Revision: 26 Oct 2018 (JFF)
Richard W. Wall, University of Idaho, rwall@uidaho.edu

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 8: Synchronous I²C Serial Communications

Purpose

The purpose of this project is to investigate concepts involving synchronous communications using a basic master-slave multi-drop network communications. The project uses the I2C protocol to communicate with the 24LC256 I2C CMOS Serial EEPROM.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB IDE
3. IO pin control
4. Use of logic analyzer or oscilloscope
5. Familiarity with the I^2C Serial Protocol

Equipment List

1. Digilent Cerebot 32MX7ck processor board with USB cable
2. Microchip MPLAB X IDE
3. Microchip MPLAB XC32 Compiler
4. Digilent PmodCLP Parallel Character LCD
5. Logic analyzer (Digilent Analog Discovery)

Software Resources

1. XC32 C/C++ Compiler Users Guide
2. PIC32 Peripheral Libraries for MBLAB C32 Compiler
3. PIC32 Family Hardware Reference Manual Section 16 Output Compare
4. Cerebot MX7ck Board Reference Manual
5. MPLAB ® X Integrated Development Environment (IDE)\

References

1. 24LC256 I2C EEPROM datasheet
2. I2C Specification
3. C Programming Reference

Synchronous Serial Communications

As discussed in Project 7, serial communications involves sending data one bit at a time as opposed to Project 6 where eight bits of data were sent at a time to the LCD. One of the biggest motivations for implementing serial communications is to minimize the number of processor pins and wires needed to pass data between two points. The longer the physical distance between the end points, the more expensive the communications media becomes.

This project will focus on the Inter-Integrated Circuit (I2C) protocol, which is only one out of a list of many. As supplemental information, Appendix B list provides a partial list of the available serial protocols. You may ask, "Why so many serial protocols?" The simple answer is that the protocols are optimized for data rates, physical media, physical distance, and connectivity. Some protocols are strictly for point to point communications such as the asynchronous serial communications used in Project 7. Other protocols are for multi drop applications where many senders can be connected to many receivers – a network is an example.

In contrast to asynchronous serial communication, synchronous serial communication requires a synchronizing clock. The synchronizing clock may be a separate signal (I2C and SPI, for example), or embedded into the data (USB, for example).

There are two kinds of operating modes for multi-drop or network configurations; master – slave and peer-to-peer. Master-slave operates similar to the microprocessor-LCD communication considered in Project 6. The microprocessor is the master and it dictates the data direction and the timing of the data exchange between the master and slave units. I2C (Inter Integrated Circuit Communications – also sometimes written as I^2C) and SPI (Serial Peripheral Interface) are examples of master-slave networks supported directly by hardware in many microprocessors. I2C is a half-duplex scheme where the slave devices are enabled or selected by encoding data in a message sent by the master. SPI uses master controlled dedicated device select signals along with separate data send and receive signals to enable simultaneous communications (full-duplex) between the master and a specific slave device.

Peer-to-peer communications on the other hand allows data exchange at any time and between any two communications nodes. Full duplex UART communications is one example of peer-to-peer communications. RS-232, CAN and Ethernet communications are additional examples of peer-to-peer communications.

I2C Protocol

The I2C protocol was developed in the early 1980's by Philips Semiconductors. Its original purpose was to provide an easy way to connect a CPU to peripheral chips in a TV-set. The original specification supported data rates up to 100 Kbits/s. Conventional hardware now supports up to 400 Kbits/s, and the standard has been extended even higher.

At any one time, there is only one active master and one or more slave devices. Each slave device has a unique device identification number. The data is always sent as 8 bit unsigned characters. An I2C message is always initiated with a start condition and terminated with a stop condition that the master controls. Each byte of data exchanged between the master and slave device is acknowledged by the receiving master or slave.

Figure 1 shows the general format of an I2C message. There are two signals used in the I2C protocol: the serial clock signal (SCL) and the serial data signal (SDA). Other than the start and stop conditions, the SDA is not allowed to change states while the SCL signal is high. When the I2C bus is in an idle state, both the SCL and the SDA signals are in the high state.

