ENG20009
Engineering Technology Inquiry Project

Unit Convenor : Dr Rifai Chai

Email: rchai@swin.edu.au

Phone: 9214 8119

Office: EN606B

Swinburne University of Technology

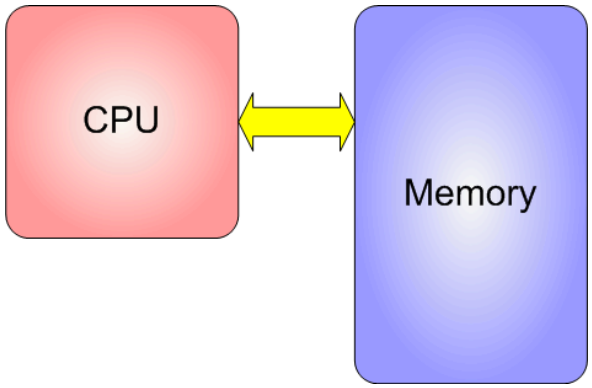
Lecture 5 – Memory

Topics:

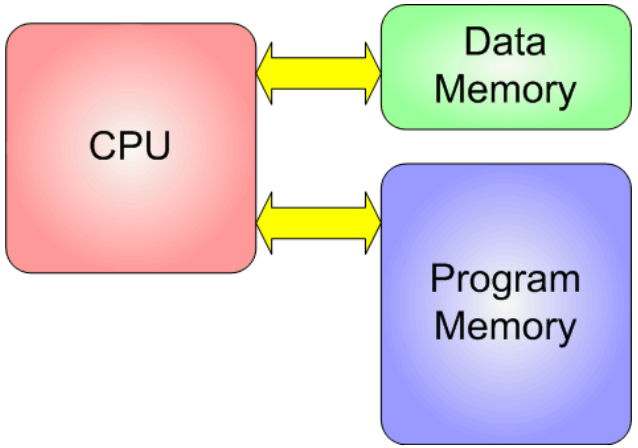
1. Memory Architecture
2. Different Memories on Arduino
3. Memory Handling
4. Internal EEPROM
5. External EEPROM
6. SD Card

1. Memory Architecture

- The Von Neumann architecture uses the same memory and data paths for both program and data storage.
- The Harvard architecture used physically separate memory and data paths for program and memory.
- The Harvard model turns out to be a good match for embedded applications such as microcontroller used in the Arduino UNO use a relatively pure Harvard architecture. Programs are stored in Flash memory and data is stored in SRAM.



Von Neumann architecture



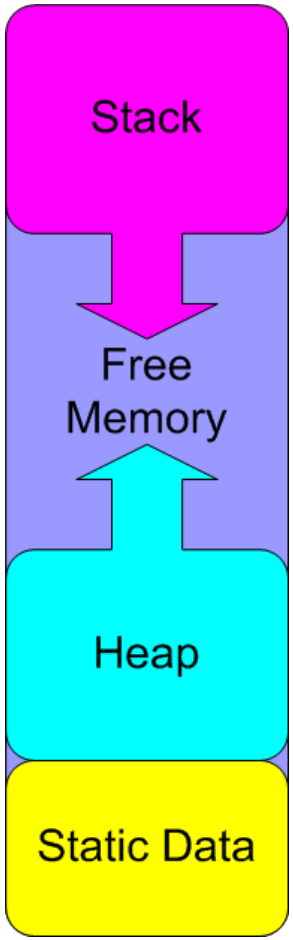
Harvard architecture

1. Different Memories on Arduino

- There are 3 types of memory in an Arduino:
  1. Flash or Program Memory
  2. Static Random Access Memory (SRAM)
  3. Electrically Erasable Programmable Read-Only Memory (EEPROM).
- Flash memory is used to store your program image and any initialized data. You can execute program code from flash, but you can't modify data in flash memory from your executing code. To modify the data, it must first be copied into SRAM
- Flash memory is the same technology used for thumb-drives and SD cards. It is non-volatile, so your program will still be there when the system is powered off.
- Flash memory has a finite lifetime of about 100,000 write cycles.

