

Notes on the RWM Shield PCB

With the silkscreened Olin logo at the bottom of the board, you will notice several silkscreened component outlines on the left side of the board. These include a column of three chips in dual in-line packages (DIPs), one 8-pin and two 14-pin (these are the same kind of packages the op-amps and instrumentation amplifiers that you have used come in), five small bypass capacitors (0.1 μF ceramic), and one chip in a TO-92 package (i.e., a D-shaped package with three leads sticking out of the bottom, which is commonly used for discrete transistors such as the 2N3904 or the 2N7000). The 8-pin DIP is for a 25LC1024, which is a 1-Mb serial electronically erasable programmable read-only memory (EEPROM). The (upside down) 14-pin DIP right above the 8-pin one is for an MCP3302, which is a quad single-ended 12-bit/dual differential 13-bit analog-to-digital converter (ADC). The (right side up) 14-pin DIP at the top of the column is for an MCP4922, which is a dual 12-bit digital-to-analog converter (DAC). For each of these DIP footprints, the location of pin 1 is denoted by a square pad. Moreover, the top of each chip is denoted by a notch in the silkscreened component outline. **In case you missed it a couple of sentences ago, please note that the MCP3302 is upside down; if you solder it in the wrong way, it will be exceedingly difficult for you to desolder it.** All three of these chips communicate with the Arduino over the serial peripheral interface (SPI) bus, which consumes digital pins 7 through 13 of the Arduino. Digital pins 0 and 1 are used by the serial interface, which leaves digital pins 2 through 6 available for other uses (e.g., status LEDs). The integrated circuit in the TO-92 package is not a transistor, but rather it is an MCP1541, which is a precision 4.096-V voltage reference. By using such a voltage reference, a 12-bit ADC/DAC will have a conversion gain of precisely 1 mV/step.

To the right of these components, there is a prototyping area for you to use as needed for the sensor interface circuitry that you will need to implement for your experiment. In the middle of this area is a section for DIPs that is arranged like the solderless breadboards that you have been using all semester. On either side of this DIP area is an arrangement of holes and traces that is meant for components like diodes, resistors, capacitors, transistors, pushbuttons, LEDs, etc. In between each section is a single power distribution bus. These power distribution busses are not connected to anything by default. Near the top of each one, you will find a ground tie point. Near the bottom of each, you will find a tie point for +5 V. To configure a power bus, you will need to bridge from the bus to one of these tie points with a little blob of solder or by soldering in a small jumper wire.

To populate the board, we suggest the following approach. First, solder in the male header pins that will mate with the female headers on the Arduino. You can use an unpopulated board constrain the bottoms of the header pins so that they will be vertical as you solder them at the top of the board that you are populating. Next, solder in the DIPs and strip socket(s). Next, solder in the capacitors. Finally, solder in the MCP1541. To solder in the capacitors and the MCP1541, it

is helpful to splay out the leads after you insert them through the holes. That way, when you turn the board over to make the solder joints, the components won't fall back through the board. When you solder, remember to heat the pad and component pin with the iron, not the solder. Then bring the solder in, touching the hot pad and component lead. You will need to work relatively quickly, applying the soldering iron tip to the component lead for no more than a second or two. You should try to clean the tip of the soldering iron once every four to eight solder connections on a damp solder sponge. Once you have soldered the component leads, you should clip the leads flush to the board with diagonal/flush cutters. You should also clip off the portions of pins 4 through 7 of the MCP4922 that protrude through the bottom of the board, so that they don't get shorted by the USB connector on the Arduino board when you plug the shield into the Arduino.

Notes on the Software/Firmware

If you haven't already done so, the first thing you should do to install the software for this lab is to obtain the latest version of the Arduino IDE from the Arduino website (<http://arduino.cc>). When you run the Arduino software for the first time, it will create a sketch folder called `Arduino` in your `My Documents` folder (on Windows). Next, obtain the `rwmarduinolab.zip` file from the course web site and unzip it in some convenient location. Inside it, you should find a folder called `Arduino`, which contains several subfolders. You should move these subfolders to the sketch folder that the Arduino IDE created. These folders contain an Arduino library for communicating with the three chips on the shield that you have built over the SPI bus and a sketch called `data_logger` that will orchestrate your data logging experiment. The `data_logger` sketch is set up to log data from one single-ended channel (A10). Depending on what you are doing, you might need to modify the `logdata` and `dump` functions in the sketch to perform the experiment that you intend to execute. In the zip file, you should also find a folder called `MATLAB`, which contains a single Matlab script, called `rwmcatch`, which receives the logged data from the Arduino over a virtual serial port.

The basic procedure to log data with the `data_logger` sketch and the RWM shield is as follows:

1. Plug the Arduino/RWM shield into a USB port on your computer and upload the `data_logger` sketch to the Arduino with any changes to the `logdata` and `dump` functions necessary for your experiment. Open the serial monitor (Tools > Serial Monitor) and set the baud rate to 115,200.
2. To check the current sampling interval value, type `interval?`; in the text entry field at the top of the serial monitor; the value will be displayed in the serial monitor. To set the value, type `interval <val>`; in the text entry. The sampling interval must be a positive integer in milliseconds.