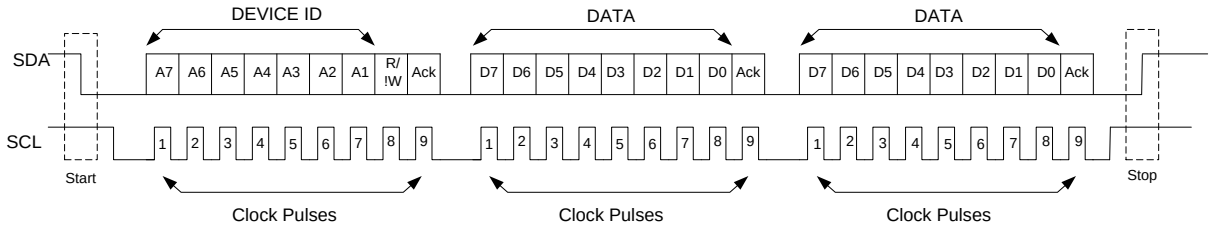

Figure 1. Waveform showing the clock and data timing for an I2C message.

The Master always initiates communications with all slave devices that share the SDA and SCL signals by generating a start condition. A start condition is generated by the master pulling the SDA line low (causing a high-to-low transition) while the SCL line is high. Because data is clocked in on the rising edge of SCL, the master must pull the SCL line low before data is transmitted. Data bits are sent by asserting the SDA signal high or low followed by asserting the SCL signal high for a specified interval of time. The most significant data bit is sent first.

The master always sends the first 8 data bits that consists of a 7-bit slave device address identified as A7 through A1 in Figure 1 and a control bit, R/!W, that specifies the direction of data flow for successive communications. The R/!W bit is low if subsequent data is to be written from the master to the slave device. If the R/!W bit is high, then subsequent bits of data are read from the slave.

Each slave device has, within the device's hardware, a unique identification number. The device that has the identification number that matches the device address field sent from the master will acknowledge the first byte in the I2C message with an ACK bit. The microprocessor serving as the I2C master will terminate communications if no acknowledge is generated after any 8-bit data transfer from the master to the slave. The communications message is terminated by the stop condition, generated by the master allowing the SDA line to float high (causing a low-to-high transition) while the SCL line is high.

I²C PHYSICAL LAYER

I2C networks consist of a data signal (SDA) and a clock signal (SCL) that have a common reference – usually digital ground. A pull-up resistor to V_{DD} is required to be connected to each signal line. Both clock and data signals are connected in a wired-AND configuration that require open collector (also called open drain for CMOS transistors) outputs from both master and slave devices. The output of both devices must be in the open collector state for the signal line to be in the high state (also called the recessive state). The SDA or SCL line is low if the output transistor of either the master or any slave device pulls the signal low. The low state is also referred to as the dominant state. As shown in Figure 2, both slave and master devices can always determine the state of the SCL or SDA lines.

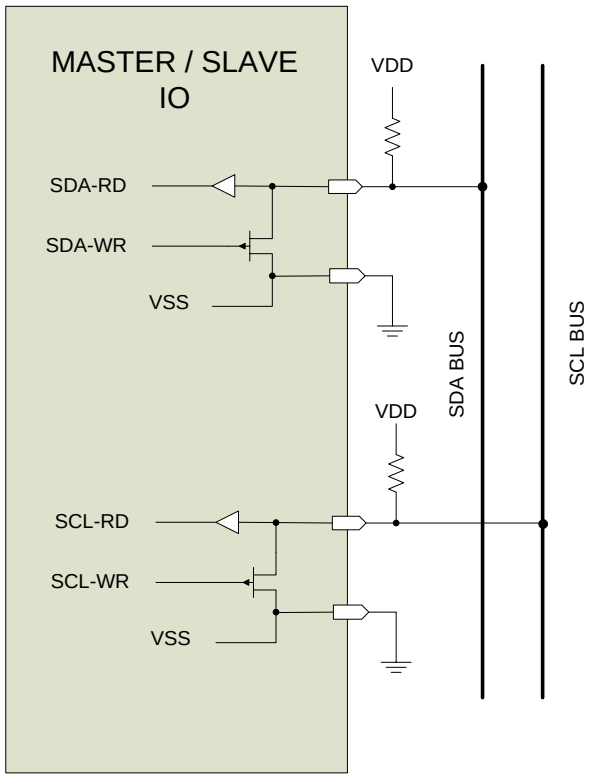

Figure 2. Circuit diagram of I2C device pins

As stated above, the SDA cannot change while the SCL is high except to implement the START and STOP conditions. As illustrated in Figure 3, a start condition is generated when the SDA line makes a recessive to dominant transition when the SCL line is in the recessive state. Figure 4 shows that a STOP condition is generated when the SDA line makes a dominant to recessive transition while the SCL line is in the recessive state. Figure 5 shows that data is either written to the slave or read from the slave during the period identified as 2 to 3. The master or slave (depending on whether it is a write or read operation) is allowed to change the state of the SDA

line while the SCL line is in the dominant state identified as 1 and 4 in Figure 5. Figure 6 shows an acknowledge sequence that is generated by the slave after the master writes a byte of data or is generated by the master after the master reads a byte of data from the slave.

