APPLICATION NOTE

ABSTRACT

Extends the basic concepts presented in AN464 (*Using the 87LPC76X microcontroller as an I²C bus master*) with special focus on multi–master configurations including the arbitration mechanism and handshake by clock synchronization. A description of various software routines is followed by an ASM example illustrating their use in a multi-master configuration with bus fault detection and recovery.

AN465

Using the 87LPC76X in multi-master I²C applications

2000 Jan 12

Philips Semiconductors

PHILIPS

Using the 87LPC76X in multi-master I²C applications AN465

INTRODUCTION

The Philips Semiconductors 87LPC76X 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 Inter IC (I²C) bus developed 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.

Interfacing the devices in an I²C based system is very simple as 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 schematics, as there is no need to design bus interfaces. In addition, functional blocks on the block diagram correspond to actual ICs. A prototype system or a final product version can be easily 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. The I²C-bus compatible ICs give cost reduction benefits through 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 re-usable software modules. In addition, the multimaster capability allows rapid testing and alignment of end-products via external connections to an assembly-line computer.

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 facilitates I/O and RAM expansion, access to EEPROM, and processor-to-processor communications.

The multimaster capability of the I²C bus allows easy integration and expansion of relatively complex systems, in which different devices can independently initiate data transfers. Integration of a multimaster system is easy as a Master on the bus does not have to coordinate its data transfer with other potential Master devices—arbitration and synchronization are taken care of by the hardware and bus protocols. Expanding a system with a new device is trivial—it is “clipped” onto the two serial bus lines, and the new device may act as a Master without any modification to the other devices (see Figure 1). Microcontrollers like the 87LPC76X on the I²C bus are extremely powerful, as they can be programmed to be both Masters and Slaves in the same system. This way the microcontroller may initiate communication on the bus, and when requested, will respond to a data transfer request by another device.

In this Application Note we shall discuss the most important technical features of the I²C bus and describe the special I²C hardware interface of the 87LPC76X. We shall demonstrate with an example how the microcontroller can be programmed for a multimaster environment. The communications routines of the example are quite general, and can be ported to many applications—so we shall discuss in detail the software interface to these routines.

The description of the 87LPC76X I²C interface hardware and part of the general discussion of the I²C bus is similar to Application Note AN464 which dealt with the microcontroller in a single-master environment. Most of the added discussions relate to the multimaster aspects of the bus.

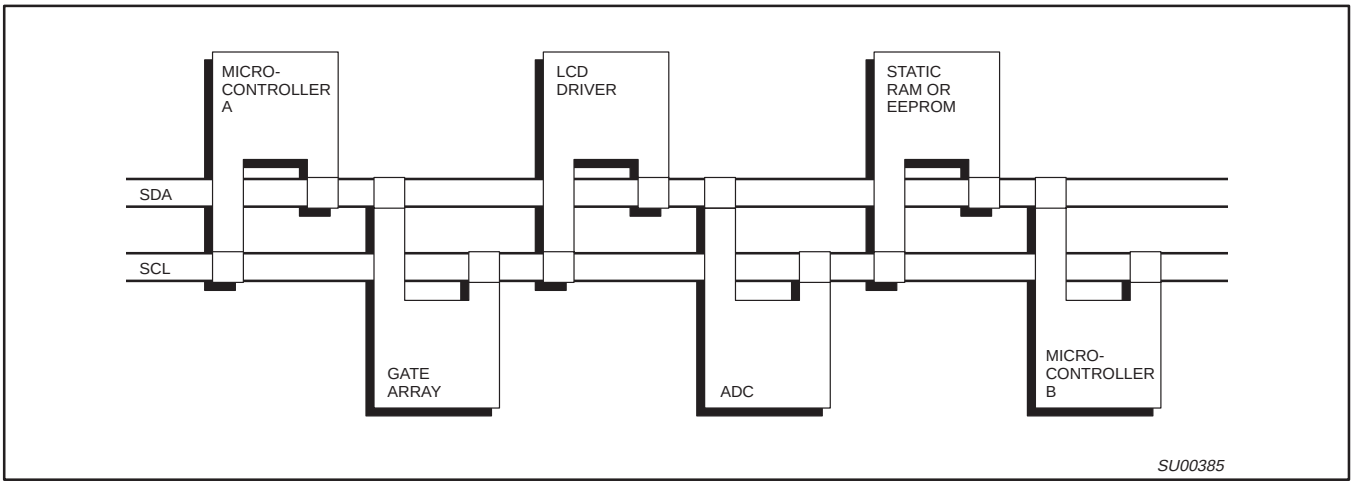

Figure 1. Example of an I²C-bus Configuration

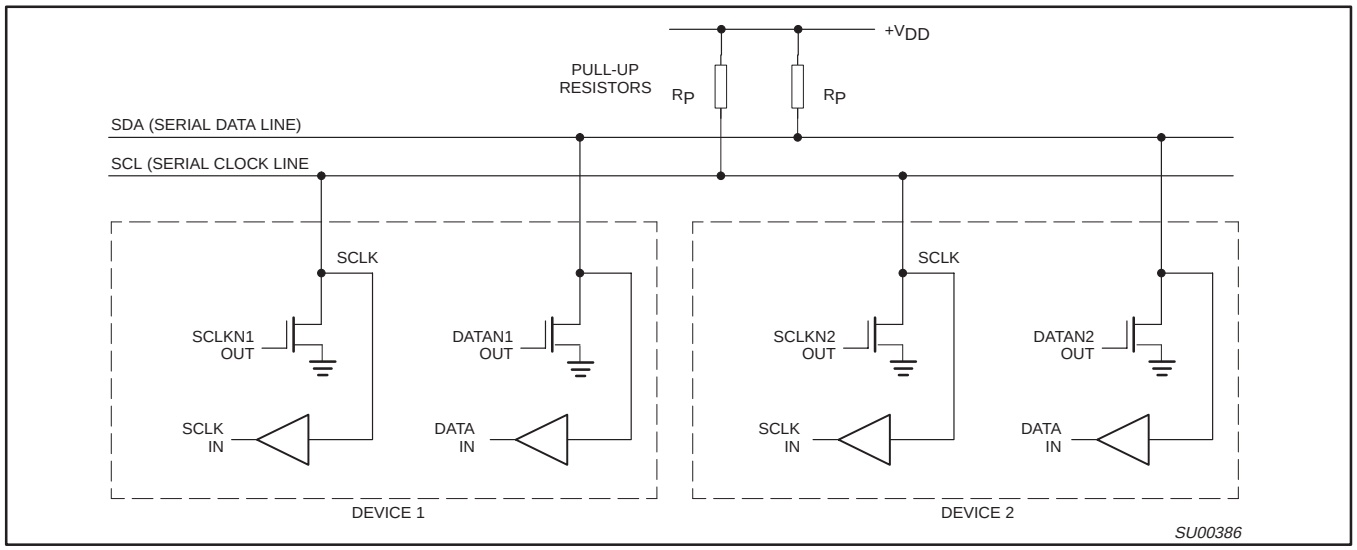

Figure 2. Connection of I²C-bus Devices to the I²C-bus

THE I²C BUS

The two lines of the I²C bus are a serial data line (SDA) and a serial clock line (SCL). A typical system configuration is shown in Figure 2. 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 a 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.

Every device connected to the bus must have an open-drain or an open-collector output for both the data (SDA) and the clock (SCL) lines. Each one of the lines is connected to the positive supply via a common pull-up resistor (see Figure 2). This implements a wired-AND function, and each of the bus lines which will have the HIGH level only if all the output transistors tied to it are switched off.

Data on the I²C bus can be transferred at a rate up to 100kbit/s. The number of devices connected to the bus is limited only by the maximum bus capacitance of 400pF. As different technology devices can be connected to the I²C bus, the levels of the logical 0 (Low) and logical 1 (High) are not fixed and depend on the appropriate level of V_{DD}.

MASTERS AND SLAVES

When a data transfer takes place on the bus, a device can be either 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 be either 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 being 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. 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 exist per transfer.

As more than one master may be connected to the bus it is possible that two devices will try to initiate 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.

DATA TRANSFERS

One data bit is transferred during each clock pulse (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 (Figure 4). The bus is considered to be busy after the Start condition and free again a certain time 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 (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 (Figure 6).

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.

An exception to the “acknowledge after every byte” rule 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 ACK”.