3. To clear the EEPROM and prepare the Arduino/RWM shield to log data the next time the Arduino is powered on or reset, type `log;` into the entry field of the serial monitor.
4. Unplug the Arduino from the USB port. When you are ready to begin logging data, either power on the Arduino by plugging a 9-V battery into the barrel power connector on the Arduino board (instead of a battery, you could also use a wall wart or USB power) or reset the Arduino if it was not being powered by USB when it was plugged in. The Arduino will log data until it is powered off or until the EEPROM fills up.
5. Finally, plug the Arduino into a USB port on your computer, launch Matlab and, run the `rwmcatch` script. The logged data should appear in your Matlab workspace as a matrix called `data`. The first column of data is time (in ms). The other columns are the various channels of voltage that you logged (in mV).

You will probably have to edit the `rwmcatch` script to make the name of the virtual serial port opened in the first line match the one the operating system assigns to the Arduino board when you plug it into a USB port on your computer. When you run the `rwmcatch` script, you should see a progression of dots appear indicating that data is being transferred from the Arduino. If you don't see the first dot right away, you may get an error indicating that the transmission has timed out. If this happens, we suggest that you restart Matlab. It seems that, if you start Matlab before you plug the Arduino in to the USB port, Matlab doesn't get the message that the device has been plugged in.

In addition to the serial commands mentioned above, the `datalogger` sketch accepts a number of other commands via the serial monitor, some of which you might find some of these useful for testing your RWM shield. They are as follows:

Command	Effect
<code>ao0 <val>;</code>	Sets the value of analog output 0.
<code>ao0?;</code>	Displays the last value sent to analog output 0.
<code>ao1 <val>;</code>	Sets the value of analog output 1.
<code>ao1?;</code>	Displays the last value sent to analog output 1.
<code>ai0?;</code>	Displays the current value of analog input 0.
<code>ai1?;</code>	Displays the current value of analog input 1.
<code>ai2?;</code>	Displays the current value of analog input 2.
<code>ai3?;</code>	Displays the current value of analog input 3.
<code>aid0?;</code>	Displays the current differential value between analog input 0 and analog input 1.
<code>aid1?;</code>	Displays the current differential value between analog input 2 and analog input 3.
<code>clear;</code>	Erases all of the values in the EEPROM (to 0xFF).

<code>address <val>;</code>	Sets the address for the EEPROM.
<code>address?;</code>	Displays the current value of the address for the EEPROM.
<code>writebyte <val>;</code>	Writes value to the EEPROM at the current address.
<code>readbyte?;</code>	Displays the value stored in the EEPROM at the current address
<code>readbytes <num>;</code>	Displays a number of consecutive values stored in the EEPROM starting at the current address.

Methods Provided by the RWM Arduino Library

The RWM Arduino library that we have provided has a number of functions/methods in it that you can use in modifying the `data_logger` sketch or in your own sketches to interact with the DAC, ADC, and EEPROM chips on the RWM shield. These functions are described below.

void RWM::DACwriteChannel(byte ch, int value)

This function produces a voltage proportional to value on the analog output channel indicated by ch. The ch parameter can be 0 or 1. The value parameter can be between 0 and 4095 and is in units of millivolts. If either the ch parameter or the value parameter is out of range, the function does nothing.

void RWM::DACwriteChannels(int value0, int value1)

This function produces a voltage proportional to value0 on the AO0 and a voltage proportional to value1 on AO1. The value0 and value1 parameters can be between 0 and 4095 and are in units of millivolts. If either parameter value is out of range, the function does nothing.

int RWM::ADCreadChannel(byte ch)

This function reads the analog input channel indicated by the ch parameter and returns the result as an integer between 0 and 4095. This integer reflects the value of the voltage read on the specified analog input channel in units of millivolts. The ch parameter can be between 0 and 3. If the ch parameter is out of range, the function returns -32,768.

int RWM::ADCreadChannelDiff(byte ch)

This function reads the pair of analog input channels (either AI0/AI1 or AI2/AI3) indicated by the ch parameter differentially and returns the result as an integer between -4096 and 4095. This integer reflects the value of the differential voltage read on the specified analog input channel in units of millivolts. The ch parameter can be 0 (indicating AI0/AI1) or 1 (indicating AI2/AI3). If the ch parameter is out of range, the function returns -32,768.

byte RWM::EEPROMreadStatus(void)

This function reads the value of the EEPROM's internal status register and returns it as a byte. The least significant bit of the status register indicates that a write is in progress. For details about the meaning of the other bits in the return value, see the 24LC1024 datasheet.

void RWM::EEPROMwriteEnable(void)

This function enables the EEPROM for writing.