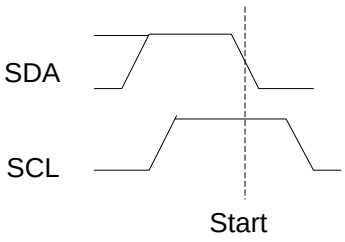

Figure 3. I2C START condition

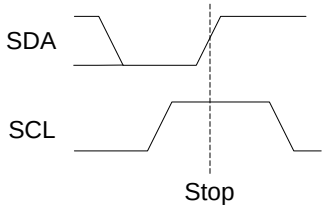

Figure 4. I2C STOP condition

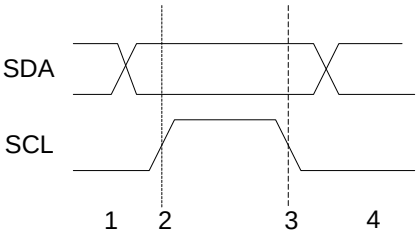

Figure 5. I2C data read / write sequence

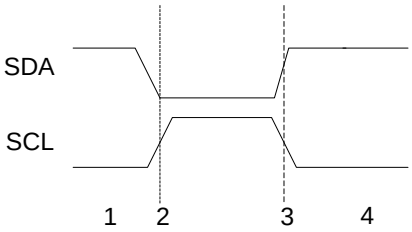

Figure 6. I2C acknowledge sequence

The acknowledge (ACK) bit is generated by the SDA being in the dominant state when the SCL is asserted in the recessive state during the ninth clock cycle. The ACK always follows the eight data bits. The slave device must always ACK the first byte in and I2C message which contains the device identification and the R/W bit. The master device must pull the SDA line low to generate an ACK when receiving data from the slave device.

No ACK is generated if the SDA line is not pulled to the dominant state by either the master or the slave during the ninth clock pulse. There are two uses of a no ACK condition. When a master intends to terminate a read sequence, it will not generate an ACK. This is called a NACK condition and does not constitute an error condition. The second use of a *NO ACK* is when the slave fails to pull the SDA to the dominant state during the ninth SCL pulse. This is an error condition and is detectible by the master. Possible reasons for a *NO ACK* condition are: the slave is not connected to the I2C lines, the incorrect device address was sent by the master, or that the slave device is not functional or “busy”.

Communications Networks

Fundamentally, a communications network is the connection of multiple devices to exchange information. A network system can consist of one device talking to many listeners (one-to-many) or many devices talking to one listener (many-to-one). A many-to-one system is characteristic of master-slave operations. A one-to-many is characteristic of a broadcast based network. Peer-to-peer networks have many devices talking and listening making a many-to-many network system. Most often, network systems share a common resource – the media over which they communicate. If more than one sender exists on a network, there must be a mechanism to detect and arbitrate conflicts when two devices attempt to transmit at the same time. Master-slave networks with a single master do not require any form of arbitration.

Master-Slave I2C Network Architecture

Figure 7 show a conventional I2C network connection between the master and one or more slave devices. The architecture is classified as a bus. All devices on the network share the same physical communications medium.

Slower slave devices can synchronize high speed masters by asserting control over the SCL signal. The master generates its own clock on the SCL line to transfer messages on the I2C-bus. Data is only valid during the HIGH period of the clock. A well-defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

I2C clock synchronization is performed using the wired-AND connection of I2C interfaces to the SCL line. As soon as the master asserts the SCL line in the recessive state, a slave device that wants to slow the master down simply holds the SCL line in the dominant state until the slave determines when to release the SCL line to the recessive state. Meanwhile, the master always reads of the SCL IO pin and doesn't start the I2C clock cycle until it detects the SCL line in the recessive state. The slowest device on the I2C network dictates the maximum data transfer rate.

