Lecture Four: Interfacing and Communication

1. Introduction to Interfacing

Interfacing is the process of connecting devices together so that they can exchange information. There are three components to microcomputer interfacing. The first step is the mechanical design of the physical components. Often, the mechanical design is simply selecting the physical devices from a list of available components. The next step is the analog and digital electronics used to connect the physical devices to the computer. The voltage levels of the external device must be translated into values compatible with the microcontroller. The RS232 interface using the MAX3232 interface is a typical example of this translation. Some external devices need the interface to source or sink current. The input/output information may be encoded as simple digital signals or variable analog signals. More complex systems may use frequency, period, phase, or pulse width to represent the signals. The third component of interfacing is the low-level software that transforms the mechanical and electrical devices into objects that perform the desired tasks. The group of these low level functions is often designated as an I/O device driver.

2. Synchronous Serial Interface, SSI

Microcontrollers employ multiple approaches to communicate synchronously with peripheral devices and other microcontrollers. The synchronous serial interface (SSI) system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. With multiple slaves, the configuration can be a star (centralized master connected to each slave), or a ring (each node has one receiver and one transmitter, where the nodes are connected in a circle.) The master initiates all data communication. Stellaris ® and Tiva ® microcontrollers have 0 to 4 Synchronous Serial Interface or SSI modules. Another name for this protocol is Serial Peripheral Interface or SPI. The fundamental difference between a UART, which implements an asynchronous protocol, and a SSI, which implements a synchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (UART) operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two UART devices can communicate with each other as long as the two clocks have frequencies within ±5% of each other. Two devices communicating with synchronous serial

interfaces (SSI) operate from the same clock (synchronized). With a SSI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.) The SSI protocol includes four I/O lines. The slave select SSI0Fss is an optional negative logic control signal from master to slave signal signifying the channel is active. The second line, SCK, is a 50% duty cycle clock generated by the master. The SSI0Tx (master out slave in, MOSI) is a data line driven by the master and received by the slave. The SSI0Rx (master in slave out, MISO) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data. Figure 5.1 shows the I/O port locations of the SSI ports discussed in this lecture.

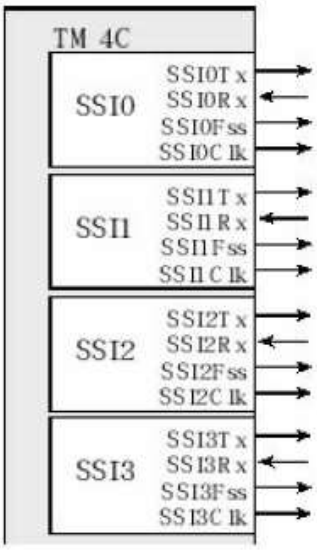


Fig. 5.1 Synchronous serial port pins on Stellaris ® TM4C microcontrollers.

On the TM4C the shift register can be configured from 4 to 16 bits. The shift register in the master and the shift register in the slave are linked to form a distributed register. Figure 5.2 illustrates communication between master and slave. Typically, the microcontroller and the I/O device slave are so physically close we do not use interface logic. The interface is classified as synchronous because the hardware clock is shared between devices. The SSI on the TM4 Employs two hardware FIFOs. Both FIFOs are 8 elements deep and 4 to 16 bits wide, depending on the selected data width. When performing I/O the software puts into the transmit FIFO by writing to the SSI0_DR_R register and gets from the receive FIFO by reading from the SSI0_DR_R register.

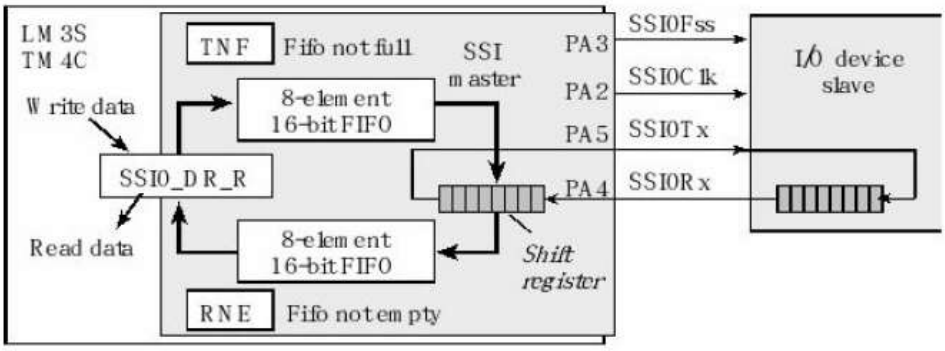


Figure 5.2. A synchronous serial interface between a microcontroller and an I/O device.