2. Different Memories on Arduino

- SRAM or Static Random Access Memory, can be read and written from your executing program. SRAM memory is used for several purposes by a running program



2. Different Memories on Arduino

- Static Data - This is a block of reserved space in SRAM for all the global and static variables from your program. For variables with initial values, the runtime system copies the initial value from Flash when the program starts.
- Heap - The heap is for dynamically allocated data items. The heap grows from the top of the static data area up as data items are allocated.
- Stack - The stack is for local variables and for maintaining a record of interrupts and function calls. The stack grows from the top of memory down towards the heap. Every interrupt, function call and/or local variable allocation causes the stack to grow. Returning from an interrupt or function call will reclaim all stack space used by that interrupt or function.

2. Different Memories on Arduino

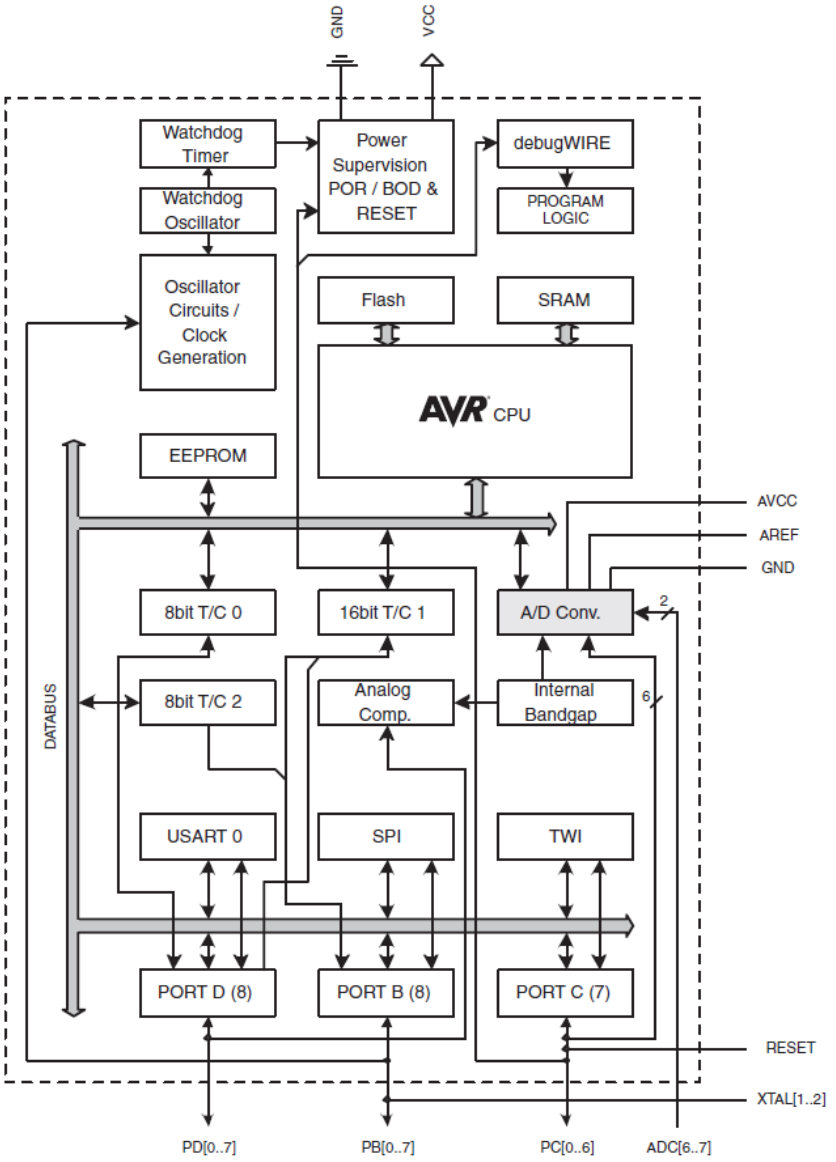
- Most memory problems occur when the stack and the heap collide. When this happens, one or both of these memory areas will be corrupted with unpredictable results. In some cases it will cause an immediate crash. In others, the effects of the corruption may not be noticed until much later.
- EEPROM is another form of non-volatile memory that can be read or written from your executing program. It can only be read byte-by-byte, so it can be a little awkward to use. It is also slower than SRAM and has a finite lifetime of about 100,000 write cycles (you can read it as many times as you want).