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

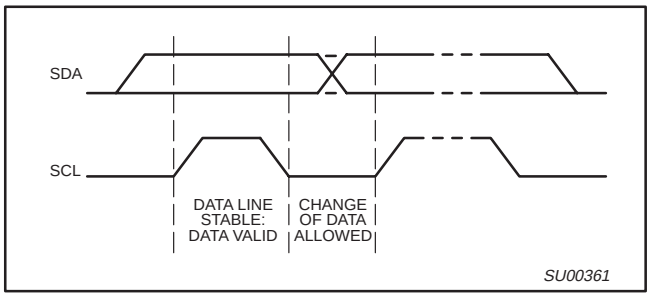

Figure 3. Bit Transfer on the I²C Bus

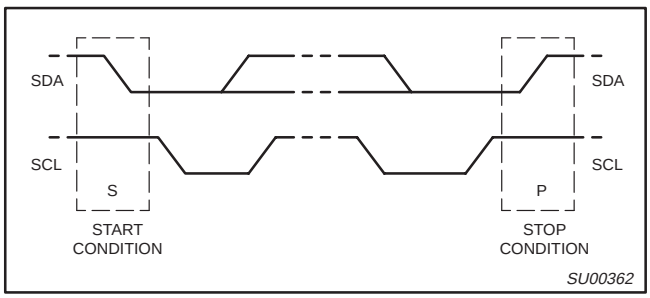

Figure 4. Start and Stop Conditions

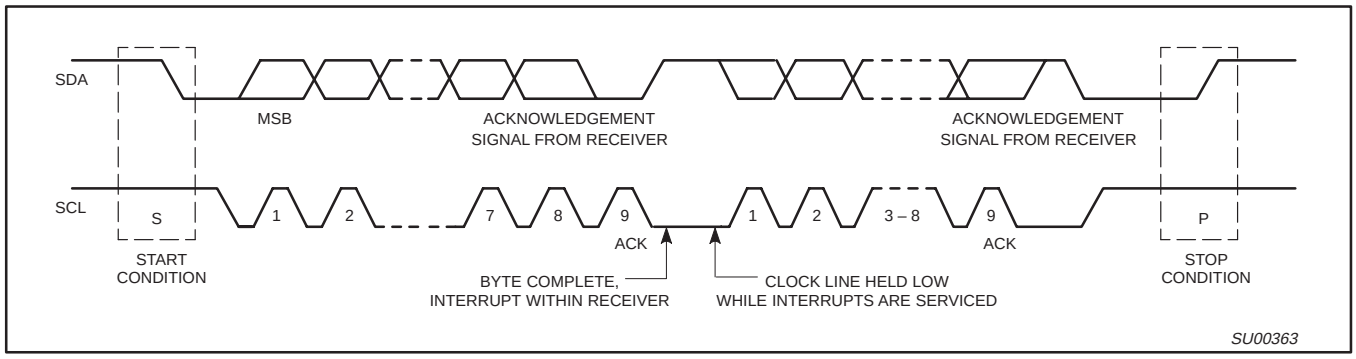

Figure 5. Data Transfer on the I²C Bus

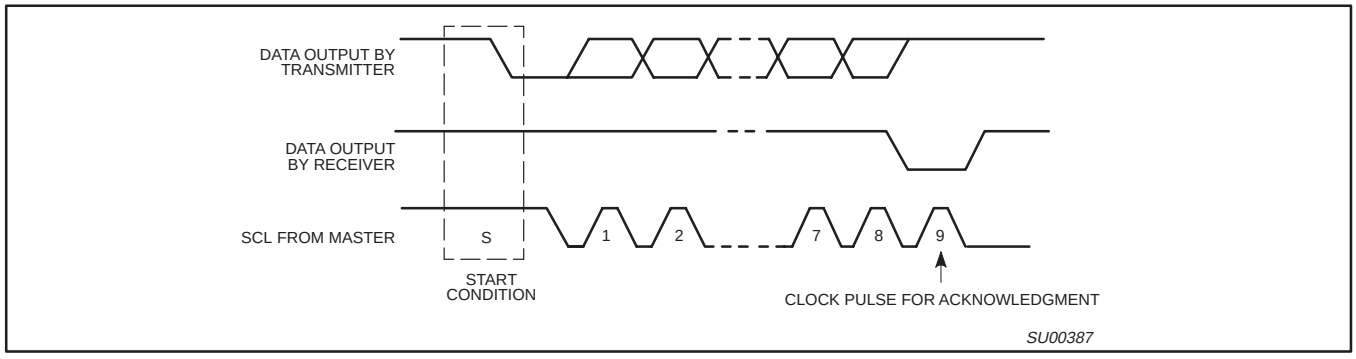

Figure 6. Acknowledge on the I²C Bus

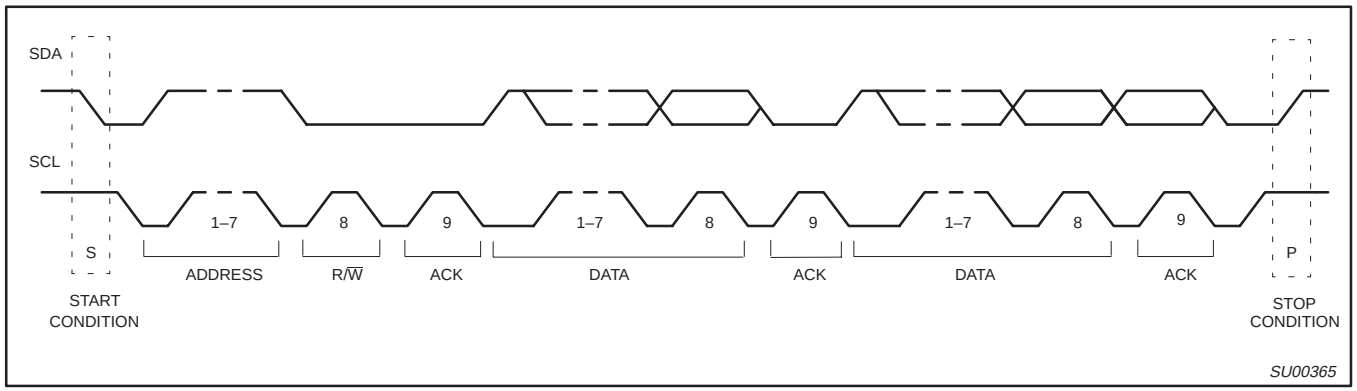

Figure 7. A Complete Data Transfer on the I²C-Bus

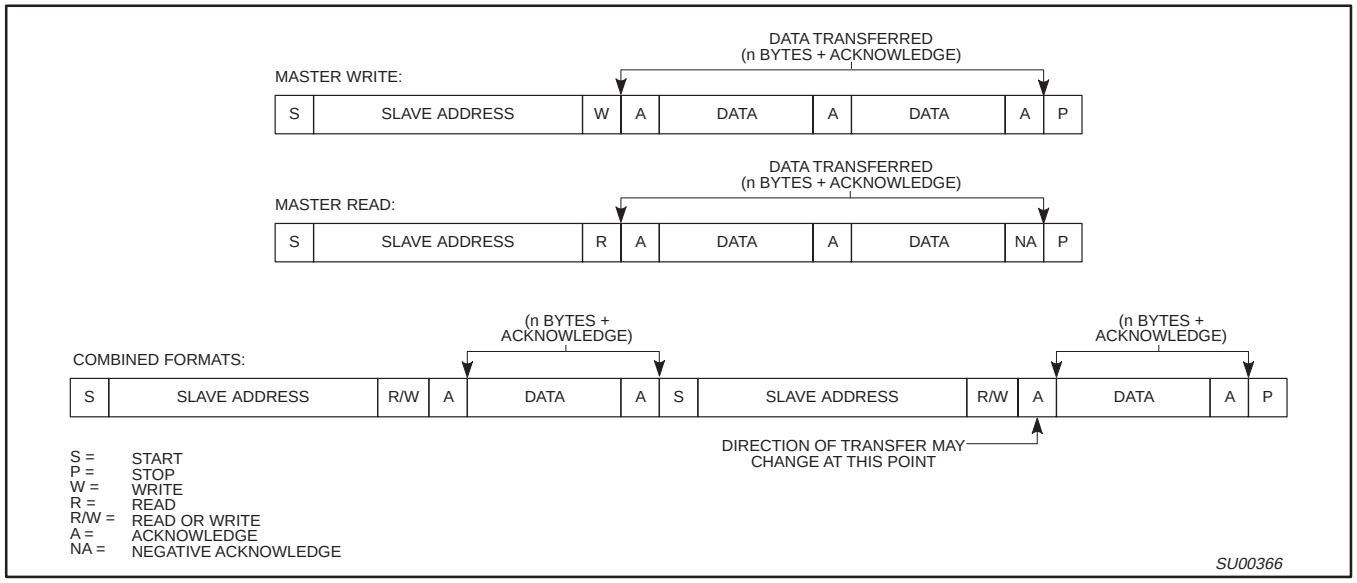

Figure 8. I²C Data Formats

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 of this transaction. A well-behaved slave, if it exists on the network, should of course acknowledge the master's addressing. The addressing is done with 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 8 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 start another transaction by 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 ACK), the master issues again a 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. Only after the transfer with the last slave took 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 is 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 prevents other devices from initiating transfers.

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 is addressing memories, when we want to access a specific word inside the device 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 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 eight 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. The data bytes which follow will be written one after the other starting with the sub-address location and the register is incremented automatically.

The memory read cycle (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 with each followed by a master-generated acknowledge. The last byte of the read cycle will be followed by a Negative ACK, signalling the end of transfer. The cycle is terminated by a Stop signal.