Table 5.1 lists the SSI0 registers on the TM4C.

Address	31-6				3	2	1	0	Name
$400F.E61C					SSI3	SSI2	SSI1	SSI0	SYSCTL_RCGCSSI_R
	31-16	15-8		7	6	5-4	3-0		
$4000.8000		SCR		SPH	SPO	FRF	DSS		SSI0_CR0_R
	31-16	15-0							
$4000.8008		Data							SSI0_DR_R
	7	6	5	4	3	2	1	0	
$4000.8004					SOD	MS	SSE	LBM	SSI0_CR1_R
$4000.800C				BSY	RFF	RNE	TNF	TFE	SSI0_SR_R
$4000.8010	CPSDVSR								SSI0_CPSR_R
$4000.8014					TXIM	RXIM	RTIM	RORIM	SSI0_IM_R
$4000.8018					TXRIS	RXRIS	RTRIS	RORRIS	SSI0_RIS_R
$4000.801C					TXMIS	RXMIS	RTMIS	RORMIS	SSI0_MIS_R
$4000.8020							RTIC	RORIC	SSI0_ICR_R
$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	**SEL**	GPIO_PORTA_AFSEL_R
$4000.441C	DEN	DEN	DEN	DEN	**DEN**	**DEN**	DEN	**DEN**	GPIO_PORTA_DEN_R
$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	**DIR**	GPIO_PORTA_DIR_R
$400F.E608	GPIOH	GPIOG	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	**GPIOA**	SYSCTL_RCGCGPIO_R

Table 5.1. The TM4C SSI0 registers. Each register is 32 bits wide. Bits 31 − 8 are zero.

If there is data in the transmit FIFO, the SSI module will transmit it. With SSI it transmits and receives bits at the same time. When a data transfer operation is performed, this distributed 8- to 32- bit register is serially shifted 4 to 16 bit positions by the SCK clock from the master so the data is effectively exchanged between the master and the slave. Data in the master shift register are transmitted to the slave. Data in the slave shift register are transmitted to the master. Typically, the microcontroller is master and the I/O module is the slave, but one can operate the microcontroller in slave mode.

The SSI clock frequency is established by the 8-bit field SCR field in the SSI0_CR0_R register and the 8-bit field CPSDVSR field in the SSI0_CPSR_R register. SCR can be any 8-bit value from 0 to 255. CPSDVSR must be an even number from 2 to 254. Let f BUS be the frequency of the bus clock.

The frequency of the SSI is

f SSI = f BUS / (CPSDVSR * (1 + SCR))

Common control features for the SSI module include:

Baud rate control register, used to select the transmission rate

Data size 4 to 16 bits

Mode bits in the control register to select master versus slave

Freescale mode with clock polarity and clock phase

Interrupt arm bit

Ability to make the outputs open drain (open collector)

Common status bits for the SPI module include:

BSY, SSI is currently transmitting and/or receiving a frame, or the transmit FIFO is not empty

RFF, SSI receive FIFO is full

RNE, SSI receive FIFO is not empty

TNF, SSI transmit FIFO is not full

TFE, SSI transmit FIFO is empty

The key to proper transmission is to select one edge of the clock (shown as “T” in Figure 5.3) to be used by the transmitter to change the output, and use the other edge (shown as “R”) to latch the data in the receiver. In this way data is latched during the time when it is stable.

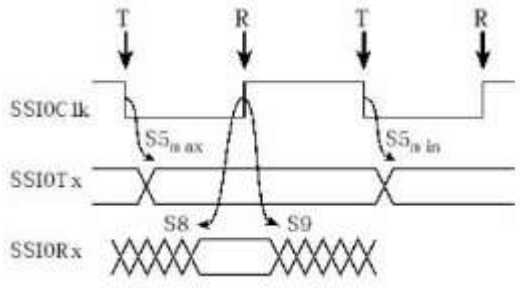


Figure 5.3. Synchronous serial timing showing the data available interval overlaps the data required interval.