2. Different Memories on Arduino

Model	SRAM (KB)	EEPROM (KB)	Program (KB)
UNO Rev 3	2	1	32
Mega 2560 Rev 3	**8**	**4**	**256**
Nano	2	1	32
Due	**96**	**0**	**512**
Leonardo	2.5	1	32
Micro	2.5	1	32
UNO WiFi Rev 2	6	0.25	48
MKR WiFi 1010	**32**	**0**	**256**
Zero	**32**	**0**	**256**

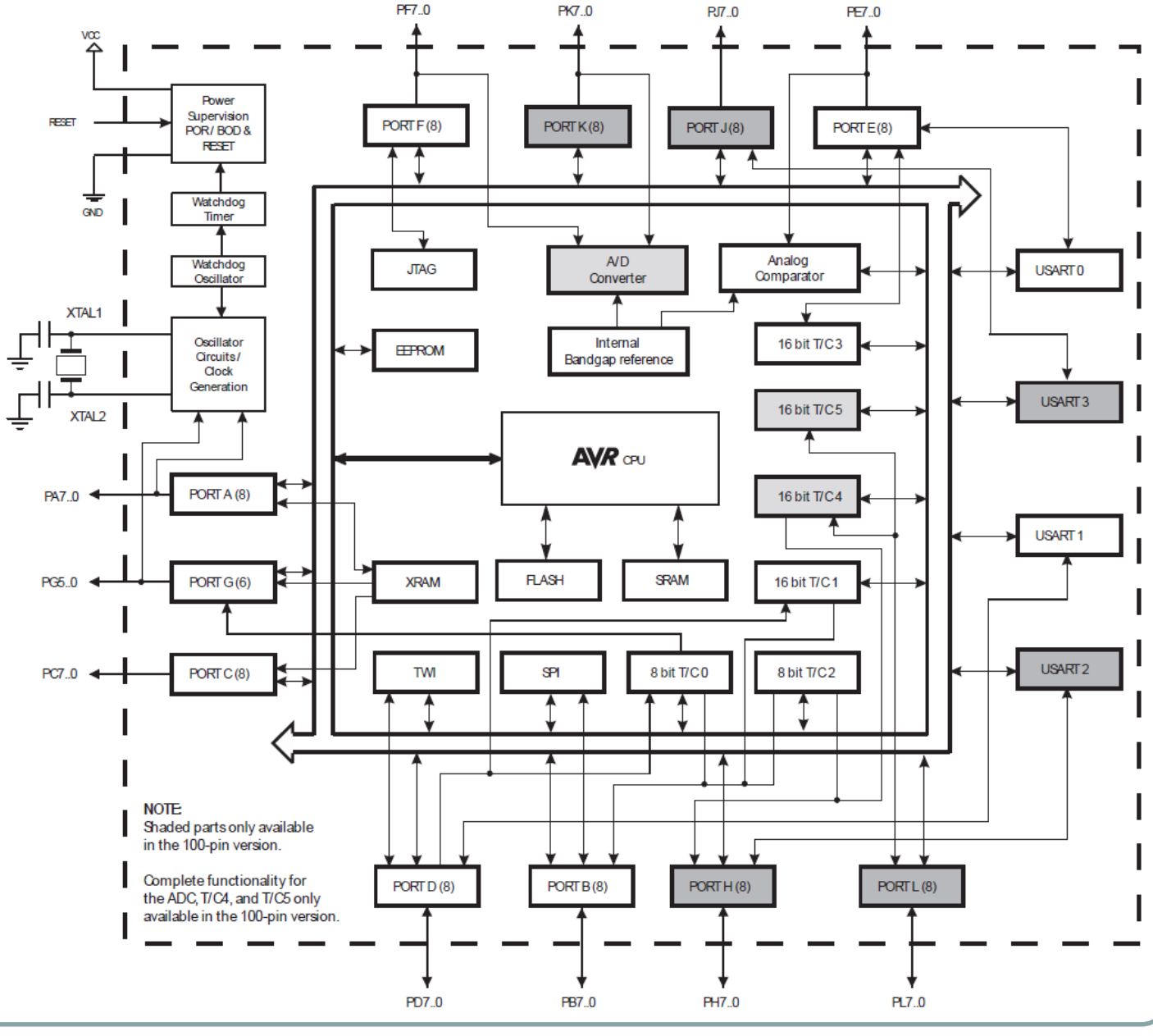
2. Different Memories on Arduino – Arduino Uno

TheATmega48A/PA/88A/PA/168A/PA/328/P is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture



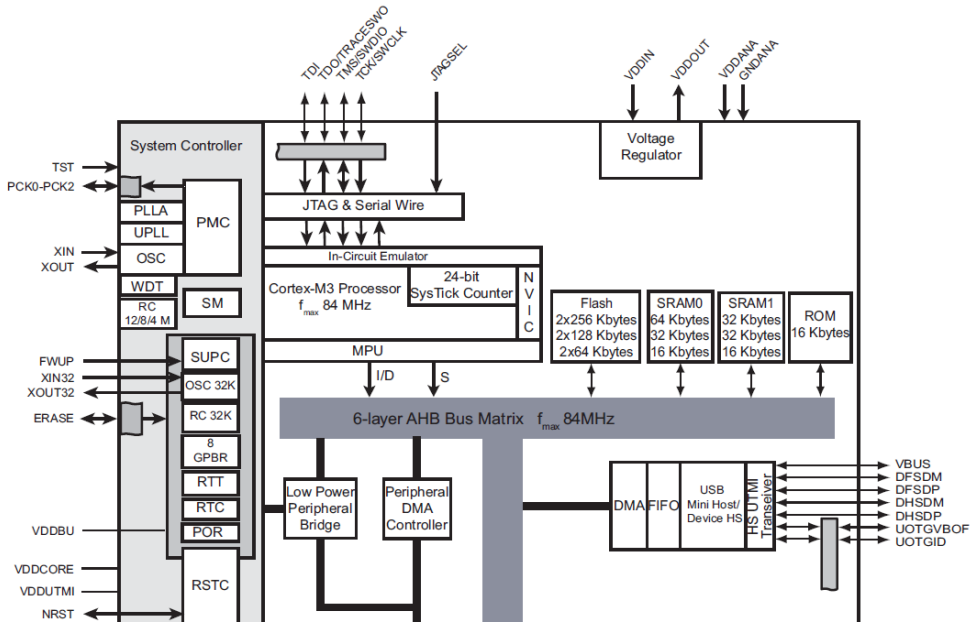
2. Different Memories on Arduino – Arduino Mega

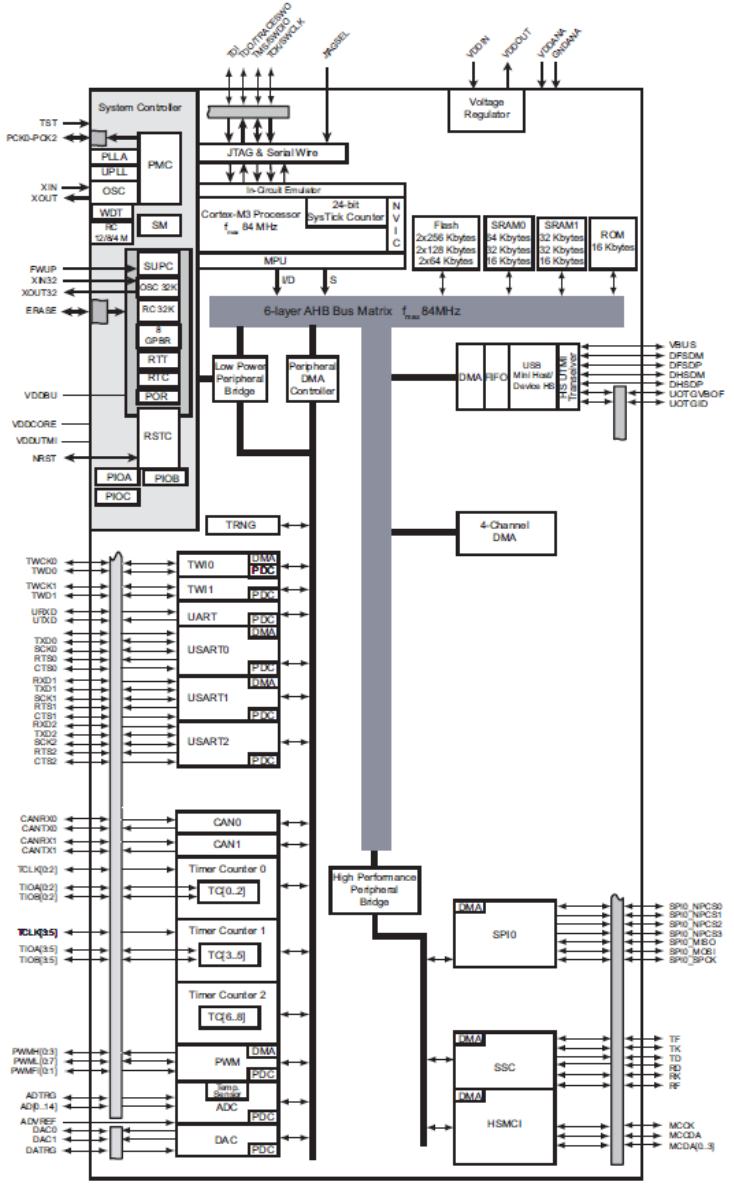
ATmega640/1280/1281/2560/2561 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture



2. Different Memories on Arduino – Arduino Due

SAM3A4/8C



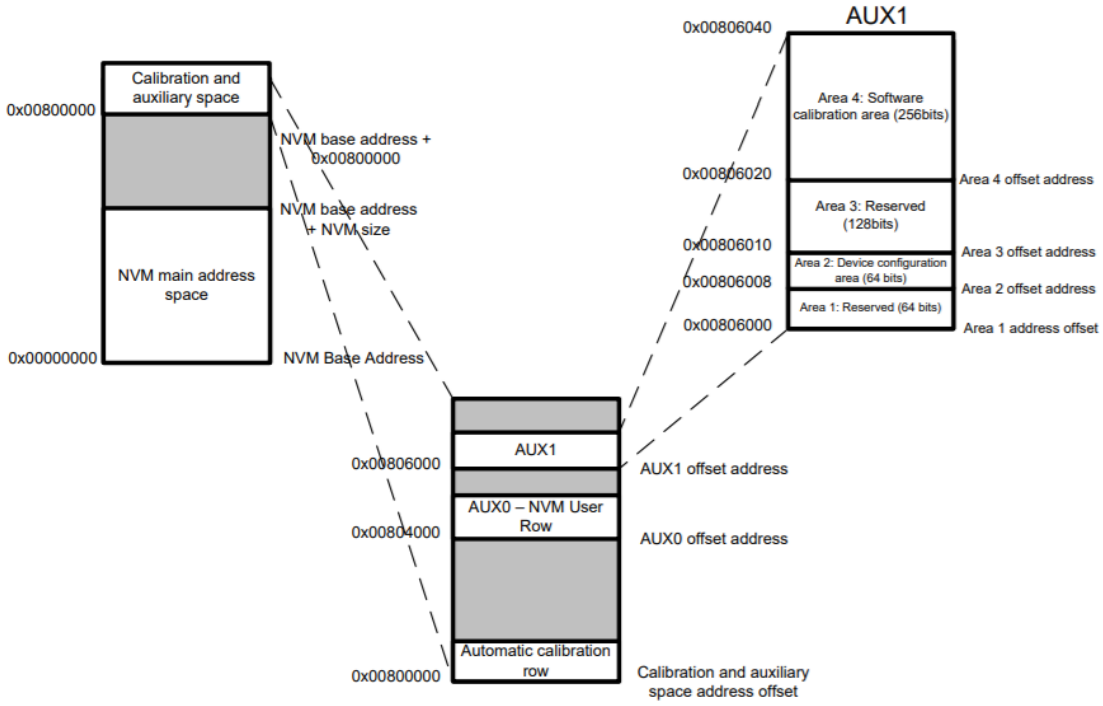


2. Different Memories on Arduino – Arduino Zero

SAM D21 Physical Memory Map

Memory	Start address	Size					
		SAMD21x18	SAMD21x17	SAMD21x16	SAMD21x15	SAMD21x16L	SAMD21x15L
Internal Flash	0x00000000	256 Kbytes	128 Kbytes	64 Kbytes	32 Kbytes	64 Kbytes	32 Kbytes
Internal RWW section	0x00400000	-	-	2 Kbytes	1 Kbytes	2 Kbytes	1 Kbytes
Internal SRAM	0x20000000	32 Kbytes	16 Kbytes	8 Kbytes	4 Kbytes	8 Kbytes	4 Kbytes
Peripheral Bridge A	0x40000000	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes
Peripheral Bridge B	0x41000000	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes
Peripheral Bridge C	0x42000000	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes
IOBUS	0x60000000	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes

Calibration and Auxiliary Space



3. Memory Handling

- The Arduino build process was designed to hide complex aspects of C and C++, as well as the tools used to convert a sketch into the bytes that are uploaded and run on an Arduino board. But if your project has performance and resource requirements beyond the capability of the standard Arduino environment, you should find the way to access including the memory handling.

