames we monitor bus activity with a wait-loop which polls the ATN flag. As we expect the bus to operate in its full-speed mode, we can assume that only a small amount of time will be wasted in those loops, and the use of interrupts would be less efficient.

The 87LPC76X has single-bit I²C hardware interface, where the registers may directly affect the levels on the bus and the software interacting with the register takes part in the protocol implementation. The hardware and the low-level routines dealing with the registers are tightly coupled. Therefore, one should take extra care if trying to modify these lower level routines.

The beginning of the program, at address 0, contains the reset vector, where the microcontroller begins executing code after a hardware reset. In this case, the code simply jumps to the main part of the program, which begins at the label ‘Reset’ near the end of the listing.

The main program is a simple demonstration of the I²C routines which comprise the balance of the listing. It first enables the Timer I interrupt, and sets up parameters in order to read data from a slave device. In this example, the slave device is a PCF8574A 8-bit I/O port that has pushbuttons connected to bits 3:0, and LEDs connected to bits 7:4. The program causes the I/O port data to be read by calling the “RcvData” routine. Once the data byte from the PCF8574A has been read, the pushbutton data is saved and copied to the LED bit position and the switch data set high. The program then prepares to write this new value to the PCF8574A I/O port, and performs the write operation by calling the “SendData” routine. The SendData and RcvData routines can send or receive multiple bytes of data, the number of which is determined by the variable ‘ByteCnt’.

Upon return from both SendData and RcvData, the program checks the system flag named ‘Retry’ to see if the transfer was completed correctly. If not, it loops back and attempts the same transfer again.

This entire process is repeated indefinitely by jumping back to MainLoop.

Back at the beginning of the program, the next location after the reset vector is the Timer I interrupt service routine. The microcontroller will go to address 73 hexadecimal if Timer I overflows. This routine stops the timer, clears the timer interrupt, clears the pending interrupt so that other interrupts will be enabled, restores the stack pointer, and jumps to the ‘Recover’ routine to try to correct whatever stopped the I²C bus and allowed Timer I to overflow.

Next in the listing come the main I²C service routines. These are the routines SendData, RcvData, SendSub, and RcvSub that were called from the main program. Both of the send routines use the data area labeled ‘XmtDat’ as the transmit data buffer. In this sample program, four bytes were reserved for this area, but it could be larger or

smaller depending on the application. The two receive routines use another four byte buffer labeled 'RcvDat' to store received data. All of these routines use the variables 'SlvAdr' and 'ByteCnt' to determine the slave address and the number of bytes to be sent or received, respectively. The SendSub and RcvSub routines use the variable 'SubAdr' as the sub-address to send to the slave device.

Following the main I2C service routines in the listing are the subroutines that are called by the main routines to deal intimately with the I2C hardware.

The 'SendAddr' subroutine requests mastership of the I2C bus and calls the routine 'XmitAddr' to complete sending the slave address. The bulk of the XmitAddr routine is shared with the 'XmitByte' subroutine which sends data bytes on the I2C bus. XmitByte is also used to send I2C sub-addresses. Both subroutines check for an acknowledge from the slave device after every byte is sent on the I2C bus.

The next subroutine 'RDAck' calls the 'RcvByte' routine to read in a byte of data. It then sends an acknowledge to the slave device. RDAck is used to receive all data except for the last byte of a receive data frame, where the acknowledge is omitted by the bus master. The RcvByte subroutine is called directly for the last byte of a frame.

The 'SendStop' subroutine causes a stop condition on the I2C, thus ending a frame. The 'RepStart' subroutine sends a repeated start condition on the I2C bus, to allow the master to start a new frame without first having to send an intervening stop.

The lower level subroutines deal directly with the hardware. The tight coupling between hardware and software is best demonstrated by the following explanations, relating to two cases in which the code is not self evident.

Sending the Address

When sending the address byte in the Send Addr subroutine, the first bit is written to I2DAT prior to the loop where the other seven bits are sent (SendAd2). The reason is that we need to clear the Start condition in order to release the SCL line, and this is done explicitly by the subsequent command. When SCL is released, the correct bit (MSB of address) must already be in I2DAT.