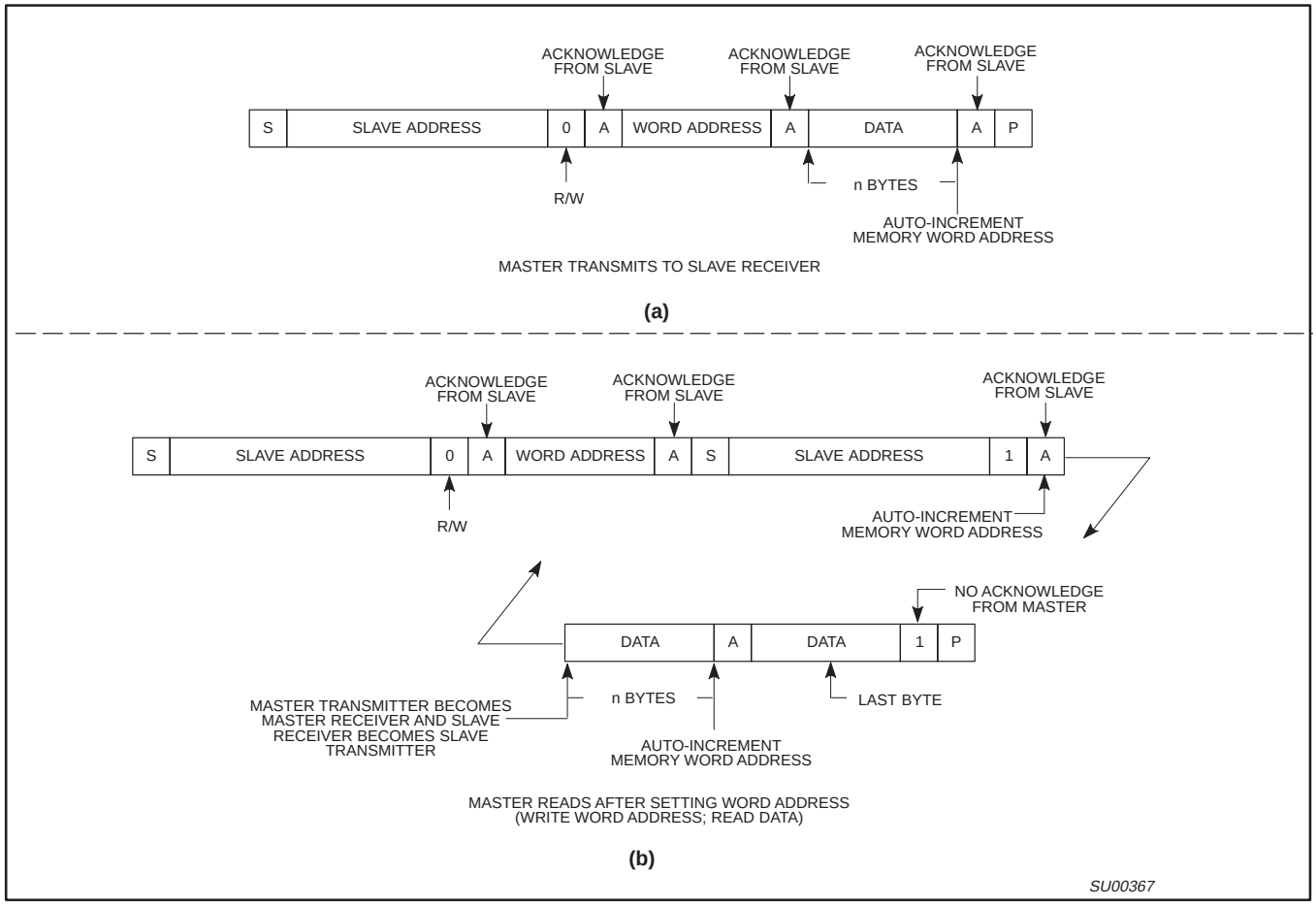

Figure 9. I²C Sub-Address Usage

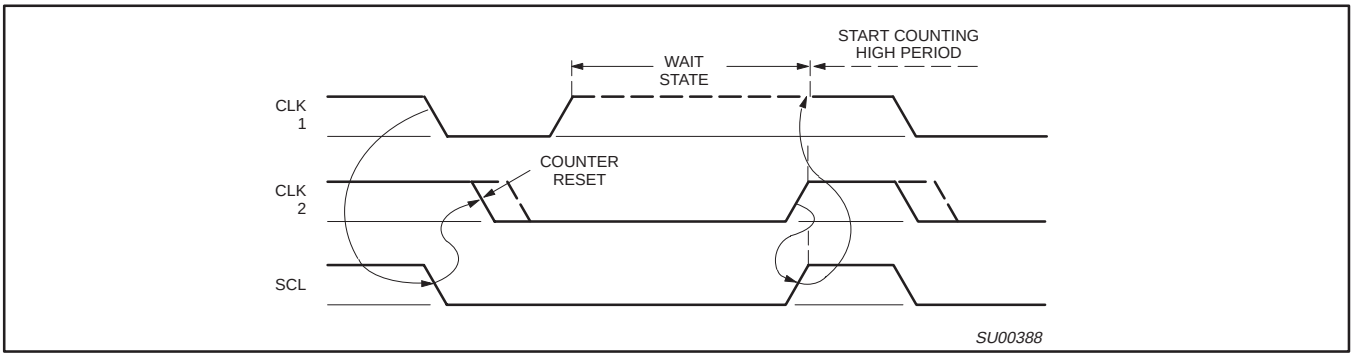

Figure 10. Clock Synchronization During the Arbitration Procedure

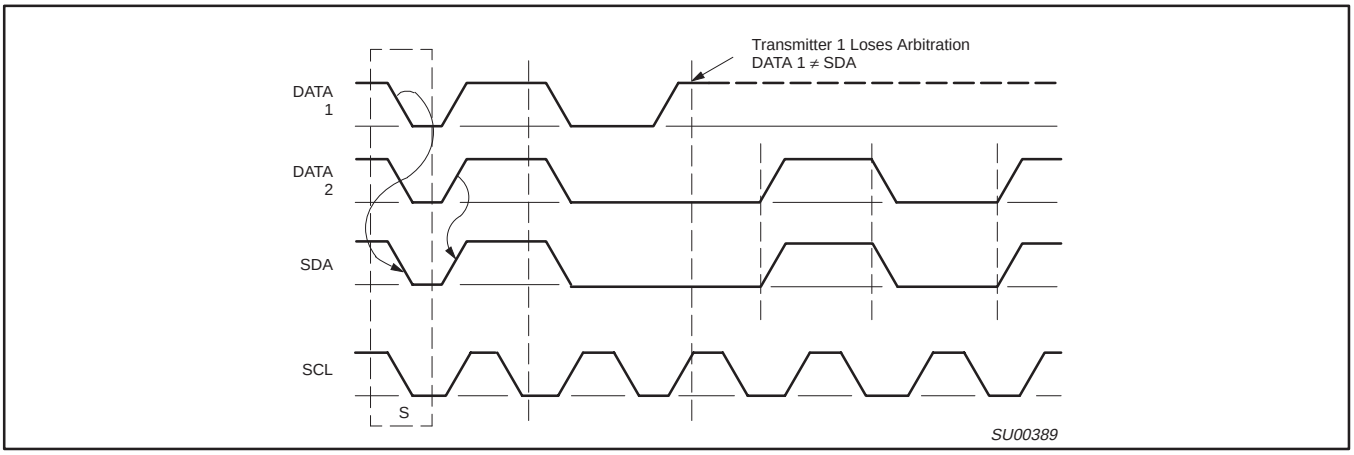

Figure 11. Arbitration Procedure of Two Masters

ARBITRATION IN A MULTIMASTER SYSTEM

The decision about which master has control over the I2C bus is based solely on the address and data sent by competing masters, and there is no central master or any order of device priority on the bus. Any device connected to the I2C bus is allowed to become a master, but devices are not supposed to “steal” the bus from other devices when a transfer is in process. If a device wishing to be a Master is aware that a transaction (initiated by another master) is taking place, it will wait until the transfer is concluded with a Stop condition on the bus—and only then try to seize it by sending its own Start. It is possible, however, that two or more masters may want to start a transfer at exactly the same moment. A scenario that may happen quite frequently in a loaded system: two devices are waiting for a long transaction to be completed, and simultaneously try to get the bus when detecting the Stop condition. An arbitration procedure synchronizes the different clocks, ensuring that the data is not corrupted, and causes all masters except one to withdraw from the bus, so only one master will control the transfer. This procedure applies only when masters initiate transfers simultaneously.

The clock synchronization, illustrated in Figure 10, ensures that only one defined clock is generated on the bus. It occurs naturally, as a result of the wired-AND property of the SCL line. Suppose two masters want to initiate a transfer on the bus. Clk1 and Clk2 in Figure 10 illustrate the desired clock outputs of each device, which would actually occur on the bus if each were the only master. The SCL waveform is the resulting wired-AND of the two clocks. The device that pulls the SCL down first will succeed. The other masters continuously monitor the clock line, and reset their internal clock counter to start counting their own Low clock period. This way, the first falling edge will synchronize all clock generators to the beginning of the Low time.

Once a device clock has gone Low it will hold the SCL line in this state until its internal clock High state is reached, and then will release the line. The Low to High change in this device will not change the state of the SCL line if another device, which is still within its Low period, is pulling down the line. This way, SCL will be held Low by the device with the longest Low period. A master that has finished its Low time earlier will enter a wait state until SCL is released by the slowest master and goes high. Upon the rising edge of SCL all masters start counting their High period, the first device to complete its High period will pull the SCL Low. In this way a single, synchronized clock is generated on the bus where the rising edge is being defined by the slowest master and the falling edge by the fastest master.

Arbitration between masters takes place on the SDA line. A master which tries to transmit a High while another device transmits a Low will withdraw, shutting off its data output stage and not interfering with the transfer until a Stop condition is detected. Due to the wired-AND property of the SDA line, a device “knows” that it lost arbitration by the fact that the Low SDA is different than its desired High output. Arbitration starts by comparing the address bits. When masters transmit different addresses the one transmitting the address with the lowest binary value wins. If all masters in arbitration transmit to the same address, arbitration continues into

the comparison of data. Figure 11 illustrates the arbitration process between two masters.

By definition, the transfer that forces the wired-AND result is the one that wins the arbitration, so the address and data of a winning device are not corrupted and no information is lost in the arbitration process. A master losing arbitration may generate clock pulses until the end of the byte. Thus it may affect the clock speed, but not the data on the bus.

If a master loses arbitration during the addressing stage it is possible that the winning master is trying to address it. In an efficient design, the losing master should switch immediately to its slave receiver mode, receive the data transmitted and acknowledge it—otherwise the message will have to be re-transmitted or is lost. A well designed master will take into account "illegal" protocol situations and will determine that it lost arbitration when it detects a Stop or a Start which are not synchronized with its own transmission. Electrical interference or a malfunctioning device may cause such a situation which actually corrupts the message transfer.

HANDSHAKE BY CLOCK SYNCHRONIZATION

The clock synchronization mechanism as described above actually implements a handshake mechanism, enabling receiving devices to "slow down" fast transfers when necessary.

On the bit level, a slow slave device like a microcontroller that does not have hardware I²C interface port, can extend each clock period and slow down the bus clock. The speed of any master is adapted to the operating rate of this device as long as it is active on the bus.

On the byte level the synchronization mechanism takes effect as a "handshake" mechanism when a slave device that was fast enough to receive or transmit a byte still needs extra time to store the received byte or prepare the next byte for transmission. The slave can hold the SCL line low after the reception and acknowledge of a byte, thus forcing the Master into a wait state—until the slave is ready for the next transfer.

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 error checks, and takes care of clock stretching and synchronization. The hardware support includes a bus timeout 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 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 033h. An I²C interrupt is usually requested (if enabled) when a rising edge of SCL indicates new data 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 073h.

The I²C port is controlled through four special function registers: I²C Control (I2CON), I²C Configuration (I2CFG), I²C Data (I2DAT) and I²C Status (I2STA).

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 rate timer.

For the bus monitoring function, Timer I is being 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 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 into the I²C caused all masters to withdraw from the I²C arbitration.

The time-out interval of Timer I is fixed: 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 we let Timer I run, but clear it whenever a frame is not in progress (reset or Stop occurred more recently than the last Start condition) or SCL changes within a frame. (Note: we wrote "about 1024 machine cycles" for the sake of accuracy—this number may slightly change according to the setting of the CT0 and CT1 bits mentioned below. In any case, the exact number of cycles for a time out does not have any practical significance).

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 this same 87LPC76X caused the bus hang-up due to a lack of software response. SCL will be released and I²C operation between other devices could continue.

I2CON Register

The I²C Control register can be read or written to (see Figure 12).

When writing to the I2CON register one should use bit masks as demonstrated in the examples. 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 CLRB reads the register, sets the bit and writes it back—and the write-back may affect other bits.

I2CFG Register

The configuration register is a read/write register (see Figure 13).

I2DAT Register

The I²C data register 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 14).

I2CSTA Register

The I²C STAtus Register is a read-only register reflecting the internal status of the I²C interface hardware (see Figure 15).

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 ARL. 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 12. 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 13. I2CFG Register

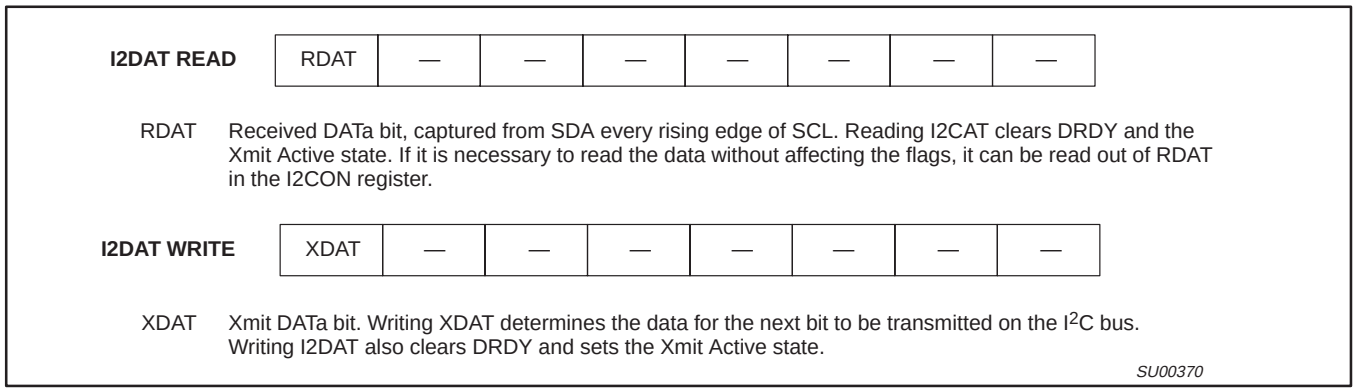

Figure 14. I2DAT Register

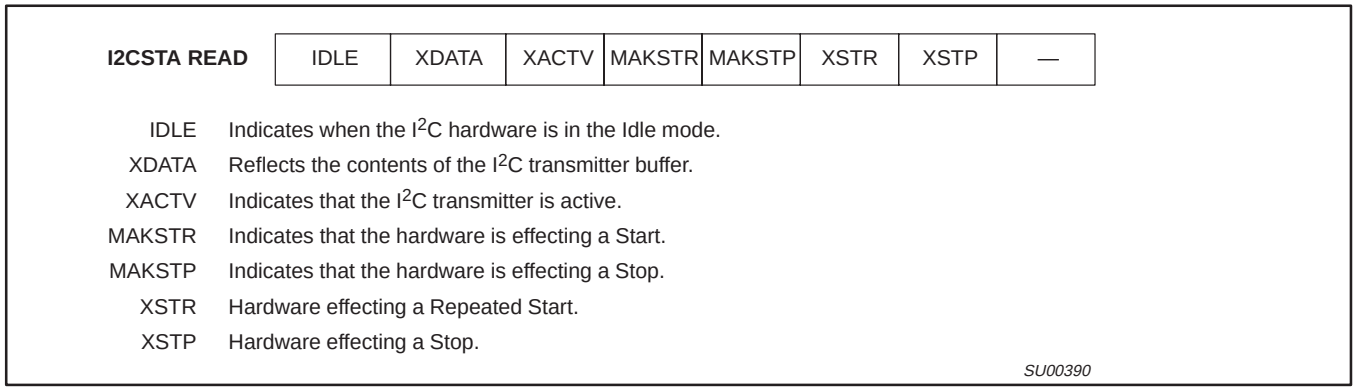

Figure 15. I2CSTA Register

I²C COMMUNICATIONS SOFTWARE

The software listing demonstrates programming the 87LPC76X for a multimaster I²C environment where the device can be both a Master or a Slave responding to other Masters on the I²C network. The bulk of the software is communications routines which are not only for demonstration but could be ported to other user programs with minimal or no modifications. The routines are quite general and could be useful in most applications. We have tried to design a well-defined software interface, enabling most users to copy the routines as they are, modifying only the pre-defined interface elements to fit the specific applications. We encourage users to use the routines without modifications whenever possible, as the lower levels of the hardware-software integration could be quite involved.