- Program memory (also known as flash) is where the executable sketch code is stored. The contents of program memory can only be changed by the bootloader in the upload process initiated by the Arduino software running on your computer.

3. Memory Handling

- After the upload process is completed, the memory cannot be changed until the next upload.
- There is far more program memory on an Arduino board than RAM, so it can be beneficial to store values that don't change while the code runs (e.g., constants) in program memory.
- The bootloader takes up some space in program memory. If all other attempts to minimize the code to fit in program memory have failed, the bootloader can be removed to free up space, but an additional hardware programmer is then needed to get code onto the board.
- If your code is larger than the program memory space available on the chip, the upload will not work and the IDE will warn you that the sketch is too big when you compile.

3. Memory Handling

- RAM is used by the code as it runs to store the values for the variables used by your sketch (including variables in the libraries used by your sketch). RAM is volatile, which means it can be changed by code in your sketch. It also means anything stored in this memory is lost when power is switched off.
- Arduino has much less RAM than program memory. If you run out of RAM while your sketch runs on the board (as variables are created and destroyed while the code runs) the board will misbehave (crash).
- EEPROM is memory that code running on Arduino can read and write, but it is non-volatile memory that retains values even when power is switched off. EEPROM access is significantly slower than for RAM, so EEPROM is usually used to store configuration or other data that is read at startup to restore information from the previous session.

3. Memory Handling

- Preprocessor: Preprocessing is a step in the first stage of the build process in which the source code (your sketch) is prepared for compiling. Various find and replace functions can be performed.
- Preprocessor commands are identified by lines that start with #. You have already seen them in sketches that use a library—#include tells the preprocessor to insert the code from the named library file.
- Sometimes the preprocessor is the only way to achieve what is needed, but its syntax is different from C and C++ code, and it can introduce bugs that are subtle and hard to track down, so use it with care.

3. Memory Handling

- AVRfreaks is a website for software engineers that is a good source for technical detail on the controller chips used by Arduino: http://www.avrfreaks.net

- Technical details on the C preprocessor are available at https://gcc.gnu.org/onlinedocs/cpp.pdf

- The memory specifications for all of the official boards can be found on the Arduino website. https://www.arduino.cc/en/Main/Products?from=Main.Hardware

3. Understanding the Arduino Build Process

- If you want to see what is happening under the covers when you compile and upload a sketch, you can choose to display all the command-line activity that takes place when compiling or uploading a sketch through the Preferences dialog added in Arduino 1.0. Select File→Preferences to display the dialog box to check or uncheck the boxes to enable verbose output for compile or upload messages.
- In releases earlier than 1.0, you can hold down the Shift key when you click on Compile or Upload. The console area at the bottom of the IDE will display details of the compile process.
- In releases earlier than 1.0, you need to change a value in the Arduino preferences.txt file to make this detail always visible.

3. Understanding the Arduino Build Process

- Some preferences can be controlled from the Preferences dialog within the Arduino environment. Access it from the File menu in Windows or Linux, or the Arduino menu on the Mac.
- Other preferences must be changed in the preferences.txt file. This file's location is displayed in the preferences dialog. It should be:

  - \Arduino15\preferences.txt (Windows, Arduino IDE 1.6.6 and newer)
  - \Arduino15\preferences.txt (Windows, Arduino IDE 1.6.0 - 1.6.5)
  - \Arduino\preferences.txt (Windows, Arduino IDE 1.0.6 and older)
  - \Documents\ArduinoData\preferences.txt (Windows app version)
  - ~/Library/Arduino15/preferences.txt (Max OS X, Arduino IDE 1.6.0 and newer)
  - ~/Library/Arduino/preferences.txt (Max OS X, Arduino IDE 1.0.6 and older)
  - ~/.arduino15/preferences.txt (Linux, Arduino IDE 1.6.0 and newer)
  - ~/.arduino/preferences.txt (Linux, Arduino IDE 1.0.6 and older)
  - <Arduino IDE installation folder>/portable/preferences.txt (when used in portable mode)

3. Understanding the Arduino Build Process

- Only edit the file when Arduino is not running - otherwise your changes will be overwritten when Arduino exits.
- The definitions that determine the contents of the Board menu are found in boards.txt in the hardware/ sub-directory of the Arduino application directory. The definitions for the Burn Bootloader menu are in the programmers.txt file in the same directory. To create a new board or programmer definition, copy an existing one, change the prefix used in the preference keys (e.g. "diecimila." or "avrisp."), and alter the values to fit your hardware. Note that only certain board configurations are supported.
- Make sure the Arduino IDE is not running (changes made to preferences.txt will not be saved if the IDE is running). Open the file and find the line build.verbose=false (it is near the bottom of the file). Change false to true and save the file.