Capturing the Received Data

Typically, a program receiving data waits in a loop for ATN, and when detected, checks DRDY. If DRDY = 1 then there was a rising SCL, and the new data can be read from RDAT in I2CON or I2DAT. Reading or writing I2DAT clears DRDY, thus releasing SCL.

When reading the last bit in a byte, it should be read from I2CON, and not I2DAT (see the end of the RcvByte routine). This way the Data Ready (DRDY) flag is not cleared, and the low period on SCL is stretched. The reason for doing so is that upon reception of the last bit of a received byte the master must react with an acknowledge. In order to ensure that we "wait" with the acknowledge clock (release of SCL) until the acknowledge level is issued on SDA, the last bit is read out of I2CON and not I2DAT. SCL is stretched low until the acknowledge level is written into I2DAT by the software.

Bus Faults and Other Exceptions

Bus exceptions are detected either by Timer I time-out, or "illegal" logic states tested for and detected by the software. Upon Timer I time-out, a bus recovery is attempted by the Recover routine. The final section of the listing is this 'Recover' routine. Its job is to try to restore control of the I2C bus to the main program. First, the subroutine 'FixBus' is called. It checks to see if only the SDA line is 'stuck', and if so, tries to correct it by sending some extra clocks on the SCL line, and forcing a stop condition on the bus. If this does not work, another subroutine 'BusReset' is called. This generally happens when a severe bus error occurs, such as a shorted clock line. The philosophy used in this code is that the only chance of recovering from a severe error is to cause a reset of the I1C hardware by deliberately forcing Timer I to time out. This method allows recovery from a temporary short or other serious condition on the I2C bus.

```
;*************************************************************************

;                I2C Single Master Routines for the 87LPC764

; Modified from code published for the 8xC751/752 in AN422. This program
; reads an I2C slave device using subaddressing, alters the data, and
; returns it to the same slave.

;*************************************************************************

; Notes on 87LPC764 I2C differences:
;     – I2C interrupt vector address.
;     – Timer I interrupt vector address.
;     – IEN0 SFR name (IE on 751) and addition of IEN1.
;     – I2C interrupt enable location (now in IEN1 and a different bit).
;     – Timer I interrupt enable location (now in IEN1 and a different bit).
;     – I2C SFR addresses (altered by inclusion of the MOD764 file).

$DEBUG
$MOD764


; I2C Demo Board I2C Addresses

LCD     equ   74h         ; PCF8577 LCD display.
LED7    equ   76h         ; SAA1064 LED display.
RTCLK   equ   0A2h        ; PCF8583 clock calendar.
RAM     equ   0AEh        ; PCF8570 256 byte RAM.
EEPROM  equ   0A6h        ; PCF8582 256 byte EEPROM.
DTMF    equ   4Ah         ; PCD3312 DTMF generator.
PIO     equ   4Eh         ; PCF8574 8–bit I/O port.
KEYLED  equ   7Eh         ; PCF8574A that runs the discrete LEDs & keypad.
ADDAC   equ   09Eh        ; PCF8591 A/D and DAC.

; Value definitions.

CTVAL   equ   02h         ; CT1, CT0 bit values for I2C.

; Masks for I2CFG bits.

BTIR    equ   10h         ; mask for TIRUN bit.
BMRQ    equ   40h         ; mask for MASTRQ bit.

; Masks for I2CON bits.

BCXA    equ   80h         ; mask for CXA bit.
BIDLE   equ   40h         ; mask for IDLE bit.
BCDR    equ   20h         ; mask for CDR bit.
BCARL   equ   10h         ; mask for CARL bit.
BCSTR   equ   08h         ; mask for CSTR bit.
BCSTP   equ   04h         ; mask for CSTP bit.
BXSTR   equ   02h         ; mask for XSTR bit.
BXSTP   equ   01h         ; mask for XSTP bit.

; RAM locations used by I2C routines.

bitCnt  data  21h         ; I2C bit counter.
ByteCnt data  22h
SlvAdr  data  23h         ; address of active slave.
SubAdr  data  24h

RcvDat     data  25h         ; I2C receive data buffer (4 bytes).