The rest of this application note will relate to the programming example. We shall discuss the general operation of the routines and how they are integrated into an application. Then we shall describe in detail all the software interface elements and how to use them.

I²C COMMUNICATIONS ROUTINES—OVERVIEW

In order to function well in a multimaster environment the microcontroller must be able to take control of the I²C bus as a Master, “tolerate” message transactions between other Masters and other devices, and respond efficiently as a Slave to other bus Masters. The communications routines should allow a Master “graceful” recovery from an arbitration loss and other situations when a message transaction is not completed, allowing for communication re-tries.

For Slave operation the microcontroller must be interrupt driven relative to an I²C frame Start, as any Master on the bus could request a transaction at any moment, not synchronized to the application program executing on the controller. An interrupt service routine monitors the address transmitted on the bus. When the microcontroller is addressed it takes care to either read the data from the bus into a buffer or write buffer data onto the bus. When such a transaction is successfully completed, one of several “Slave Event Routines” is called prior to returning to the main application program. Such an “Event Routine” is a part of the application, allowing an immediate response to the data received, or the fact that data was transmitted to a requesting Master. This allows “synchronization” of the application to a “slave” bus transaction. Typical uses of the Event Routine mechanism will be a computation based on new data, or re-loading the transmit buffer with new data getting ready for the next random request. The actual Event Routines will be programmed differently for different applications, but the names and the calls will remain the same as long as the communications routines are left unmodified.

A transaction as a Master is initiated by the application program. Our implementation uses the interrupt mechanism for the Master communications as well. The application issues a request for the bus by setting the MASTRQ bit of the I²C port control, and when the bus is available an interrupt occurs. This way, if the bus is free there will be an immediate response. If the bus is busy, the application may go on executing (if so programmed) until this controller can get control of the bus. When the microcontroller gets mastership of the bus it initiates a bus transaction according to “directives” set by the

application program. The most important directives are the address (and subaddress if relevant) of the slave device addressed, and the length of the message to be transmitted or received.

When a Master transaction is concluded, a Master Event Routine (called MastNext) is called to perform whatever task the application demands. As with the Slave Event Routines it will typically respond to a successful transmission or reception of data. In addition, it could handle situations where a slave does not respond at all, or does not acknowledge a data byte (thus causing data transfer to terminate). A program might react to the fact that a slave does not respond by re-trying to communicate at a later time, by issuing a message to another peripheral device or just ignoring it. The handling of such cases is application dependent, and should be programmed into the routine called “MastNext”. The MastNext routine is invoked when the Master terminates the transaction “willingly”, but not upon arbitration loss.

The microcontroller operating as a bus Master may lose arbitration to another Master which happens when two Masters transmit in synchronization, commencing with the same Start signal. If arbitration is lost while transmitting or receiving data, our processor withdraws from the bus and turns itself into a slave—an active Slave upon a Start, or returning to the calling program as an idle slave. When the arbitration loss occurs while transmitting an address, our processor turns itself immediately into an active slave, “listening” to the rest of the address transmitted by the new Master. If our processor reads its own address from the bus (as transmitted by the new Master) our processor responds as a willful slave. If this mechanism would not have been implemented, there could be potential inefficiency when a device that happened to be synchronized to another Master loses arbitration, but is not able to respond to the winning device.

Another situation for arbitration loss could be a bus exception resulting from a device operating not according to the bus protocol or interference on the bus lines. In addition to “regular” arbitration loss detected with the ARL hardware flag, such a situation may occur with detecting a Start or a Stop in the middle of transmitting an address or data byte. In such a situation the microcontroller withdraws from the bus as well—active Slave upon a Start detection, or returning as an idle slave in other cases.

When a Master transaction is terminated by an arbitration loss, the Master Request flag (MASTRQ) of the hardware I2C port remains in effect. As a result when the bus gets free, our device will take control, issue a Start, and the transaction that was cut will start again. This restart will happen automatically, without any application involvement (unlike non-acknowledgement, where the MastNext routine determines what shall be done).

The I2C communications routines are structured as an interrupt service routine responding to an I2C port interrupt upon a frame Start. Within a frame the I2C processing is continuous, where the I2C port is polled for hardware response, and the I2C interrupts are disabled. Other interrupts are enabled during the service routine. The set-up requirements from the mainline program are minimal, and interfacing is done via RAM buffers and some pre defined RAM locations. The lower level interface with the hardware is done inside the service routine, and can typically be ignored by the application programmer.

BUS WATCHDOG AND ERROR RECOVERY

A malfunctioning device (in hardware or software) may hold the SCL line low, thus causing the bus to be “stuck”. It might even be possible that a transient protocol violation (due to hardware interference, such as a device turning on) may cause some devices (non programmable, or even microcontrollers which were not carefully programmed) to hold the bus. Since within a frame the bus is software-polled, a “stuck” bus might cause the application software to “hang forever”. Here the TIMERI watchdog comes to the rescue, interrupting when there is no bus activity for a long period of time.

When the I2C service routine is interrupted by the watchdog timer, the processing of the current frame is not completed and the event routines are not called. The software returns to execute the mainline application, and will be interrupted again for the next frame (next Start, received as a slave or induced as a Master). A status flag and a counter report on the watchdog interrupt, so the application program can be made to inhibit the I2C port if there are too many occurrences of a “hanging” bus.

Bus protocol errors and “hangups” might be an issue in systems which are susceptible to noise, temporary bus line shorts, “hot plug in” of devices or even erroneously programmed devices—and a “fail safe” controller program should be able to detect bus problems and possibly assist in resolving them. The RECOVER routine resets the I2C interface of the microcontroller, and attempts to release some other devices on the bus by toggling the clock line. The I2C interface of the 87LPC76X is reset by letting TimerI run and expire, since this circuitry does not feature a software controlled reset. This “extreme” measure is needed in some cases of bus protocol violation.

The bus and interface circuit recovery routine can be automatically invoked whenever TimerI detects a timeout. In addition, for systems where potential bus failures are a concern and reliability is an issue, one may implement mechanisms to invoke bus and interface recovery from the application code. This may help in cases where the bus gets “stuck” when there is no I2C frame in progress. In such an instance the watchdog timer will not give any timeout indications, as it has not been activated. Another case emanates from a design peculiarity of the interface circuitry on the 87LPC76X: if the SCL line is externally grounded when there is a Start condition, this Start might be ignored, and the watchdog may not be activated. Our programming example deals with potential failures by testing for transaction completion and retrying transmissions when necessary (these are explicit retries, in addition to an “automatic” retry after a Master’s arbitration loss, invoked by the MASTRQ bit). Too many transmission failures activate the RECOVER routine.

I2C COMMUNICATIONS ROUTINES—INTERFACE

The I2C service routine deals with the transmission and reception of messages, without any concern for the contents of the message. In order to provide a general interface for different applications the data is transferred via buffers. The service routine does not have to "know" where the data goes to or comes from—as long as the application program specifies the required pointers for these buffers. The interface to the actual application (which "cares" about message contents, timing, addressing and so forth) is done in a well defined manner, allowing usage of the same service routine with different application programs.

The interface is carried out with the use of buffers, pre-defined names for Application Event Routines, interface RAM locations for transferring parameters, pointers and flags, and constants. A more detailed discussion of the interface follows.

Buffers

There are three buffers for data transfers between the I2C bus and the application program.

MasBuf is used for Master transmission and reception. The number of data bytes for each Master message—reception or transmission, is specified by the memory location MASTCNT. The value in MASTCNT should be less than the length of MasBuf. For Master transmission the message is placed in MasBuf before the transmission is initiated. In Master reception, the received message will be contained in the same buffer. There is only one Master message transaction occurring at the same time, so we may use the same buffer both for transmission and reception.

For Slave operation we must accommodate data transfers which may come randomly, asynchronous to each other or to possible operation of the same device as a Master. Therefore it is necessary to allocate additional RAM area as buffers dedicated to Slave operation: SRcvBuf for receiving data, STxBuf for transmission.

The length of the Slave receive buffer is defined by the symbol RBufLen. It is used by the code for protection, avoiding overwriting RAM beyond the allocated buffer size in case a Master sends a message which is too long. There is no need for RAM protection for transmission, but the Master should not request more data than STxBuf can supply.

Interface RAM Locations

RAM location MyAddr contains the address of this processor.

Status flag MSGSTAT is used for reporting to the application on I2C communications status—mainly on the successful, or unsuccessful, completion of a message transaction. The contents of MSGSTAT may be used by the mainline application code or by the Event Routines. The different codes that could be placed by the I2C service routine are described later in the text. When the message processing commences, a code indicating Slave or Master processing is inserted to MSGSTAT, and is updated as we go along.

There could be many applications that will not need to use MSGSTAT contents, as the very fact of calling a certain event routine implies completion of a processing stage.

For Master transactions, in addition to the data buffer MasBuf, there are several RAM locations into which the application inserts Master message "directives". These directives provide the service routine with the information necessary to carry out the next Master transaction. The one byte RAM locations used for directives are DESTADRW, DESSUBAD, MASTCNT and MASCMD.

DESTADRW contains the destination slave address in bits 7-1, while bit 0 is the R/W bit. Bit 0 contains 0 for a Write operation (the message is to be transmitted to the salve) and 1 for a Read operation (message is being read from the slave and received by this Master).

DESSUBAD contains the 8 bit sub-address of the slave, if necessary. For transactions without a sub-address, the contents of DESSUBAD is ignored.

MASTCNT contains the number of data bytes in the message to be sent from or received into MasBuf. This number should not be bigger than the length of MasBuf.