void RWM::EEPROMchipErase(void)

This function issues the command to erase the EEPROM. Before this function is called, the EEPROM must first have been enabled for writing with a call to the EEPROMwriteEnable function.

void RWM::EEPROMwriteByte(unsigned long address, byte value)

This function writes the value indicated by the value parameter to the EEPROM in the location indicated by the address parameter. The address parameter can be between 0 (0x00000) and 131,071 (0x1FFFF). If the address parameter falls outside of this range, the function does not attempt to write to the EEPROM.

void RWM::EEPROMwriteBytes(unsigned long address, byte num, byte *buffer)

This function writes num bytes contained in the array pointed to by buffer to the EEPROM starting at the location indicated by the address parameter. The address parameter can be between 0 (0x00000) and 131,071 (0x1FFFF), but the entire array must fit inside this address range. If the array would fall outside of the allowed range, the function does not attempt to write to the EEPROM. Care should be taken to ensure that the array being written does not cross a page boundary (pages are 256 bytes long), otherwise the address will wrap around at the page boundary and the data will not be written to the EEPROM contiguously.

byte RWM::EEPROMreadByte(unsigned long address)

This function reads the value stored in the EEPROM at the location specified by the address parameter and returns it as a byte. The address parameter can be between 0 (0x00000) and 131,071 (0x1FFFF). If the address parameter falls outside of this range, the function does not attempt to read from the EEPROM, but instead returns a value of 255 (0xFF).

void RWM::EEPROMwriteInt(unsigned long address, int value)

This function writes the value indicated by the value parameter to the EEPROM in the location indicated by the address parameter. The address parameter can be between 0 (0x0000) and 65,533 (0xFFFFD). If the address parameter falls outside of this range, the function does not attempt to write to the EEPROM. The value indicated by the value parameter is stored in the EEPROM as two successive bytes (little endian) starting at the location given by twice the value of the address parameter.

void RWM::EEPROMwriteInts(unsigned long address, byte num, int *buffer)

This function writes num integers contained in the array pointed to by buffer to the EEPROM starting at the location indicated by the address parameter. The address parameter can be between 0 (0x00000) and 65,533 (0xFFFFD), but the entire array must fit inside this address range. If the array would fall outside of the allowed range, the function does not attempt to write to the EEPROM. Care should be taken to ensure that the array being written does not cross a page boundary (pages are 256 bytes long), otherwise the address will wrap around at the page boundary and the data will not be written to the EEPROM contiguously. The integer values are stored in the EEPROM as two successive bytes (little endian) starting at the location given by twice the value of the address parameter.

int RWM::EEPROMreadInt(unsigned long address)

This function reads the integer value stored in the EEPROM at the location specified by the address parameter and returns it as an int. The address parameter can be between 0 (0x0000) and 65,533 (0xFFFFD). If the address parameter falls outside of this range, the function does not attempt to read from the EEPROM, but instead returns -32,768 (0xFFFF). The integer value is stored in the EEPROM as two successive bytes (little endian) starting at the location given by twice the value of the address parameter.

void RWM::EEPROMwrite2Ints(unsigned long address, int *buffer)

This function writes two integer values contained in an array pointed to by the buffer parameter to the EEPROM in the location indicated by the address parameter. The address parameter can have a value between 0 (0x0000) and 32,766 (0x7FFE). If the address parameter falls outside this range, the function does not attempt to write to the EEPROM. The values are stored in the EEPROM as four successive bytes (each of the integer values is stored little endian), starting at the location given by four times the value of the address parameter.

void RWM::EEPROMread2Ints(unsigned long address, int *buffer)

This function reads two integer values stored in the EEPROM at the location specified by the address parameter and returns them in the integer array pointed to by the buffer parameter. The address parameter can have a value between 0 (0x0000) and 32,766 (0x7FFE). If the address parameter falls outside this range, the function does not attempt to read from the EEPROM, but instead returns values of -32,768 (0xFFFF) in the integer array pointed to by the buffer parameter.

void RWM::EEPROMwrite4Ints(unsigned long address, int *buffer)

This function writes four integer values contained in an array pointed to by the buffer parameter to the EEPROM in the location indicated by the address parameter. The address parameter can have a value between 0 (0x0000) and 16,382 (0x3FFE). If the address parameter falls outside this range, the function does not attempt to write to the EEPROM. The values are stored in the EEPROM as eight successive bytes (each of the integer values is stored little endian), starting at the location given by eight times the value of the address parameter.

void RWM::EEPROMread4Ints(unsigned long address, int *buffer)

This function reads four integer values stored in the EEPROM at the location specified by the address parameter and returns them in the integer array pointed to by the buffer parameter. The address parameter can have a value between 0 (0x0000) and 16,382 (0x3FFE). If the address parameter falls outside this range, the function does not attempt to read from the EEPROM, but instead returns values of -32,768 (0xFFFF) in the integer array pointed to by the buffer parameter.