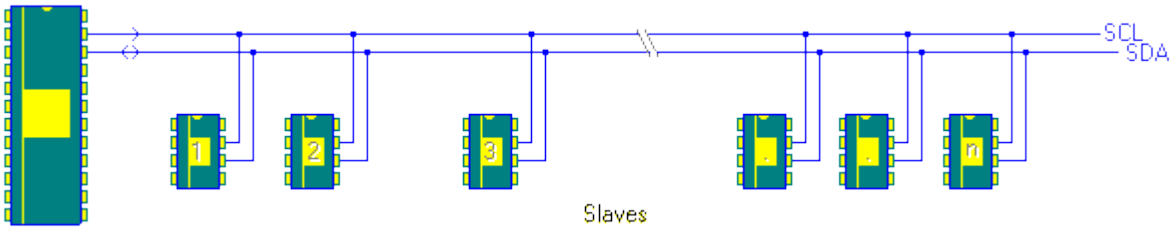

Figure 7. Single master I2C network architecture

Multi Master I2C Network

I2C technology allows for multiple masters to operate on a single I2C network as shown in Figure 8. As stated previously, the controlling master asserts the state of the SCL line. Hence, the master that is ***not*** controlling the SCL line must behave like a slave device. Each master must be able to implement an arbitration scheme that dictates that if two devices start to

communicate at the same time, the one writing more zeros (dominant bits) to the bus wins the arbitration and the other master device immediately discontinues any operation on the bus. The second requirement is that each master device must be able to detect when the network is in use by another master. Each device must detect an ongoing bus communication and must not interfere with it. This is achieved by recognizing traffic and waiting for a stop condition to appear before starting to transmit on the bus. Figure 9 shows a timing diagram where two master devices attempt to simultaneously access the same slave device. The SCL signals will be automatically synchronized according to the process described above for slow slave devices. A bus conflict will be detected by the master device that attempts to send a recessive bit but detects the SDA line in the dominant state as shown for CPU2. At this point, CPU2 places its SCL and SDA outputs in the recessive state and continues to monitor the SCL line. After there has been no activity on the SCL line for a predetermined length of time, CPU2 will attempt to assume control of the I2C bus once again.

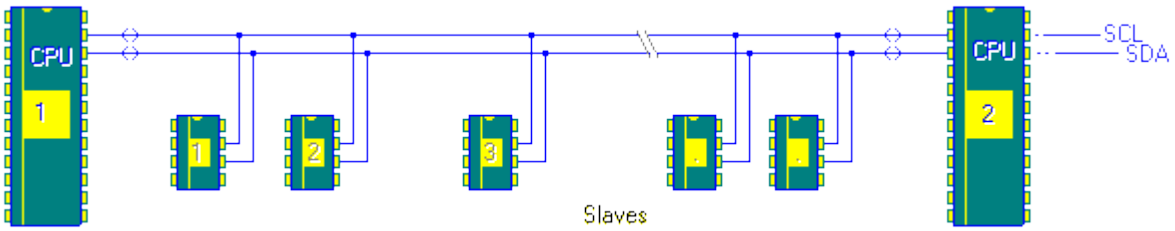

Figure 8. Multiple master I2C network architecture

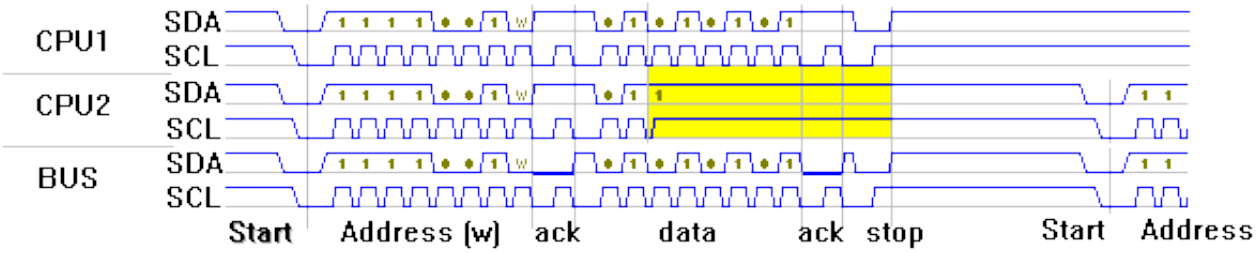

Figure 9. Clock arbitration for dual master I^2C operation

EEPROMs

EEPROM stands for electrically erasable programmable read only memory. It is a form of non-volatile memory used in many computer based systems (including PCs) to store configuration settings and other data that must be remembered from one power up cycle to the next, as the data is retained even if the power has been removed from the device. In this project we will be interfacing with the 24LC256 EEPROM. The value 256 in the part name has the significance

that the device is capable of storing 256Kbits of data organized as 32768 bytes. Pay special attention to Section 6 of the 24LC256 data sheet before writing the code to write data to the EEPROM and in particular the control flow diagram in Figure 7 of that data sheet. This data sheet should be studied closely before continuing on with project.