                             ;   addresses 25h through 28h.
```

```
XmtDat     data  29h         ; I2C transmit data buffer (4 bytes).
                             ;   addresses 29h through 2Ch.

StackSave  data  2Dh         ; saves stack address for bus recovery.

Flags      data  20h         ; I2C software status flags.
NoAck      bit   Flags.0     ; indicates missing acknowledge.
Fault      bit   Flags.1     ; indicates a bus fault of some kind.
retry      bit   Flags.2     ; indicates that last I2C transmission
                             ;   failed and should be repeated.

;*************************************************************************
;                                 Begin Code
;*************************************************************************

; Reset and interrupt vectors

           org    0000h
           ajmp   Reset           ; reset vector.

           org    0073h           ; Timer I interrupt address.
TimerI:    setb   CLRTI           ; Clear timer I interrupt.
           clr    TIRUN
           acall  ClrInt          ; Clear interrupt pending.
           mov    SP,StackSave    ; Restore stack for return to main.
           ajmp   Recover         ; Attempt bus recovery.
ClrInt:    reti


           org    0100h


;*************************************************************************
;                    Main Transmit and Receive Routines
;*************************************************************************

; Send data byte(s) to slave.
;   Enter with slave address in SlvAdr, data in XmtDat buffer, # of data
;   bytes to send in ByteCnt.

Senddata:
           clr    NoAck           ; clear error flags.
           clr    Fault
           clr    retry
           mov    StackSave,SP    ; save stack address for bus fault.
           mov    A,SlvAdr        ; get slave address.
           acall  SendAddr        ; get bus and send slave address.
           jb     NoAck,SDEX      ; check for missing acknowledge.
           jb     Fault,SDatErr   ; check for bus fault.
           mov    R0,#XmtDat      ; set start of transmit buffer.

SDLoop:    mov    A,@R0           ; get data for slave.
           inc    R0
           acall  XmitByte        ; send data to slave.
           jb     NoAck,SDEX      ; check for missing acknowledge.
           jb     Fault,SDatErr   ; check for bus fault.
           djnz   ByteCnt,SDLoop

SDEX:      acall  SendStop        ; send an I2C stop.
           ret

; Handle a transmit bus fault.

SDatErr:   ajmp   Recover         ; attempt bus recovery.
```

```
; Receive data byte(s) from slave.
;   Enter with slave address in SlvAdr, # of data bytes requested in ByteCnt.
;   Data returned in RcvDat buffer.

Rcvdata:
         clr   NoAck             ; clear error flags.
         clr   Fault
         clr   retry
         mov   StackSave,SP      ; save stack address for bus fault.
         mov   A,SlvAdr          ; get slave address.
         setb  ACC.0             ; get bus read bit.
         acall SendAddr          ; send slave address.
         jb    NoAck,RDEX        ; check for missing acknowledge.
         jb    Fault,RDatErr     ; check for bus fault.

         mov   R0,#RcvDat        ; set start of receive buffer.
         djnz  ByteCnt,RDLoop    ; check for count = 1 byte only.
         sjmp  RDLast

RDLoop:  acall RDAck             ; get data and send an acknowledge.
         jb    Fault,RDatErr     ; check for bus fault.
         mov   @R0,A             ; save data.
         inc   R0
         djnz  ByteCnt,RDLoop    ; repeat until last byte.

RDLast:  acall RcvByte           ; get last data byte from slave.
         jb    Fault,RDatErr     ; check for bus fault.
         mov   @R0,A             ; save data.

         mov   I2DAT,#80h        ; send negative acknowledge.
         jnb   ATN,$             ; wait for NAK sent.
         jnb   DRDY,RDatErr      ; check for bus fault.

RDEX:    acall SendStop          ; send an I2C bus stop.
         ret

; Handle a receive bus fault.

RDatErr:
         ajmp  Recover           ; attempt bus recovery.