3. Nokia 5110 Graphics LCD Interface

In this section we will interface a Nokia 5110 LCD using busy-wait synchronization. See Figure 5.4. Before we output data or commands to the display, we will check a status flag and wait for the previous operation to complete. Busy-wait synchronization is very simple and is appropriate for I/O devices that are fast and predicable.

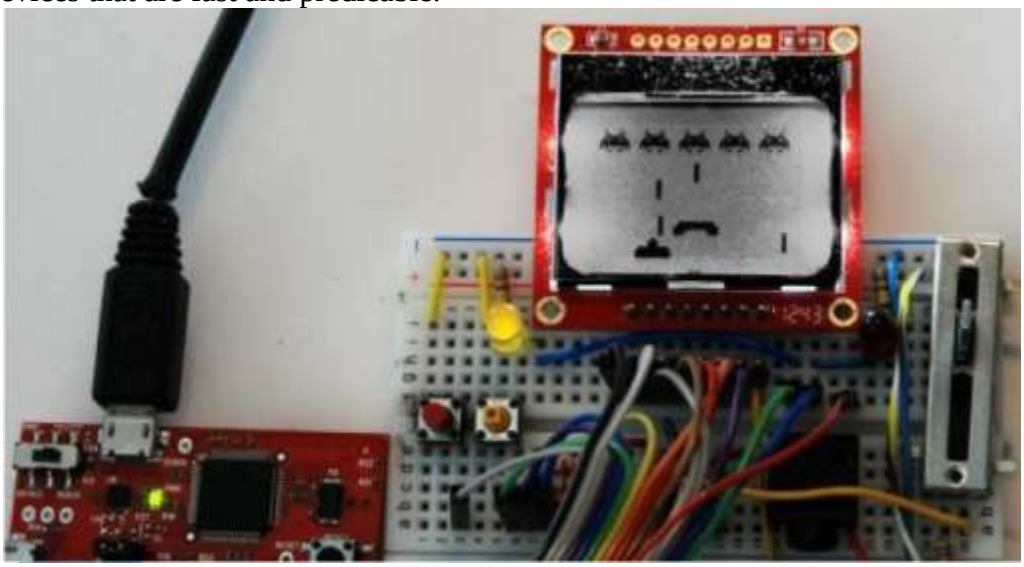


Fig. 5.4 Nokia 5110 display with 84 by 48 monochrome pixels.

The Nokia 5110 uses the synchronous serial interface (SSI) described in the last section to control PA5 (MOSI), PA3 (Fss), and PA2 (Sclk), as shown in Figure 5.5. Pins PA7 and PA6 are regular GPIO pins. The microcontroller will be master and the LCD slave. There are multiple Nokia 5110 displays for sale on the market with the same LCD but different pin locations for the signals. Figure 5.5 shows two of the possible pin configurations. Please look on your actual display for the pin name and not the pin number. Be careful when connecting the backlight, because at 3.3V the back light draws 80 mA. If you want a dimmer back light connect 3.3V to a 100 ohm resistor, and the other end of the resistor to the LED/BL pin.

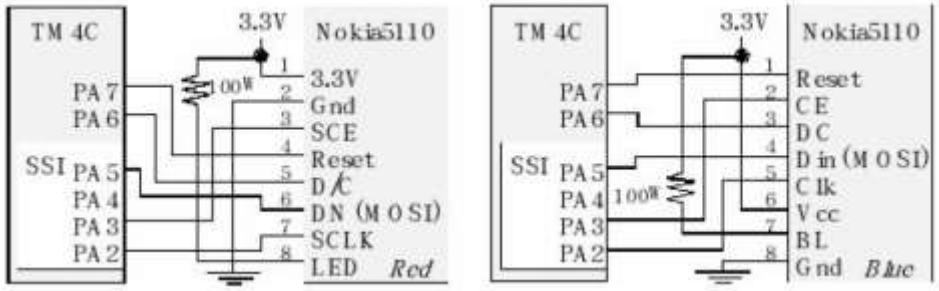


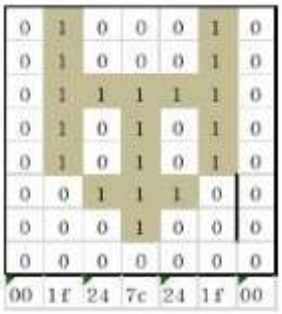
Fig. 5.5 Nokia 5110 interface to a TM4C123.