MASCMD byte contains the bit flags SUBADD, RPSTRT and SETMRQ. SUBADD is 0 (cleared) for a message with a regular address, and 1 (set) when a subaddress is required. When SUBADD is set, the service routine takes care of all the protocol required for sub-addressing, which includes a Repeated Start for Read operations. A message with a subaddress is considered to be a single message, even if it includes a Repeated Start.

The RPSTRT and SETMRQ are kept cleared in regular applications, and will be used only for "tailoring" the bus transfers in special cases. When RPSTRT is cleared the message will terminate, as usually required, with a Stop. When RPSTRT is set a Repeated Start will be sent on the bus, and Master operation will resume. The RPSTRT directive relates to terminating the message after all the data was transferred, and not to the mandatory Repeated Start in the middle of sub-addressed Read operation. A single message with a subaddress will typically have RPSTRT cleared. SETMRQ indicates what will be loaded into the MASTRQ flag of the hardware when Stop is transmitted. Typically it will be cleared. When SETMRQ is 1, MASTRQ will be set, thus trying to issue a new Start immediately following the Stop. In such a case the service routine will not return upon Stop, but will continue as a Master.

TITOCNT is used to count time-outs of the watchdog timer. Whenever such a timeout invokes the TIMER I interrupt service routine the contents of the location TITOCNT are incremented, and the timeout is reported in MSGSTAT. The count is saturated at 0FFh. This mechanism may be used in an application that is very much "concerned" with potential bus failures, allowing some type of "failure monitoring" by the application even for Slave transactions.

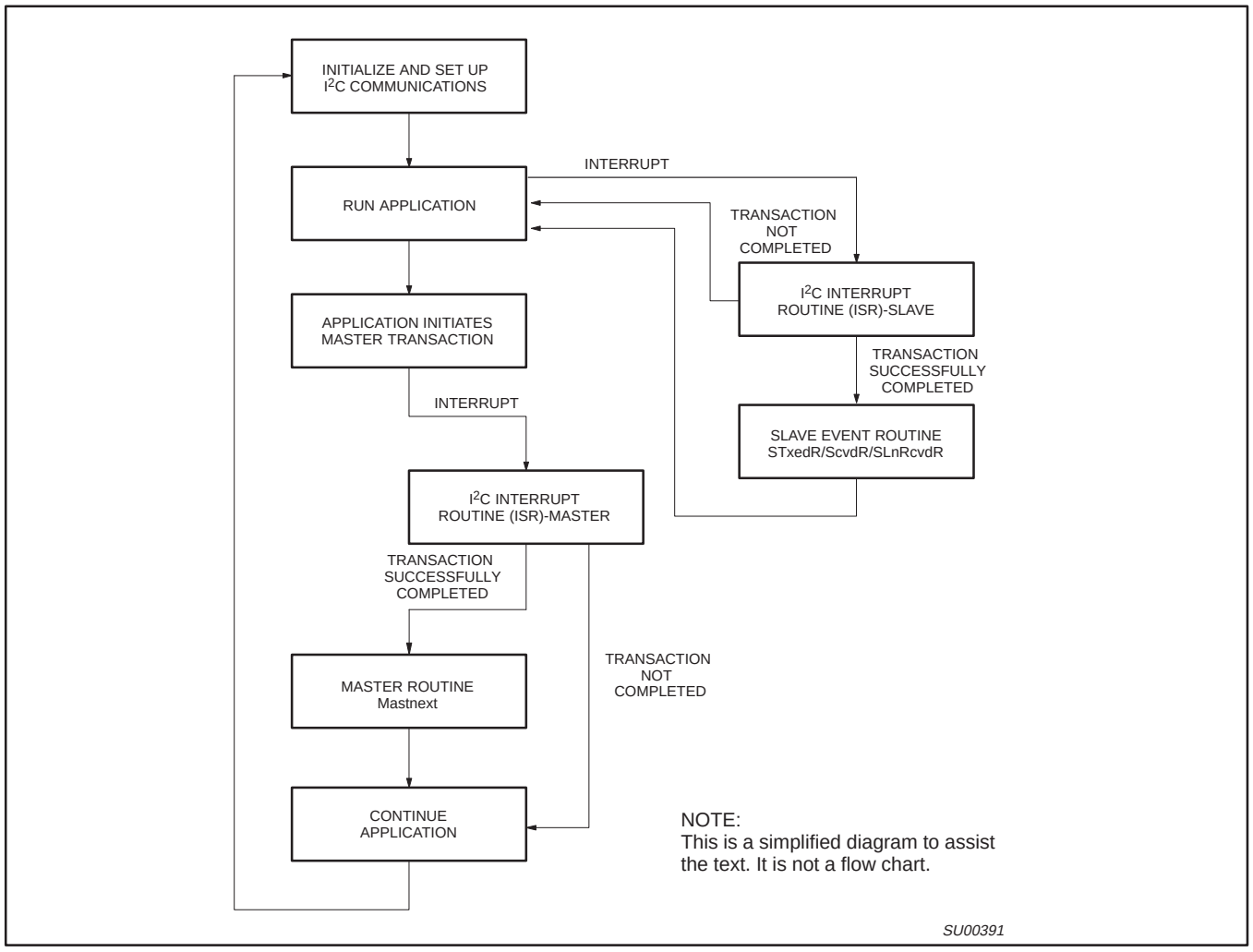

Figure 16. Typical Communications Scenario—A Simplified Diagram

APPLICATION EVENT ROUTINES

The service routine calls Event Routines with pre-defined names (Figure 16), and these routines must be provided by the application program. The actual code of the routines will differ from application to application, but the routine names are being kept the same.

These routines are being called when successful processing of a message (send or receive) is completed. The routines may perform whatever action the application was designed for, which is not necessarily related to the I²C communications mechanism. In addition, the routines may perform the data interface tasks for the I²C port, like emptying buffers from received data or preparing the next message by setting up the buffers.

The mechanism of calling the event routines out of the service routine allows an immediate reaction to the event of message processing completion, before any new activity happens on the bus. In some simple applications this may not be necessary. For example, one may have a main program for a slave which is just a wait loop monitoring a flag set by the service routine when a message transfer, initiated by some master, is completed. In such a case the application could react to the message completion after the interrupt service routine returns. However, in the general case this will not be sufficient. An example could be a slave with an application which is constantly busy doing another task, in an environment where the communication requests on the I²C bus are frequent. If there is a new message request shortly after the current message is completed, having to wait for the application until it “has time” may result in not reacting, or sending the same data again, or overwriting the received data in the buffer. Another obvious case demanding event routine calls is a Master sending different messages with a Repeated Start—the new data for the following message must be prepared in the interrupt service routine as the current message is completed (there is no return from interrupt prior to the new data transmission).

The programmer has the flexibility to decide where to prepare the next message according to the requirements of the application. This can be done after return from the event routine, in the application code after the return from interrupt, or a combination of both, where the time critical events are performed in the event routines. The application may monitor the MSGSTAT flag for message processing completion. If the event routines are not used, it is recommended to simply code them as a “RET” instruction, thus turning them into dummy routines (this an easier and better practice than changing the service routine itself, eliminating the calls).

Master Event Routine:

MastNext
This routine is called by the service routine when the processing of the current Master message is completed. For an indication on the type of message processing completion, MastNext may inspect the contents of MSGSTAT RAM location.

When MastNext is called, MSGSTAT will contain one of the following codes for message processing completion:

MRCVED (= 21h)—a complete message (with number of data bytes indicated by MASTCNT) was received from the slave.

MTXED (= 22h)—the number of data bytes indicated by MASTCNT were successfully sent and acknowledged by the slave.

MTXNAK (= 23h)—the slave did not acknowledge a data byte of the message, even though it had acknowledged its address. The message transmission was terminated upon the NAK.

MTXNOSLV (= 24h)—no slave acknowledged the address indicated by memory location DESTADDR.

The MastNext routine may perform any task(s) necessary for the application. Data handling tasks will typically be dependent on the MSGSTAT indication. One possible task could be setting the directives for the next message. The necessity for executing this task here (versus the main-line code initiating the transfer) is of course application dependent.

Slave Event Routines:

These routines are called when a message transaction as a slave has been completed. In many cases it could be important to utilize the calls to such routines as the requests for message transactions as a slave can come randomly, asynchronous to the application program. The application may demand that new data coming in should immediately initiate some tasks (e.g. control an output port)—and the event routine can be used to process the result of the slave interrupt.

In most cases it will be necessary for a slave to react immediately to a message received simply in order not to lose the data. As a new message may come randomly, it may overwrite the reception buffer before the data has been transferred out of it or acted upon.

For applications in which the reaction for slave events is performed after the return from the service routine, the event is reported by placing an appropriate code in the MSGSTAT flag. The programmer may use event routines, other mainline routines inspecting MSGSTAT, or both. If the event routines are not used, it is recommended to code them as a “RET” instruction.

SRcvdR:
Called by the service routine when a new, complete message has been received into SRcvBuf. When SRcvdR is called, R1 points to the address of the last byte received into the buffer. In a typical application SRcvdR will transfer the new data out of SRcvBuf, so it will not be written over by a subsequent slave reception.

The equivalent MSGSTAT indication for this event is SRCVD (= 11h).

SLnRcvdR:
Called when a slave message has been received into SRcvBuf, but the message was longer than the SRcvBuf buffer (as specified by RbufLen).

The equivalent MSGSTAT indication for this event is SRLNG (= 12h).

If the program is supposed to react to a too long a message the same way as to a message that can be contained in the buffer, one may code SLnRcvdR simply as a call to SRcvdR.

STXedR:
Called by the service routine when data has been transmitted out of the slave STxBuf buffer according to a master’s request. This routine may insert new data into the buffer, preparing it for the next slave transmission.

The equivalent MSGSTAT indication for this event is STXED (= 13h).

Note that we do not have a separate routine for the case that the master requested too many bytes—more than STxBuf length—and we sent out meaningless bytes. It is the master’s responsibility to specify the message length, and it should be able to request messages with the appropriate length from each slave on the bus.

SRErrR:
This routine relates more to bus communications than to the application itself. It can be called when we positively detect a bus error upon reception as a slave, in case the application is supposed to know about it. In most cases this call will not be used, as dealing with bus communications difficulties is usually left to the Master.

Just prior to calling SRErrR, the code SRERR (= 14h) is placed in MSGSTAT.

0Completion Routine:
I2CDONE
This routine is called every time, before returning from the I2C interrupt service routine, whether the transaction was successful or not. It can be used to “safely” monitor MSGSTAT without any risk of a new interrupt modifying the current indication. Simple application programs will not make use of this routine. A more sophisticated application implementing a fail-safe communications protocol may use it to count errors of a certain type in order to determine a recovery scheme. In our programming example, I2CDONE inhibits I2C interrupts when it is evident that as a result of protocol errors interrupts are not caused by legitimate Starts.

CONSTANTS

RBufLen—the length of SRcvBuf, the slave receive buffer. This constant may be used both by the I2C routines and the application program, and it is the responsibility of the application programmer to define the correct buffer length.

MYNUM—This ROM constant is dependent on the application environment. It is a small integer defining a “serial number” of the node, out of all the processors running the same code. This constant is used only when recovering from a timeout, in order to “de-synchronize” masters from each other when trying to recover the bus.