; Send data byte(s) to slave with subaddress.
;   Enter with slave address in ACC, subaddress in SubAdr, # of bytes to
;   send in ByteCnt, data in XmtDat buffer.

SendSub:
         clr   NoAck             ; clear error flags.
         clr   Fault
         clr   retry
         mov   StackSave,SP      ; save stack address for bus fault.
         mov   A,SlvAdr          ; get slave address.
         acall SendAddr          ; get bus and send slave address.
         jb    NoAck,SSEX        ; check for missing acknowledge.
         jb    Fault,SSubErr     ; check for bus fault.

         mov   A,SubAdr          ; get slave subaddress.
         acall XmitByte          ; send subaddress.
         jb    NoAck,SSEX        ; check for missing acknowledge.
         jb    Fault,SSubErr     ; check for bus fault.
         mov   R0,#XmtDat        ; set start of transmit buffer.

SSLoop:  mov   A,@R0             ; get data for slave.
         inc   R0
         acall XmitByte          ; send data to slave.
```

```
         jb    NoAck,SSEX        ; check for missing acknowledge.
         jb    Fault,SSubErr     ; check for bus fault.
         djnz  ByteCnt,SSLoop

SSEX:    acall SendStop          ; send an I2C stop.
         ret

; Handle a transmit bus fault.

SSubErr:
         ajmp  Recover           ; attempt bus recovery.

; Receive data byte(s) from slave with subaddress.
;   Enter with slave address in SlvAdr, subaddress in SubAdr, # of data bytes
;   requested in ByteCnt. data returned in RcvDat buffer.

RcvSub:  clr   NoAck             ; clear error flags.
         clr   Fault
         clr   retry
         mov   StackSave,SP      ; save stack address for bus fault.
         mov   A,SlvAdr          ; get slave address.
         acall SendAddr          ; send slave address.
         jb    NoAck,RSEX        ; check for missing acknowledge.
         jb    Fault,RSubErr     ; check for bus fault.

         mov   A,SubAdr          ; get slave subaddress.
         acall XmitByte          ; send subaddress.
         jb    NoAck,RSEX        ; check for missing acknowledge.
         jb    Fault,RSubErr     ; check for bus fault.

         acall RepStart          ; send repeated start.
         jb    Fault,RSubErr     ; check for bus fault.
         mov   A,SlvAdr          ; get slave address.
         setb  ACC.0             ; get bus read bit.
         acall SendAd2           ; send slave address.
         jb    NoAck,RSEX        ; check for missing acknowledge.
         jb    Fault,RSubErr     ; check for bus fault.

         mov   R0,#RcvDat        ; set start of receive buffer.
         djnz  ByteCnt,RSLoop    ; check for count = 1 byte only.
         sjmp  RSLast

RSLoop:  acall RDAck             ; get data and send an acknowledge.
         jb    Fault,RSubErr     ; check for bus fault.
         mov   @R0,A             ; save data.
         inc   R0
         djnz  ByteCnt,RSLoop    ; repeat until last byte.

RSLast:  acall RcvByte           ; get last data byte from slave.
         jb    Fault,RSubErr     ; check for bus fault.
         mov   @R0,A             ; save data.

         mov   I2DAT,#80h        ; send negative acknowledge.
         jnb   ATN,$             ; wait for NAK sent.
         jnb   DRDY,RSubErr      ; check for bus fault.

RSEX:    acall SendStop          ; send an I2C bus stop.
         ret

; Handle a receive bus fault.

RSubErr:
         ajmp  Recover           ; attempt bus recovery.
```

```
;*************************************************************************
;                                Subroutines
;*************************************************************************

; Send address byte.
;   Enter with address in ACC.

SendAddr: mov   I2CFG,#BMRQ+BTIR+CTVAL  ; request I2C bus.
          jnb   ATN,$             ; wait for bus granted.
          jnb   Master,SAErr      ; should have become the bus master.
SendAd2:  mov   I2DAT,A           ; send first bit, clears DRDY.
          mov   I2CON,#BCARL+BCSTR+BCSTP ; clear start, releases SCL.