Fundamentals of Drivers

Drivers abstract (or "hide") the low-level details required to interface user software to peripheral devices. With a little effort, code that you have already developed could be packaged as drivers. Examples of possible drivers are: the software delay function, the stepper motor step FSM, the LCD interface, and even the button decoding function. For this project you will generate code in a file called "I2C_EEPROM_LIB.c" that provides multiple byte read and write access to the 24LC256 EEPROM.

PIC32 I2C Programming

Microchip provides example code for a PIC32 I^2C interface with a 24LC256 EEPROM as well as the application note, AN735, concerning I2C communications using PIC processors (not PIC32). The PIC32 I2C example code can be used as a reference for this project, but may be difficult to follow. You will be required to create a device driver for the 24LC256 serial EEPROM according to the driver specifications below. The Microchip example code uses the PIC32 I2C1 port whereas the Cerebot MX7cK uses the PIC32 I2C2 port to interface to the on board 24LC256 EEPROM. Note that the example code writes less than one page of data to the EEPROM starting at a page boundary. The driver that you will produce will not be so constrained in that you will be able to start at any address and store any amount of data up to the capacity of the EEPROM (32K bytes).

Project Tasks

This project requires you to create a library file to provide read and write access to the 24LC245 EEPROM. To test the library functions, you will create an additional source file that uses your library to write a block of data to the EEPROM and read the block of data back from the EEPROM. The program should then compare the two blocks of data to check that each byte written to the EEPROM matches each byte read from the EEPROM. You will report the success or failure of the write / read operation on the LCD. The specifications of the functions in the I2C_EEPROM_LIB.c file that you will generate the functions that are described below.

Project Specifications

I. EEPROM Device driver Specifications:

1. void init_I2C2(int SCK_FREQ) - Initializes the I2C2 port for the requested speed.

2. int I2CReadEEPROM(int SlaveAddress, int mem_addr, char *i2cData, int len) - Reads *len* number of bytes from the slave device with device ID *SlaveAddress* starting at memory location *mem_addr* into byte buffer **i2cData* and returns a value of 0 if no error was detected or a non zero value if an I2C error was detected.

3. int I2CWriteEEPROM(int SlaveAddress, int mem_addr, char *i2cData, int len) - Writes *len* number of bytes to the slave device with device ID *SlaveAddress* starting at memory location *mem_addr* from byte buffer **i2cData* and returns a value of 0 if no error was detected or a non zero value if an I^2C error was detected.

4. int wait_i2c_xfer(int SlaveAddress) - Used by I2CWriteEEPROM to determine if the device has completed the internal page write. This is a blocking function until the slave device has completed the operation. Use of the return value depends upon implementation.[1]

II. Generate a test application that will write a block of data of arbitrary size to the EEPROM at an arbitrary EEPROM memory location and verify that the data has been correctly stored in the EEPROM. The results of "*Test passed*" or "*Test failed*" will be displayed on the LCD.

Project Testing

1. Run the test application and verifies that every byte of data read from the EEPROM matches the data written to the EEPROM.

2. Connect an oscilloscope or logic analyzer to the PIC32 SCL and SDA lines. Capture the waveforms for an I2C transaction that reads 2 bytes from the EEPROM. Refer to Appendix A for the location of the SDA and SCL test points.

[1] For example, a non-zero return value might indicate a timeout error if the EEPROM doesn't respond with an ACK within a set time period.