Program 5.1 lists the prototypes for public functions available in the software starter project. The **Init** function must be called once, before any of the other functions can be called. The **SetCursor** function, defines where on the screen subsequent character output will occur. Each ASCII character is 7 pixels wide and 8 pixels high. This means there can be 84/7 = 12 characters by 48/8 = 6 rows. The cursor is defined by character position, not pixel location, so $0 \le$ **newX** ≤ 11 and $0 \le$ **newY** ≤ 5, with 0,0 being the top row on left. The **Clear** function erases the entire screen. It

takes 4,032 bits, or 504 bytes, to represent an entire 84 by 48 pixel image. The **DrawFullImage** function takes a 504-byte array and copies it onto the display.

```
void Nokia5110_Init(void);
void Nokia5110_SetCursor(uint8_t newX, uint8_t newY);
void Nokia5110_Clear(void);
void Nokia5110_DrawFullImage(const uint8_t *ptr);
void Nokia5110_OutChar(char data);
void Nokia5110_OutString(char *ptr);
void Nokia5110_OutUDec(unsigned uint16_t n);
```

Program 5.1. Software prototypes for Nokia 5110 display.



The **OutChar OutString** and **OutUDec** functions draw ASCII characters on the screen. These three functions maintain a cursor so you can call these three functions in any order. The matrix ASCII**[][5]** contains the pixel image for each character.

4. Scanned Keyboards

In a **scanned interface**, the switches are placed in a row/column matrix. In this way, many keys can be interfaced with just a few I/O pins. Figure 5.6 shows a matrix keyboard with 4 rows and 4 columns. In general, if there are **n** rows and **m** columns, there could be **n*m** switches, but we would need only **n+m** I/O pins. The four outputs signifies open collector (an output with two states Hi Z and low.)

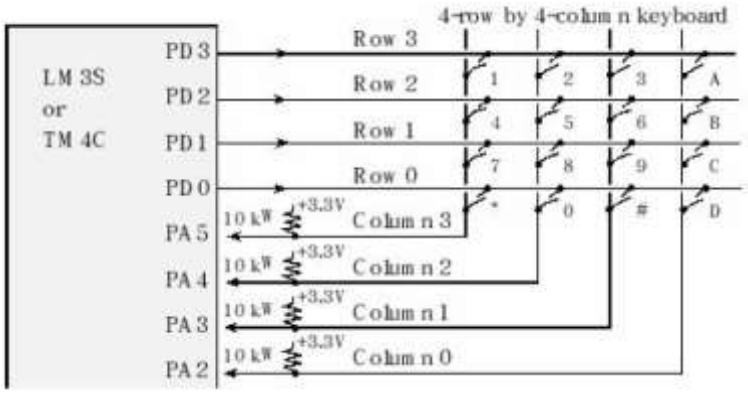


Fig. 5.6 A matrix keyboard interfaced to the microcomputer.

The computer drives one row at a time to zero, while leaving the other rows at Hi Z. By reading the column, the software can detect if a key is pressed in that row. The software "scans" the device by checking all rows one by one. For most microcontrollers, the open collector functionality can

be implemented by toggling the direction register. Remember, open collectors have two states low and off.

5. Inter-Integrated Circuit (I2C) Interface

5.1. The Fundamentals of I2C

Ever since microcontrollers have been developed, there has been a desire to reduce the size of an embedded system, reduce its power requirements, and increase its performance and functionality. Two mechanisms to make systems smaller are to integrate functionality into the microcontroller and to reduce the number of I/O pins. The inter-integrated circuit I2C interface was proposed by Philips in the late 1980s as a means to connect external devices to the microcontroller using just two wires. The SSI interface has been very popular, but it takes 3 wires for simplex and 4 wires for full duplex communication. In 1998, the I2C Version 1 protocol become an industry standard and has been implemented into thousands of devices. The I2C bus is a simple two-wire bi-directional serial communication system that is proposed for communication between microcontrollers and their peripherals over short distances. It also provides flexibility, allowing additional devices to be connected to the bus for further expansion and system development. The interface will operate at baud rates of up to 100 kbps with maximum capacitive bus loading. The module can operate up to a baud rate of 400 kbps provided the I2C bus slew rate is less than 100ns. Version 2.0 supports a high speed mode with a baud rate up to 2.4 MHz (supported by LM4F/TM4C).