          acall XmitAddr          ; finish sending address.
          ret

SAErr:    setb  Fault             ; return bus fault status.
          ret

; Byte transmit routine.
;   Enter with data in ACC.
;   XmitByte : transmits 8 bits.
;   XmitAddr : transmits 7 bits (for address only).

XmitAddr:  mov   bitCnt,#8        ; set 8 bits of address count.
           sjmp  Xmbit2
XmitByte:  mov   bitCnt,#8        ; set 8 bits of data count.
Xmbit:     mov   I2DAT,A          ; send this bit.
Xmbit2:    rl    A                ; get next bit.
           jnb   ATN,$            ; wait for bit sent.
           jnb   DRDY,XMErr       ; should be data ready.
           djnz  bitCnt,Xmbit     ; repeat until all bits sent.
           mov   I2CON,#BCDR+BCXA ; switch to receive mode.
           jnb   ATN,$            ; wait for acknowledge bit.
           jnb   RDAT,XMBX        ; was there an ack?
           setb  NoAck            ; return no acknowledge status.
XMBX:      ret

XMErr:     setb  Fault            ; return bus fault status.
           ret


; Byte receive routines.
;   RDAck: receives a byte of data, then sends an acknowledge.
;   RcvByte : receives a byte of data.
;   data returned in ACC.

RDAck:     acall RcvByte          ; receive a data byte.
           mov   I2DAT,#0         ; send receive acknowledge.
           jnb   ATN,$            ; wait for acknowledge sent.
           jnb   DRDY,RdErr       ; check for bus fault.
           ret

RcvByte:   mov   bitCnt,#8        ; set bit count.
           clr   A                ; init received byte to 0.
Rbit:      orl   A,I2DAT          ; get bit, clear ATN.
           rl    A                ; shift data.
           jnb   ATN,$            ; wait for next bit.
           jnb   DRDY,RdErr       ; should be data ready.
           djnz  bitCnt,Rbit      ; repeat until 7 bits are in.
           mov   C,RDAT           ; get last bit, don't clear ATN.
           rlc   A                ; form full data byte.
           ret

RdErr:     setb  Fault            ; return bus fault status.
          ret
```

```
; I2C stop routine.

SendStop:
          clr   MASTRQ             ; release bus mastership.
          mov   I2CON,#BCDR+BXSTP  ; generate a bus stop.
          jnb   ATN,$              ; wait for atn.
          mov   I2CON,#BCDR        ; clear data ready.
          jnb   ATN,$              ; wait for stop sent.
          mov   I2CON,#BCARL+BCSTP+BCXA ; clear I2C bus.
          clr   TIRUN              ; stop timer I.
          ret


; I2C repeated start routine
;   Enter with address in ACC.

RepStart:
          mov   I2CON,#BCDR+BXSTR  ; send repeated start.
          jnb   ATN,$              ; wait for ATN.
          mov   I2CON,#BCDR        ; clear data ready.
          jnb   ATN,$              ; wait for repeated start sent.
          mov   I2CON,#BCARL+BCSTR ; clear start.
          ret


; Bus fault recovery routine.

Recover:
          acall FixBus             ; See if bus is dead or can be ‘fixed’.
          jc    BusReset           ; If not ‘fixed’, try extreme measures.
          setb  Retry              ; If bus OK, return to main routine.
          clr   Fault
          clr   NoAck
          setb  CLRTI
          setb  TIRUN              ; Enable timer I.
          setb  ETI                ; Turn on timer I interrupts.
          ret

; This routine tries a more extreme method of bus recovery.
;   This is used if SCL or SDA are stuck and cannot otherwise be freed.
;   (will return to the Recover routine when Timer I times out)

BusReset:
          clr   MASTRQ             ; Release bus.
          mov   I2CON,#0BCh        ; Clear all I2C flags.
          setb  TIRUN
          sjmp  $                  ; Wait for timer I timeout (this will
                                   ;   reset the I2C hardware).

; This routine attempts to regain control of the I2C bus after a bus fault.
;   Returns carry clear if successful, carry set if failed.

FixBus:   clr   MastRQ             ; Turn off I2C functions.
          setb  c
          setb  SCL                ; Insure I/O port is not locking I2C.