Appendix A: Project 8 Parts Configuration

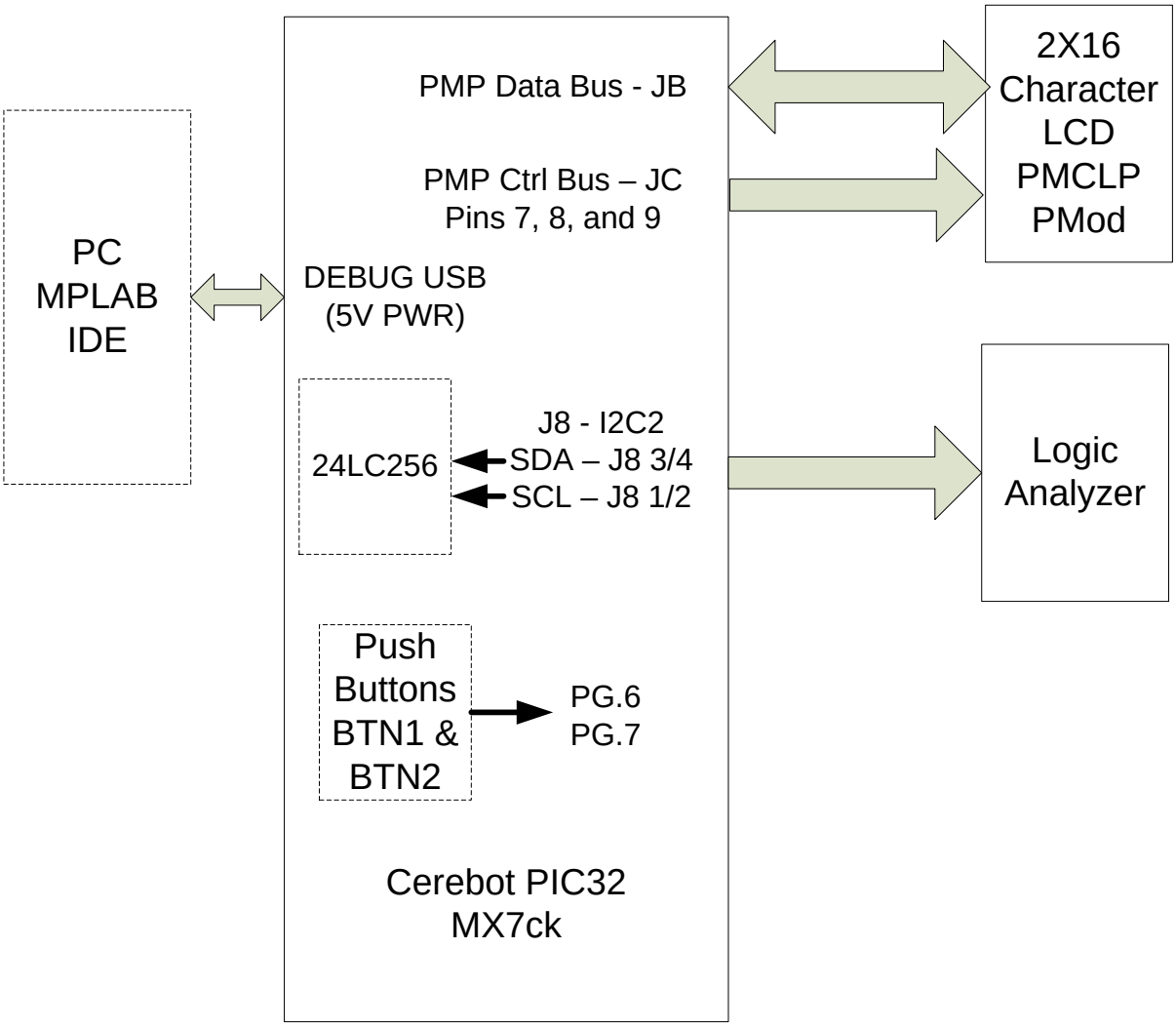

Figure 10. Block diagram of the equipment used in Project 8.

Appendix B: Serial Protocols

- Morse code telegraphy
- RS-232, RS-422, RS-423, RS-485 Asynchronous communications (Project 7)
- CAN Controller Area Network
- I²C Inter-Integrated Circuit
- SPI Serial Peripheral Interface
- SDLC/HDLC High Level Data Link Control (Traffic Control)

- 1-wire i-Button
- ARINC 818 Avionics Digital Video Bus
- USB Universal Serial Bus (moderate-speed, for connecting peripherals to computers)
- FireWire IEEE 1394
- Ethernet IEEE 802.3
- Fiber Channel (high-speed, for connecting computers to mass storage devices)
- InfiniBand (very high speed, broadly comparable in scope to PCI)
- MIDI control of electronic musical instruments
- DMX512 control of theatrical lighting
- SDI-12 industrial sensor protocol
- SpaceWire Spacecraft communication network