Figure 5.7 shows a block diagram of a communication system based on the I2C interface. The master/slave network may consist of multiple masters and multiple slaves. The **Serial Clock Line** (SCL) and the **Serial Data line** (SDA) are both bidirectional. Each line is open drain, meaning a device may drive it low or let it float. A logic high occurs if all devices let the output float, and a logic low occurs when at least one device drives it low. The value of the pull-up resistor depends on the speed of the bus. 4.7 k Ω is recommended for baud rates below 100 kbps, 2.2 k Ω is recommended for standard mode, and 1 k Ω is recommended for fast mode.

The SCL clock is used in a synchronous fashion to communicate on the bus. Even though data transfer is always initiated by a master device, both the master and the slaves have control over the data rate. The master starts a transmission by driving the clock low, but if a slave wishes to slow down the transfer, it too can drive the clock low (called **clock stretching**). In this way, devices on the bus will wait for all devices to finish. Both address (from Master to Slaves) and information (bidirectional) are communicated in serial fashion on SDA.

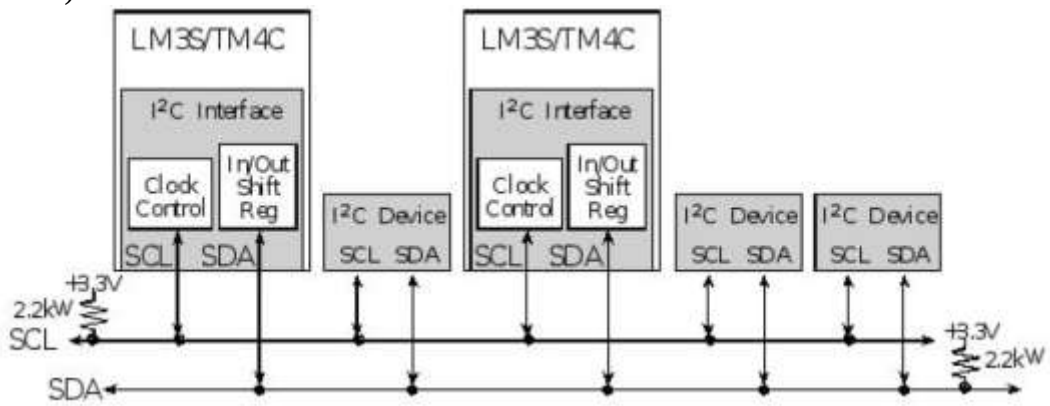


Figure 5.7. Block diagram of an I2C communication network.

The bus is initially idle where both SCL and SDA are both high. This means no device is pulling SCL or SDA low. The communication on the bus, which begins with a START and ends with a STOP, consists of five components:

START (S) is used by the master to initiate a transfer

DATA is sent in 8-bit blocks and consists of 7-bit address and 1-bit direction from the master control code for master to slaves information from master to slave information from slave to master

ACK (A) is used by slave to respond to the master after each 8-bit data transfer

RESTART (R) is used by the master to initiate additional transfers without releasing the bus

STOP (P) is used by the master to signal the transfer is complete and the bus is free. The basic timings for these components are drawn in Figure 5.8. For now we will discuss basic timing. A slow slave uses clock stretching to give it more time to react, and masters will use control when two or more masters want the bus at the same time. An idle bus has both SCL and SDA high. A transmission begins when the master pulls SDA low, causing a START (S) component. The timing of a RESTART is the same as a START. After a START or a RESTART, the next 8 bits will be an address (7-bit address plus 1-bit direction). There are 128 possible 7-bit addresses, however, 32 of them are reserved as special commands. The address is used to enable a particular slave. All data transfers are 8 bits long, followed by a 1-bit acknowledge. During a data transfer, the SDA data line must be stable (high or low) whenever the SCL clock line is high. There is one clock pulse on SCL for each data bit, the MSB being transferred first. Next, the selected slave will respond with a positive acknowledge (Ack) or a negative acknowledge (Nack). If the direction bit is 0 (write), then subsequent data transmissions contain information sent from master to slave. For a write data transfer, the master drives the SDA data line for 8 bits, then the slave drives the acknowledge condition during the 9th clock pulse. If the direction bit is 1 (read), then subsequent data transmissions contain information sent from slave to master. For a read data transfer, the slave drives the SDA data line for 8 bits, then the master drives the acknowledge condition during the 9th clock pulse. The STOP component is created by the master to signify the end of transfer. A STOP begins with SCL and SDA both low, then it makes the SCL clock high, and ends by making SDA high. The rising edge of SDA while SCL is high signifies the STOP condition.