CTVAL1 is a constant defined in ROM. It is used by the application code portion which initializes the I2C, for loading CT0 and CT1 with a value appropriate for the crystal being used.

MYADDR1 is a ROM constant containing the address of the processor’s I2C node. This value is used by the application demo to load the RAM location MyAddr.

USING THE COMMUNICATIONS SUBROUTINES

In order to use the I2C Communications Routines an application program should take care of the following:

– Upon initialization, load bits CT1, CT0 of I2CFG register according to the clock crystal used (refer to the table of CT1, CT0 values in the 87LPC76X section of the Philips Semiconductors Microcontroller Data Handbook (IC28)).
– Load MyAddr RAM location with the address of this node.
– For Slave operation, load STxBuf with the initial data to be transmitted.
– For slave operation, set the SLAVEN bit in the I2CFG register.
– Enable I2C and watchdog interrupts by setting the ETI, EI2 and EA bits of the interrupt enable register.
– For Master operation, set up the next transaction by loading the appropriate directives into MASCMD, DESTADRW, DESSUBAD (if applicable) and MASTCNT, and load MasBuf with the appropriate data if it is a Write message.
– For Master operation, initiate the next transaction by setting MASTRQ bit in I2CFG.
– For both Master and Slave operation, handle data transmission and reception via the buffers in main-line code or the Event Routines.

PROGRAMMING EXAMPLE

The assembler listing includes the I2C Communications Routines and a demo application exercising these routines. In most real-life applications the code of the routines could be used without modifications. For those who follow the coding of the routines, one should note that in many instances code speed and program space have been slightly compromised in order to improve readability. The almost “general purpose” interface to the routines affects efficiency as well, and it is possible to write more compact and somewhat faster code for specific applications. The reader is encouraged, though, to use the code “as is” whenever possible.

The “application” demo is simple—two microcontrollers exchange messages in a “ping-pong” game. In addition to trivial message exchange, the code demonstrates recovery mechanisms from communications errors and bus “hangups”. We tried this code with two pairs of controllers exchanging messages on the same bus. The message exchange could repeatedly recover and restart when the SCL and SDA lines were temporarily shorted to ground or between themselves. Simpler versions, without the “protection” mechanisms, could “hang up” under such conditions.

Source Code Availability

The source code file is available from the Philips Semiconductors website: www.semiconductors.philips.com

```
;*************************************************************************
;         Multi–Master Test Code for 8xC751 from AN430
;         Modified for use with the 87LPC764, 15 June 1999
;*************************************************************************

; 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).

$title(I2C – Multi–Master Demo for 87LPC764)
$pagewidth(132)
$debug
$object
$mod764


;*************************************************************************
;       Symbols and RAM definitions
;*************************************************************************

; Symbols (masks) for I2CFG bits.

BTIR       EQU     10h                ; TIRUN bit.
BMRQ       EQU     40h                ; MASTRQ bit.

; Symbols (masks) for I2CON bits.

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

; Note:
;
; Specific bits of the I2CON register are set by writing into this register a
; combination of the masks defined above using the MOV command.
; The SETB command should not be used with I2CON, as it is implemented by
; reading the contents of the register, setting the appropriate bit and
; writing it back into the register. As the functionality of the Read and
; Write portions of the I2CON register is different, using SETB may cause
; unwanted results.

; Message transaction status indicated in MSGSTAT:

SGO        EQU     10h            ; Started Slave message processing.
SRCVD      EQU     11h            ; as a slave, received a new message
SRLNG      EQU     12h            ; received as slave a message which is too
                                  ;   long for the buffer
STXED      EQU     13h            ; as slave, completed message transmission.
SRERR      EQU     14h            ; bus error detected when operating as a slave.

MGO        EQU     20h            ; Started Master message processing.
MRCVED     EQU     21h            ; As Master, received complete message from
                                  ;   slave.
MTXED      EQU     22h            ; As Master, completed successful message
                                  ;   transmission (slave acknowledged all data
                                  ;   bytes).
```

```
MTXNAK     EQU    23h             ; As Master, truncated message since slave did
                                  ;   not acknowledge a data byte.
MTXNOSLV   EQU    24h             ; AS Master, did not receive an acknowledgement
                                  ;   for the specified slave address.

TIMOUT     EQU    30h             ; TIMERI Timed out.
NOTSTR     EQU    32h             ; Master did not recognize Start.

; RAM locations used by I2C interrupt service routines.


MASCMD     DATA   20h
SUBADD     BIT    MASCMD.0
RPSTRT     BIT    MASCMD.1
SETMRQ     BIT    MASCMD.2

           DSEG   AT 24h

MSGSTAT:   DS     1               ; I2C communications status.
MYADDR:    DS     1               ; Address of this I2C node.
DESTADRW:  DS     1               ; Destination address + R/W  (for Master).
DESSUBAD:  DS     1               ; Destination subaddress.
MASTCNT:   DS     1               ; Number of data bytes in message (Master,
                                  ;   send or receive).

TITOCNT:   DS     1               ; Timer I bus watchdog timeouts counter.
StackSave: DS     1               ; SP save location (used when returning from
                                  ; bus recovery routine).

MasBuf:    DS     4               ; Master receive/transmit buffer, 8 bytes.
SRcvBuf:   DS     4               ; Slave receive buffer, 8 bytes.
STxBuf:    DS     4               ; Slave transmit buffer, 8 bytes.


RBufLen    EQU    4h              ; The length of SRcvBuf

;*************************************************************************
;          APPLICATION output pins and RAM definitions
;*************************************************************************

; Outputs used by the application:

TogLED     BIT    P0.0            ; Toggling output pin, to confirm
                                  ;   that the ping–pong game proceeds fine.
ErrLED     BIT    P0.1            ; Error indication.
OnLED      BIT    P0.3            ;

; Application RAM

APPFLAGS   DATA   21h
TRQFLAG    BIT    APPFLAGS.0      ; Flag for monitoring I2C transmission success.
SErrFLAG   BIT    APPFLAGS.1

FAILCNT:   DS      1

TOGCNT:    DS      1              ; Toggle counter.
```

```
;*******************************************************************************
;
;                                Program Start
;
;*******************************************************************************
          CSEG

; Reset and interrupt vectors.

          AJMP   Reset           ; Reset vector at address 0.

; A timer I timeout usually indicates a 'hung' bus.

          org    0033h
iicint:   ajmp   I2CISR          ; I2C interrupt

          org    0073h
TimerI:   SETB   CLRTI           ; Timer I timeout interrupt.
          AJMP   TIISR           ; Go to Interrupt Service Routine.



;*******************************************************************************
;                     I2C Interrupt Service Routine
;*******************************************************************************
;
; Notes on the interrupt mechanism:
;
; Other interrupts are enabled during this ISR upon return from XRETI.
; Limitations imposed on other ISR's:
; - Should not be long (close to 1000 clock cycles).  A long ISR will cause
;   the I2C bus to 'hang', and a TIMERI interrupt to occur.
; - Other interrupts either do not use the same mechanism for allowing
;   further interrupts, or if they do - disable TIMERI interrupt beforehand.
;
; The 751 hardware allows only one level of interrupts.  We simulate an
; additional level by software: by performing a RETI instruction (at location
; XRETI) the interrupt-in-progress flip-flop is cleared, and other interrupts
; are enabled.  The second level of interrupt is a must in our implementation,
; enabling timeout interrupts to occur during "stuck" wait loops in the I2C
; interrupt service routine.

I2CISR:   CLR    EI2             ; Disable I2C interrupt.
          ACALL  XRETI           ; Allow other interrupts to occur.
          PUSH   PSW
          PUSH   ACC
          MOV    A,R0
          PUSH   ACC
          MOV    A,R1
          PUSH   ACC
          MOV    A,R2
          PUSH   ACC

          MOV    StackSave, SP
          CLR    TIRUN
          SETB   TIRUN

          JB     STP,NoGo
          JNB    MASTER, GoSlave
          MOV    MSGSTAT,#MGO
          JB     STR,GoMaster
NoGo:     MOV    MSGSTAT,#NOTSTR
          AJMP   Dismiss         ; Not a valid Start.

XRETI:    RETI
```

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

;  SLAVE CODE –
;  GET THE ADDRESS

GoSlave:   MOV    MSGSTAT,#SGO
AddrRcv:   ACALL  ClsRcv8
           JNB    DRDY, SMsgEnd   ; Must be some strange Start or Stop
                                  ;   before the address byte was completed.
                                  ;   Not a valid address.
STstRW:    MOV    C,ACC.0         ; Save R/W~ bit in carry.
           CLR    ACC.0           ; Clear that bit, leaving “raw” address
           JZ     GoIdle          ; If it is a General Address
                                  ;   – ignore it.

                                  ; NOTE:
                                  ; One may insert here a different
                                  ;   treatment for general calls, if
                                  ;   these are relevant.

           JC     SlvTx           ; It’s a Read – (requesting slave
                                  ;   transmit).


; It is a Write (slave should receive the message).

;          Check if message is for us

SRcv2:     CJNE   A,MYADDR,GoIdle ; If not my address – ignore the
                                  ;   message.
           MOV    R1,#SRcvBuf     ; Set receive buffer address.
           MOV    R2,#RbufLen+1   ;
           SJMP   SRcv3

SRcvSto:   MOV    @R1,A           ; Store the byte
           Inc    R1              ; Step address.
SRcv3:     ACALL  AckRcv8
           JNB    DRDY,SRcvEnd    ; Exit loop –end reception.
           DJNZ   R2,SRcvSto      ; Go to store byte if buffer not full.

; Too many bytes received – do not acknowledge.
           MOV    MSGSTAT,#SRLNG  ; Notify main that (as slave) we
                                  ;   have received too long a message.
           ACALL  SLnRCvdR        ; Handle new data – slave event routine.
           SJMP   GoIdle


; Received a byte, but not DRDY – check if a legitimate message end.

SRcvEnd:   CJNE   R0,#7,SRcvErr   ; If bit count not 7, it was not
                                  ;   a Start or a Stop.

; Received a complete message

           MOV    MSGSTAT,#SRCVD
                                  ; Calculate number of bytes received
           MOV    A,R1
           CLR    C
           SUBB   A,#SRcvBuf      ; number of bytes in ACC
           ACALL  SRCvdR          ; Handle new data – slave event routine.
           SJMP   SMsgEnd

; It is a Read message, check if for us.
```

```
SlvTx:      NOP

STx2:       CJNE    A,MYADDR,GoIdle   ; Not for us.
            MOV     I2DAT,#0          ; Acknowledge the address.
            JNB     ATN,$             ; Wait for attention flag.
            JNB     DRDY,SMsgEnd      ; Exception – unexpected Start
                                      ;   or Stop before the Ack got out.
            MOV     R1,#STxBuf        ; Start address of transmit buffer.
STxlp:      MOV     A,@R1             ; Get byte from buffer
            INC     R1
            ACALL   XmByte
            JNB     DRDY,SMsgEnd      ; Byte Tx not completed.
            JNB     RDAT,STxlp        ; Byte acknowledge, proceed trans.
            MOV     I2CON,#BCDR+BIDLE ; Master Nak'ed for msg end.
            MOV     MSGSTAT,#STXED
            ACALL   STXedR            ; Slave transmitted event routine.
            AJMP    Dismiss

SRcvErr:    MOV     MSGSTAT,#SRERR    ; Flag bus/protocol error
            ACALL   SRErrR            ; Slave error event routine.
            SJMP    SMsgEnd
StxErr:     MOV     MSGSTAT,#SRERR    ; Flag bus/protocol error
            ACALL   SRErrR

SMsgEnd:    JB      MASTER,SMsgEnd2
            JB      STR,GoSlave       ; If it was a Start, be Slave
SMsgEnd2:   AJMP    Dismiss

; End of Slave message processing

GoIdle:
            AJMP    Dismiss



;
;

GoMaster:


; Send address & R/W~ byte

            MOV     R1,#MasBuf        ; Master buffer address
            MOV     R2,MASTCNT        ; # of bytes, to send or rcv
            MOV     A,DESTADRW        ; Destination address (including
                                      ;   R/W~ byte).
            JB      SUBADD,GoMas2     ; Branch if subaddress is needed.