3. Understanding the Arduino Build Process

- When you click on Compile or Upload, a lot of activity happens that is not usually displayed on-screen. The command-line tools that the Arduino IDE was built to hide are used to compile, link, and upload your code to the board.
- First your sketch file(s) are transformed into a file suitable for the compiler (AVRGCC) to process. All source files in the sketch folder that have no file extension are joined together to make one file. All files that end in .c or .cpp are compiled separately.
- Header files (with an .h extension) are ignored unless they are explicitly included in the files that are being joined.
- #include "Arduino.h" (WProgram.h in previous releases) is added at the top of the file to include the header file with all the Arduino-specific code definitions, such as digitalWrite() and analogRead(). If you want to examine its contents, you can find the file on Windows under the directory where Arduino was installed; from there, you can navigate to Hardware→Arduino→Cores→Arduino.

3. Understanding the Arduino Build Process

- On the Mac, Ctrl+click the Arduino application icon and select Show Package Contents from the drop-down menu. A folder will open; from the folder, navigate to Contents→ Resources→Java→Hardware→Arduino→Cores→Arduino.
- To make the code valid C++, the prototypes of any functions declared in your code are generated next and inserted.
- Finally, the setting of the board menu is used to insert values (obtained from the boards.txt file) that define various constants used for the controller chips on the selected board.
- This file is then compiled by AVR-GCC, which is included within the Arduino main download (it is in the tools folder).
- The compiler produces a number of object files (files with an extension of .o that will be combined by the link tool). These files are stored in /tmp on Mac and Linux. On Windows, they are in the applet directory (a folder below the Arduino install directory).

3. Understanding the Arduino Build Process

- The object files are then linked together to make a hex file to upload to the board. Avrdude, a utility for transferring files to the Arduino controller, is used to upload to the board.
- The tools used to implement the build process can be found in the hardware\tools directory.
- Another useful tool for experienced programmers is avr-objdump, also in the tools folder. It lets you see how the compiler turns the sketch into code that the controller chip runs.
- This tool produces a disassembly listing of your sketch that shows the object code intermixed with the source code. It can also display a memory map of all the variables used in your sketch. To use the tool, compile the sketch and navigate to the folder containing the Arduino distribution. Then, navigate to the folder with all the intermediate files used in the build process (as explained earlier).

3. Understanding the Arduino Build Process

- The file used by avr-objdump is the one with the extension .elf. For example, if you compile the Blink sketch you could view the compiled output (the machine code) by executing the following on the command line:

```
..\hardware\tools\avr\bin\avr-objdump.exe -S blink.cpp.elf
```

- It is convenient to direct the output to a file that can be read in a text editor. You can do this as follows:

```
..\hardware\tools\avr\bin\avr-objdump.exe -S blink.cpp.elf > blink.txt
```

- This version adds a list of section headers (helpful for determining memory usage):

```
..\hardware\tools\avr\bin\avr-objdump.exe -S -h blink.cpp.elf > blink.txt
```

- You can create a batch file to dump the listing into a file. Add the path of your Arduino installation to the following line and save it to a batch file:

```
hardware\tools\avr\bin\avr-objdump.exe -S -h -Tdata %1 > %1%.txt
```

3. Determining the Amount of Free and Used RAM

- You want to be sure you have not run out of RAM. A sketch will not run correctly if there is insufficient memory, and this can be difficult to detect.
- The following program shows you how you can determine the amount of free memory available to your sketch. This sketch contains a function called memoryFree that reports the amount of available RAM.
- The memoryFree function uses system variables to calculate the amount of RAM. System variables are not normally visible (they are created by the compiler to manage internal resources). It is not necessary to understand how the function works to use its output.
- The function returns the number of bytes of free memory.
- The number of bytes your code uses changes as the code runs.