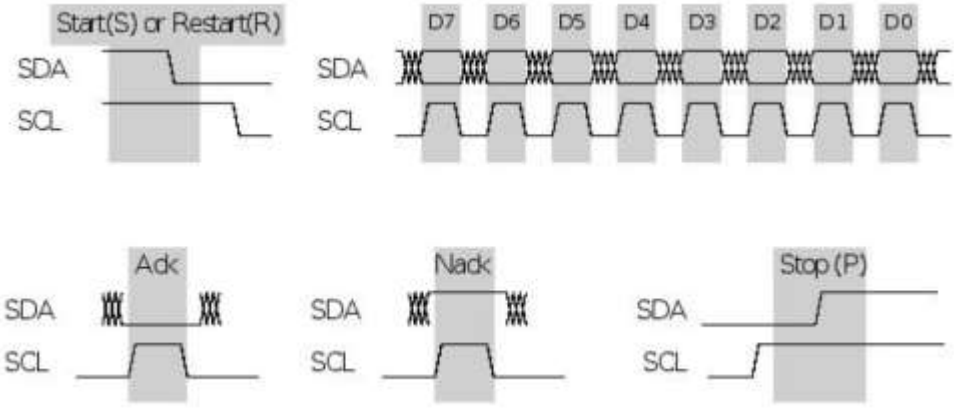


Figure 5.8. Timing diagrams of I2C components.

Figure 5.9 illustrates the case where the master sends 2 bytes of data to a slave. The shaded regions demark signals driven by the master, and the white areas show those times when the signal is driven by the slave. Regardless of format, all communication begins when the master creates a START component followed by the 7-bit address and 1-bit direction. In this example, the direction is low, signifying a write format. The 1st through 8th SCL pulses are used to shift the

address/direction into all the slaves. In order to acknowledge the master, the slave that matches the address will drive the SDA data line low during the 9th SCL pulse. During the 10th through 17th SCL pulses sends the data to the selected slave. The selected slave will acknowledge by driving the SDA data line low during the 18th SCL pulse. A second data byte is transferred from master to slave in the same manner. In this particular example, two data bytes were sent, but this format can be used to send any number of bytes, because once the master captures the bus it can transfer as many bytes as it wishes. If the slave receiver does not acknowledge the master, the SDA line will be left high (Nack). The master can then generate a STOP signal to abort the data transfer or a RESTART signal to commence a new transmission. The master signals the end of transmission by sending a STOP condition.

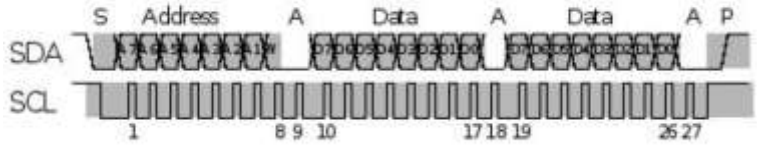


Figure 5.9. I2C transmission of two bytes from master to slave

Figure 5.10 illustrates the case where a slave sends 2 bytes of data the master. Again, the master begins by creating a START component followed by the 7-bit address and 1-bit direction. In this example, the direction is high, signifying a read format. During the 10th through 17th SCL pulses the selected slave sends the data to the master. The selected slave can only change the data line while SCL is low and must be held stable while SCL is high. The master will acknowledge by driving the SDA data line low during the 18th SCL pulse. Only two data bytes are shown in Figure 5.10, but this format can be used to receive as any many bytes the master wishes. Except for the last byte all data are transferred from slave to master in the same manner. After the last data byte, the master does not acknowledge the slave (Nack) signifying 'end of data' to the slave, so the slave releases the SDA line for the master to generate STOP or RESTART signal. The master signals the end of transmission by sending a STOP condition.