            ACALL   XmAddr

            JNB     DRDY,GM2
            JNB     ARL,GM3
GM2:        AJMP    AdTxArl           ; Arbitration loss while transmitting
                                      ;   the address.
GM3:        JB      RDAT,Noslave      ; No Ack for address transmission.
            JB      ACC.0, MRcv       ; Check R/W~ bit
            AJMP    MTx

; Handling subaddress case:

GoMas2:     NOP                       ; Subaddress needed. Address in ACC.
            CLR     ACC.0             ; Force a Write bit with address.
            ACALL   XmAddr
```

```
            JNB     DRDY,GM4
            JNB     ARL,GM5
GM4:        AJMP    AdTxArl             ; Arbitration loss while transmitting
                                        ;   the address.

GM5:        JB      RDAT,Noslave        ; No Ack for address transmission.
            MOV     A,DESSUBAD
            ACALL   XmByte              ; Transmit subaddress.
            JNB     DRDY,SMsgEnd2       ; Arbitration loss (by Start or Stop)
            JB      ARL,SMsgEnd2        ; Arbitration loss occurred.
            JB      RDAT,NoAck          ; Subaddress transmission was not ack'ed.
            MOV     A,DESTADRW          ; Reload ACC with address.
            JNB     ACC.0, MTx          ; It's a Write, so proceed
                                        ;   by sending the data.
            ; Read message, needs rp. Start and add. retransmit.

            MOV     I2CON,#BCDR+BXSTR ; Send Repeated Start.
            JNB     ATN,$
            MOV     I2CON,#BCDR         ; Clear useless DRDY while preparing
                                        ;   for Repeated Start.
            JNB     ATN,$               ; expecting an STR.
            JNB     ARL,GM6
            AJMP    MArlEnd             ; oops – lost arbitration.
GM6:        ACALL   XmAddr              ; Retransmit address, this time with the
                                        ;   Read bit set.
            JNB     DRDY,GM7
            JNB     ARL,GM8
GM7:        AJMP    AdTxArl             ; Arbitration loss while transmitting
                                        ;   the address.
GM8:        JB      RDAT,Noslave        ; No Ack – the slave disappeared.
            SJMP    MRcv                ; Proceed receiving slave's data.

; A Write message.  Master transmits the data.

MTx:        NOP

MTxLoop:    MOV     A,@R1               ; Get byte from buffer.
            INC     R1                  ; Step the address.
            ACALL   XmByte
            JNB     DRDY,SMsgEnd2       ; Arbitration loss (by Start or Stop)
            JB      ARL,SMsgEnd2        ; Arbitration loss.
            JB      RDAT,NoAck
            DJNZ    R2,MTxLoop          ; Loop if more bytes to send.

            MOV     MSGSTAT,#MTXED      ; Report completion of buffer
                                        ;   transmission.
            SJMP    MTxStop
NoSlave:    MOV     MSGSTAT,#MTXNOSLV
            SJMP    MTxStop
NoAck:      MOV     MSGSTAT,#MTXNAK
            SJMP    MTxStop


; Master receive – a Read frame

MRcv:       ACALL    ClaRcv8            ; Receive a byte.
            SJMP     MRcv2
MRcvLoop:   ACALL    AckRcv8
MRcv2:      JNB      DRDY,MArl          ; Other's Start or Stop.
            MOV      @R1,A              ; Store received byte.
            INC      R1                 ; Advance address.
            DJNZ     R2,MRcvLoop
```

```
; Received the desired number of bytes – send Nack.

           MOV     I2DAT,#80h
           JNB     ATN,$
           JNB     DRDY,MArl
           MOV     MSGSTAT,#MRCVED
           SJMP    MTxStop            ; Go to send Stop or Repeated Start.


; Conclude this Master message:
; Send Stop, or a Repeated Start

MTxStop:   JNB     RPSTRT,MTxStop2    ; Check if Repeated Start needed
                                      ; Around if not RPSTRT.
           MOV     I2CON,#BCDR+BXSTR  ; Send Repeated Start.
           SJMP    MTxStop3
MTxStop2:  MOV     C,SETMRQ           ; Set new Master Request if demanded
           MOV     MASTRQ,C           ;   by SETMRQ bit of MASCMD.

           MOV     I2CON,#BCDR+BXSTP  ; Request the HW to send a Stop.

MTxStop3:  JNB     ATN,$              ; Wait for Attention
           MOV     I2CON,#BCDR        ; Clear the useless DRDY, generated
                                      ;   by SCL going high in preparation
                                      ;   for thr Stop or Repeated Start.
           JNB     ATN,$              ; Wait for ARL, STP or STR.
           JB      ARL,MarlEnd        ; Lost arbitration trying to send
                                      ;   Stop or a ReStart.

; Master is done with this message.  May proceed with new messages, if any,
; or exit.

           ACALL   MastNext           ; Master Event Routine.  May Prepare
                                      ;   the pointers and data for the
                                      ;   next Master message.

           JNB     MASTRQ,MMsgEnd     ; Go end service routine if MASTRQ
                                      ;   does not indicate that the master
                                      ;   should continue (was set according
                                      ;   to SETMRQ bit, or by MastNext).

           JNB     STR,MMsgEnd        ; Return from the ISR, unless Start
                                      ;   (avoid danger if we do not return:
                                      ;   if there was a Stop, the watchdog
                                      ;   is inactive until next Start).
           AJMP    GoMaster           ; Loop for another Master message
                                      ;
MMsgEnd:                              ; End of Master messages,
           SJMP    Dismiss



; Terminate mastership due to an arbitration loss:

MArl:
           JNB     STR,MArl2          ; If lost arbitration due to other
                                      ;   Master's Start, go be a slave.
           AJMP    GoSlave

Marl2:
           AJMP    Dismiss


; Switch from Master to Slave due to arbitration loss after completing
; transmission of a message.  The MASTRQ bit was cleared trying to write a
```

```
; Stop, and we need to set it again on order to retry transmission when the
; bus gets free again.

MArlEnd:
           SETB     MASTRQ            ; Set Master Request – which will get
                                      ;   into effect when we are done as a
                                      ;   slave.
           AJMP     MArl

; Handling arbitration loss while transmitting an address

AdTxArl:   JB       STR,MArl          ; Non–synchronous Start or Stop.
           JB       STP,MArl

; Switch from Master to Slave due to arbitration loss while transmitting
; an address – complete receiving the address transmitted by the new Master.

           CJNE     R0,#0,AdTxArl2
                                      ; Arl on last bit of address
                                      ;   (R0 is 0 on exit from XmAddr).
           DEC      A                 ; The lsb sent, in which arl occurred
                                      ;   must have been 1. By decrementing
                                      ;   A we get the address that won.
           SJMP     AdAr3

AdTxArl2:
           RR       A                 ; Realign partially Tx'ed ACC
           MOV      R1,A              ;  and save it in R1
           MOV      A,R0              ; Pointer for lookup table
           MOV      DPTR,#MaskTable
           MOVC     A,@A+DPTR
           ANL      A,R1              ; Set address bits to be received,
                                      ;   and the bit on which we lost
                                      ;   arbitration to 0
                                      ;   Now we are ready to receive the rest
                                      ;   of the address.

           MOV      I2CON,#BCXA+BCARL ; Clear flags and release the clock.

           ACALL    RBit3             ; Complete the address using reception
                                      ;   subroutine.
           JB       DRDY,AdAr3        ; Around if received address OK
           AJMP     SMsgEnd           ; Unexpected Start or Stop – end
                                      ;   as a slave.
AdAr3:     AJMP     STstRW            ; Proceed to check the address
                                      ;   as a slave.

MaskTable: DB       0ffh,7Eh,3Eh,1Eh,0Eh,06h,02h,00h, ; 0ffh is dummy

; End I2C Interrupt Service Routine:

Dismiss:   ACALL    I2CDONE

           MOV      I2CON,#BCARL+BCSTP+BCDR+BCXA+BIDLE
           CLR      TIRUN
           POP      ACC
           MOV      R2,A
           POP      ACC
           MOV      R1,A
           POP      ACC
           MOV      R0,A
           POP      ACC
           POP      PSW
           SETB     EI2

           RET                        ; Return from I2C interrupt Service Routine
```

```
;*************************************************************************
;               Byte Transmit and Receive Subroutines
;*************************************************************************


;  XmAddr: Transmit Address and R/W~
;  XmByte: Transmit a byte

XmAddr:    MOV     I2DAT,A             ; Send first bit, clears DRDY.
           MOV     I2CON,#BCARL+BCSTR+BCSTP
                                       ; Clear status, release SCL.
           MOV     R0,#8               ; Set R0 as bit counter
           SJMP    XmBit2
XmByte:    MOV     R0,#8
XmBit:     MOV     I2DAT,A             ; Send the first bit.
XmBit2:    RL      A                   ; Get next bit.
           JNB     ATN,$               ; Wait for bit sent.
           JNB     DRDY,XmBex          ; Should be data ready.
           DJNZ    R0,XmBit            ; Repeat until all bits sent.
           MOV     I2CON,#BCDR+BCXA    ; Switch to receive mode.
           JNB     ATN,$               ; Wait for acknowledge bit.
                                       ; flag cleared.
XmBex:     RET

;
; Byte receive routines.
;
; ClsRcv8  clears the status register (from Start condition)
;          and then receives a byte.
; AckRcv8  Sends an acknowledge, and then receives a new byte.
;          If a Start or Stop is encountered immediately after the
;          ack, AckRcv8 returns with 7 in R0.
; ClaRcv8  clears the transmit active state and releases clock
;          (from the acknowledge).
;
;          A contains the received byte upon return.
;          R0 is being used as a bit counter.
;

ClsRcv8:   MOV     I2CON,#BCARL+BCSTR+BCSTP+BCXA
                                       ; Clear status register.
           JNB     ATN,$
           JNB     DRDY,RCVex
           SJMP    Rcv8

AckRcv8:   MOV     I2DAT,#0            ; Send Ack (low)
           JNB     ATN,$
           JNB     DRDY,RCVerr         ; Bus exception – exit.
ClaRcv8:   MOV     I2CON,#BCDR+BCXA    ; clear status, release clock
                                       ;   from writing the Ack.
           JNB     ATN,$

Rcv8:      MOV     R0,#7               ; Set bit counter for the first seven
                                       ;   bits.
           CLR     A                   ; Init received byte to 0.
RBit:      ORL     A,I2DAT             ; Get bit, clear ATN.
RBit2:     RL      A                   ; Shift data.
           JNB     ATN,$               ; Wait for next bit.
           JNB     DRDY,RCVex          ; Exit if not a data bit (could be Start/
                                       ;   Stop, or bus/protocol error)
RBit3:     DJNZ    R0,RBit             ; Repeat until 7 bits are in.
           MOV     C,RDAT              ; Get last bit, don't clear ATN.
           RLC     A                   ; Form full data byte.
RCVex:     RET

RCVerr:    MOV     R0,#9               ; Return non legitimate bit count
           RET
```

```
;*******************************************************************************
;          Timer I Interrupt Service Routine
;          I2C us Timeout
;*******************************************************************************

; In addition to reporting the timeout in MSGSTAT, we update a failure
; counter, TITOCNT. This allows different types of timeout handling by the
; main program.