          setb  SDA
          jnb   SCL,FixBusEx       ; If SCL is low, bus cannot be ‘fixed’.
          jb    SDA,RStop          ; If SCL & SDA are high, force a stop.
          mov   BitCnt,#9          ; Set max # of tries to clear bus.
ChekLoop: clr   SCL                ; Force an I2C clock.
          acall SDelay
          jb    SDA,RStop          ; Did it work?
          setb  SCL
          acall SDelay
```

```
          djnz  BitCnt,ChekLoop    ; Repeat clocks until either SDA clears
                                   ;   or we run out of tries.
          sjmp  FixBusEx           ; Failed to fix bus by this method.

RStop:    clr   SDA                ; Try forcing a stop since SCL & SDA
          acall SDelay             ;   are both high.
          setb  SCL
          acall SDelay
          setb  SDA
          acall SDelay
          jnb   SCL,FixBusEx       ; Are SCL & SDA still high? If so,
          jnb   SDA,FixBusEx       ;   assume bus is now OK, and return
          clr   c                  ;   with carry cleared.
FixBusEx:
          ret

; Short delay routine (10 machine cycles).

SDelay:   nop
          nop
          nop
          nop
          nop
          nop
          nop
          ret

;*****************************************************************************
;                                 Main Program
;*****************************************************************************

Reset:
          setb  ETI                ; enable timer I interrupts.
          setb  EA                 ; enable global interrupts.

; These test cases are setup to be used with the I2C Demo board.

MainLoop:
          mov   SlvAdr,#KEYLED     ; set slave address (8-bit I/O port).
          mov   ByteCnt,#1         ; set up byte count.
          mov   SubAdr,#0h         ; set slave subaddress.
          acall RcvSub             ; read data from slave.
          jb    retry,MainLoop     ; repeat if there is anything wrong.

          mov   a,RcvDat           ; get received data byte.
          anl   a,#0fh             ; mask off the pushbuttons.
          swap  a                  ; mirror the pushbuttons to the LED bits.
          orl   a,#0fh             ; don't lock the pushbutton bits.
          mov   XmtDat,a           ; echo back this value.

ML2:      mov   SlvAdr,#KEYLED     ; set slave address (8-bit I/O port).
          mov   ByteCnt,#1         ; set up byte count.
          mov   SubAdr,#0h         ; set slave subaddress.
          acall SendSub            ; send data to slave.
          jb    retry,ML2          ; repeat if there is anything wrong.

          sjmp  MainLoop           ; repeat only the pushbutton/LED transaction.

          org   0fd00h             ; EPROM Configuration Byte (UCFG1)
          db    038h               ; WDT off, RST pin on, port RST high,
                                   ;   BO=2.5V, CLK / 1, osc = high freq.
          end
```

NOTES

Using the 87LPC76X microcontroller as an I^2C bus master

AN464

Definitions

Short-form specification — The data in a short-form specification is extracted from a full data sheet with the same type number and title. For detailed information see the relevant data sheet or data handbook.

Limiting values definition — Limiting values given are in accordance with the Absolute Maximum Rating System (IEC 134). Stress above one or more of the limiting values may cause permanent damage to the device. These are stress ratings only and operation of the device at these or at any other conditions above those given in the Characteristics sections of the specification is not implied. Exposure to limiting values for extended periods may affect device reliability.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Disclaimers

Life support — These products are not designed for use in life support appliances, devices or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

Date of release: 01-00

Document order number: 9397 750 06849

Let's make things better.

Philips Semiconductors

PHILIPS