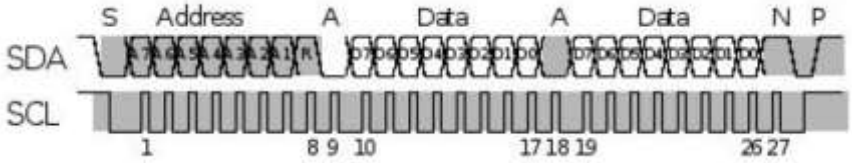


Figure 5.10. I2C transmission of two bytes from slave to master.

Figure 5.11 illustrates the case where the master uses the RESTART command to communicate with two slaves, reading one byte from one slave and writing one byte to the other. As always, the master begins by creating a START component followed by the 7-bit address and 1-bit direction. During the first start, the address selects the first slave and the direction is read. During the 10th through 17th SCL pulses the first slave sends the data to the master. Because this is the last byte to be read from the first slave, the master will not acknowledge letting the SDA data float high during the 18th SCL pulse, so the first slave releases the SDA line. Rather than issuing a STOP at this point, the master issues a repeated start or RESTART. The 7-bit address and 1-bit direction transferred in the 20th through 27th SCL pulses will select the second slave for writing. In this example, the direction is low, signifying a write format. The 28th pulse will be used by the second slave pulls SDA low to acknowledge it has been selected. The 29th through 36th SCL pulses send the data to the second slave. During the 37th pulse the second slave pulls SDA low to acknowledge the data it received. The master signals the end of transmission by sending a STOP condition.

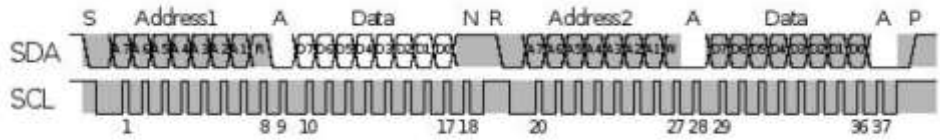


Figure 5.11. I2C transmission of one byte from the first slave and one byte to a second slave.

5.2. LM3S/TM4C I2C Details

LM3S/TM4C microcontrollers have zero to ten I2C modules, see Figure 5.12. Microcontroller pins SDA and SCL can be connected directly to an I2C network. Table 5.2 lists the I2C ports on the LM3S. The LM3S can operate in slave mode, but we will focus on master mode.

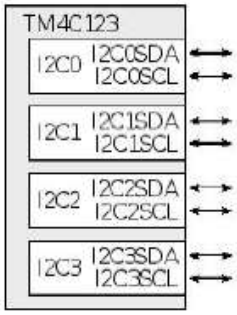


Figure 5.12. I/O port pins for I2C on various LM3S/TM4C microcontrollers.

	7	6	5	4	3	2	1	0	Name
$4002.0000	SA							R/S	I2C0_MSA_R
$4002.0004		BUSBSY	IDLE	ARBLST	DATACK	ADRACK	ERROR	BUSY	I2C0_MCS_R
$4002.0008	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	I2C0_MDR_R
$4002.000C		**TPR**							I2C0_MTPR_F
$4002.0010								IM	I2C0_MIMR_F
$4002.0014								RIS	I2C0_MRIS_R
$4002.0018								MIS	I2C0_MMIS_F
$4002.001C								IC	I2C0_MICR_R
$4002.0020			SFE	**MFE**				LPBK	I2C0_MCR_R
$4000.5420	SEL	SEL	SEL	SEL	**SEL**	**SEL**	SEL	SEL	GPIO_PORTB_
$4000.551C	DEN	DEN	DEN	DEN	**DEN**	**DEN**	DEN	DEN	GPIO_PORTB_
$4000.550C	SEL	SEL	SEL	SEL	SEL	SEL	**SEL**	**SEL**	GPIO_PORTB_
$400F.E608	GPIOH	GPIOG	GPIOF	GPIOE	GPIOD	GPIOC	**GPIOB**	GPIOA	SYSCTL_RCG
$400F.E620					I2C3	I2C2	I2C1	I2C0	SYSCTL_RCG

Table 5.2. The LM3S I2C master registers. Each register is 32 bits wide. Bits 31 – 8 are zero.