TIISR:     CLR     MASTRQ              ; “Manual” reset.
           MOV     I2CON,#BXSTP        ;
           MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP

TI1:       MOV     MSGSTAT,#TIMOUT     ; Status Flag for Main.
TI2:       MOV     A,#0FFh
           CJNE    A,TITOCNT,TI3       ; Increment TITOCNT, saturating
           SJMP    TI4                 ; at FFh.
TI3:       INC     TITOCNT

TI4:       ACALL   RECOVER
           SETB    CLRTI               ; Clear TI interrupt flag.
           ACALL   XRETI               ; Clear interrupt pending flag (in
                                       ;   order to re-enable interrupts).
           MOV     SP,StackSave        ; Realign stack pointer, re-doing
                                       ;   possible stack changes during
                                       ;   the I2C interrupt service routine.
                                       ;   TimerI interrupts in other ISR’s
                                       ;   were not allowed !
           AJMP    Dismiss             ; Go back to the I2C service routine,
                                       ;   in order to return to the (main)
                                       ;   program interrupted.


;*******************************************************************************
;       Bus recovery attempt subroutine
;*******************************************************************************

RECOVER:   CLR     EA
           CLR     MASTRQ              ; “Manual” reset.
           MOV     I2CON,#BCXA+BIDLE+BCDR+BCARL+BCSTR+BCSTP
           CLR     SLAVEN              ; Non I2C TimerI mode
           SETB    TIRUN               ; Fire up TimerI. When it overflows, it
                                       ;   will cause I2C interface hardware reset.
           MOV     R1,#0ffh
DLY5:      NOP
           NOP
           NOP
           DJNZ    R1,DLY5
           CLR     TIRUN
           SETB    CLRTI

           SETB    SCL                 ; Issue clocks to help release other devices.
           SETB    SDA
           MOV     R1,#08h
RC7:       CLR     SCL
           DB      0,0,0,0,0
           SETB    SCL
           DB      0,0,0,0,0
           DJNZ    R1,RC7
           CLR     SCL
           DB      0,0
           CLR     SDA
           DB      0,0
           SETB    SCL
           DB      0,0,0,0,0
           SETB    SDA
           DB      0,0,0,0,0,          ; Issue a Stop.
```

```
Rex:       MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP  ; clear flags
           SETB    EA
           RET

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

; Message ping pong game. Each message is transmitted by
; a processor that is a master on the I2c bus, and it contains one byte
; of data. A processor that receives this data byte as a slave increments
; the data by one and transmits it back as a master. The data received is
; confirmed to be a one increment of the data formerly sent, unless
; it is a “reset” value, chosen to be 00h.
; The two participating processors have similar code, where the node
; address of the second processor is the destination address of this
; one, and vice versa.
; The first data byte each processor tries to send is 00h. One of the
; processors will acquire the bus first, and the second processor that will
; receive this “resetting” 00h will not attempt to confirm it against an
; expected value. It will simply increment and transmit it. Subsequent
; receptions will be confirmed against the expected value, until 0ffh data
; bytes are sent and the game is effectively reset by the 00h resulting from
; the next increment.
; A toggling output (TogLED) tells the outer world that the “ping pong”
; proceeds well. If something unexpected happens we temporarily activate
; another output, ErrLED.
; The different tasks of the code are performed in a combination of main–
; line program and event routines called from the I2C interrupt service
; routine.


;  Initial set–ups:
;      Load CT1,CT0 bits of I2CFG register, according to the clock
;           crystal used.
;      Load RAM location MYADDR with the I2C address of this processor.
;    We load these values out of ROM table locations (R_CTVAL and R_MYADDR).
;    One may, instead, load with a MOV <immediate> command.

Reset:     MOV     SP,#07h             ; Set stack location.
           CLR     A
           MOV     DPTR,#R_CTVAL
           MOVC    A,@A+DPTR
           MOV     I2CFG,A             ; Load CT1,CT0 (I2C timing, crystal
                                       ;   dependent).
           CLR     A
           MOV     DPTR,#R_MYADDR
           MOVC    A,@A+DPTR           ; Get this node’s address from ROM table
           MOV     MYADDR,A            ;   into MYADDR RAM location.

           CLR     OnLED

Reset2:    CLR     ErrLED              ; Flash LED.
           ACALL   LDELAY
           SETB    ErrLED
           CLR     SErrFLAG
           CLR     TRQFLAG
           MOV     FAILCNT,#50h
           SETB    TogLED
           MOV     TOGCNT,#050h        ; Initialize pin–toggling counter

; Enable slave operation.
; The Idle bit is set here for a restart situation – in normal
```

```
; operation this is redundant, as this bit is set upon power_up reset.
           MOV     I2CON,#BIDLE        ; Slave will idle till next Start.
           SETB    SLAVEN              ; Enable slave operation.

; Enable interrupts.
; This is necessary for both Slave and Master operations.
           SETB    ETI                 ; Enable timer I interrupts.
           SETB    EI2                 ; Enable I2C port interrupts.
           SETB    EA                  ; Enable global interrupts.

; Set up Master operation.

           MOV     MASCMD,#0h          ; “Regular” master transmissions.
           MOV     DPTR,#PongADDR
           CLR     A
           MOVC    A,@A+DPTR
           MOV     DESTADRW,A          ; The partner address. The LSB is
                                       ;   low, for a Write transaction.
           MOV     MASTCNT,#01h        ; Message length – a single byte.

PPSTART:
           MOV     MasBuf,#00h

; “Ping” transmission:

PP2:
           SETB    TRQFLAG
           SETB    MASTRQ
           MOV     R1,#0ffh
PP22:      JNB     TRQFLAG,PP3         ; Transmitted OK
           DJNZ    R1,PP22
MFAIL1:    DJNZ    FAILCNT,PP2
           ACALL   RECOVER
           SJMP    Reset2

; “Pong” reception:

PP3:       MOV     R0,#0ffh            ; Software timeout loop count.
PP31:      MOV     R1,#0ffh
PP32:      JB      TRQFLAG,PP2         ; Rcvd ok as slave, go transmit.
           JB      SErrFLAG,PP5
           DJNZ    R1,PP32
           DJNZ    R0,PP31
PPTO:      ACALL   RECOVER             ; Software timeout.
           AJMP    Reset2

PP5:       CLR     ErrLED              ; Receive error.
           ACALL   LDELAY
           SETB    ErrLED
           CLR     SErrFLAG
           AJMP    PPSTART

LDELAY:    MOV     R2,#030h
LDELAY1:   MOV     R1,#0ffh
           DJNZ    R1,$
           DJNZ    R2,LDELAY1

           RET
```

```
;*******************************************************************************
;  Slave and Master Event Routines.
;*******************************************************************************

;
;  Invoked upon completion of a message transaction.
;  This is the part of the application program actually dealing
;  with the data communicated on the I2C bus, by responding to
;  new data received and/or preparing the next transaction.


; Slave Event Routines
;
; These routines are invoked by the I2C interrupt service routine when a
; message transaction as a slave has been completed. Our “application”
; reacts to a message received as a slave with the routine SRCvdR.
; The calls that indicate erroneous reception are treated the same way as
; erroneous data reception in the “ping pong” game.

;  SRcvdR
;          Invoked when a new message has been received as a Slave.

SRcvdR:    NOP
           MOV     A,SRcvBuf
           JNZ     SR2
           MOV     MasBuf,#01h         ; It was ping–pong reset value
           SJMP    SR3

SR2:       INC     MasBuf              ; The expected data.
           CJNE    A,MasBuf,ErrSR
           INC     MasBuf              ; Data for next transmission – the data
                                       ;   received incremented by 1.

;          A successful two way data exchange. Let the outside world know by
;          toggling an output pin driving a LED. We actually toggle only
;          when a number of such exchanges is completed, in order to
;          slow down the changes for a good visual indication.

           DJNZ    TOGCNT,SR3
           CPL     TodLED              ; Toggle output
           MOV     TOGCNT,#050h        ;

SR3:       CLR     SErrFLAG
           SETB    TRQFLAG             ; Request main to transmit
           RET

ErrSR:     SETB    SErrFLAG
           RET


;   SLnRcvdR
;          Invoked when a message received as a Slave is too long
;          for the receive buffer.

;   STXedR
;          Invoked when a Slave completed transmission of its buffer.
;          We do not expect to get here, since we do not plan to have
;          in our system a master that will request data from this node.
;

;   SRErrR
;          Slave error event subroutine.
;          In most applications it will not be used.
;
```

```
SLnRcvdR:
STXedR:
SRErrR:    JMP     ErrSR


;
;  MastNext – Master Event Routine.
;
;          Invoked when a Master transaction is completed, or terminated
;          “willingly” due to lack of acknowledge by a slave.
;

MastNext:
           MOV     A,MSGSTAT
           CJNE    A,#MTXED,MN1
           MOV     RAILCNT,#50h
           CLR     TRQFLAG
           RET
MN1:
           RET

; I2CDONE
;          Called upon completion of the I2C interrupt service routine.
;          In this example it monitors exceptions, and invokes the bus
;          recovery routine when too many occurred.

I2CDONE:
           MOV     A,MSGSTAT
           CJNE    A,#NOTSTR,I2CD1
           DJNZ    FAILCNT,I2CD1
           MOV     FAILCNT,#01h        ; Too many “illegal” i2c interrupts
           CLR     EI2                 ; – shut off.
I2CD1:     RET

;*************************************************************************
;          I2C Communications Table:
;*************************************************************************


; We used table driven values for clarity. One may use immediates to load
; these values and save several lines of code.

; Contents is used in the beginning of the main program to load
; RAM location MYADDR and the I2CFG register.
; The node address, in R_MYADDR, is application specific, and unique for
; each device in the I2C network.
; R_CTVAL depends on the crystal clock frequency.

R_MYADDR:   DB     4Ch         ; This node’s address

R_CTVAL:    DB     02h         ; CT1, CT0 bit values

;*************************************************************************
;          Application Code Definitions
;*************************************************************************

PongADDR:   DB     48h         ; The address of the “partner” in
                               ;   the ping–pong game.


; EPROM configuration bit definitions for the 87LPC764.

      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
```

Using the 87LPC76X in multi-master I^{2}C applications AN465

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 06851

Let's make things better.

Philips Semiconductors

PHILIPS