3. Determining the Amount of Free and Used RAM

```
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.print(memoryFree());  // print the free memory
  Serial.print(' ');           // print a space
  delay(1000);
}


// variables created by the build process when compiling the sketch
extern int __bss_end;
extern void *__brkval;

// function to return the amount of free RAM
int memoryFree()
{
  int freeValue;
  if((int)__brkval == 0)
     freeValue = ((int)&freeValue) - ((int)&__bss_end);
  else
    freeValue = ((int)&freeValue) - ((int)__brkval);

  return freeValue;
}
```

3. Determining the Amount of Free and Used RAM

- The number of bytes your code uses changes as the code runs. The important thing is to ensure that you don't consume more memory than you have. Here are the main ways RAM memory is consumed:

  - When you initialize constants:

```
#define ERROR_MESSAGE "an error has occurred"
```

  - When you declare global variables:

```
char myMessage[] = "Hello World";
```

  - When you make a function call:

```
void myFunction(int value)
{
   int result;
   result = value * 2;
   return result;
}
```

  - When you dynamically allocate memory:

```
String stringOne = "Arduino String";
```

3. Determining the Amount of Free and Used RAM

- The Arduino String class uses dynamic memory to allocate space for strings. You can see this by adding the following line to the very top of the code in the Solution:

```
String s = "\n";
```

- and the following lines just before the delay in the loop code:

```
s = s + "Hello I am Arduino \n";
  Serial.println(s);            // print the string value
```

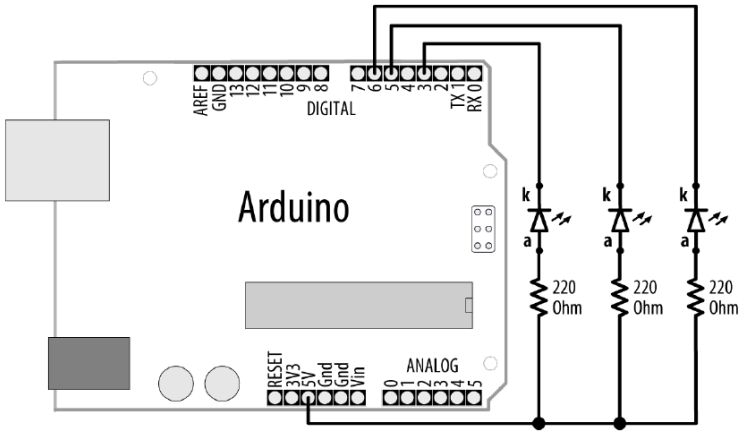
- You will see the memory value reduce as the size of the string is increased each time through the loop. If you run the sketch long enough, the memory will run out—don’t endlessly try to increase the size of a string in anything other than a test application.

3. Determining the Amount of Free and Used RAM

- Writing code like this that creates a constantly expanding value is a sure way to run out of memory. You should also be careful not to create code that dynamically creates different numbers of variables based on some parameter while the code runs, as it will be very difficult to be sure you will not exceed the memory capabilities of the board when the code runs.
- Constants and global variables are often declared in libraries as well, so you may not be aware of them, but they still use up RAM. The Serial library, for example, has a 128-byte global array that it uses for incoming serial data. This alone consumes one-eighth of the total memory of an old Arduino 168 chip.
- A technical overview of memory usage is available at http://www.gnu.org/savannah-checkouts/non-gnu/avr-libc/user-manual/malloc.html.

3. Storing and Retrieving Numeric Values in Program Memory

- If you have a lot of constant numeric data and don't want to allocate this to RAM, Store numeric variables in program memory (the flash memory used to store Arduino programs).
- The example below, this sketch adjusts a fading LED for the nonlinear sensitivity of human vision. It stores the values to use in a table of 256 values in program memory rather than RAM.
- The sketch is based on following wiring diagram on driving LEDs with results in a smooth change in brightness with the LED on pin 5 compared to the LED on pin 3:



3. Storing and Retrieving Numeric Values in Program Memory

- Program without involvement Program Memory:

```
/*
 * LedBrightness sketch
 * controls the brightness of LEDs on analog output ports
*/

const int firstLed   = 3;        // specify the pin for each of the LEDs
const int secondLed  = 5;
const int thirdLed   = 6;

int brightness = 0;
int increment = 1;

void setup()
{
  // pins driven by analogWrite do not need to be declared as outputs
}

void loop()
{
  if(brightness > 255)
  {

     increment = -1; // count down after reaching 255
  }
  else if(brightness < 1)
  {
    increment =  1; // count up after dropping back down to 0
  }
  brightness = brightness + increment; // increment (or decrement sign is minus)

  // write the brightness value to the LEDs
  analogWrite(firstLed, brightness);
  analogWrite(secondLed, brightness);
  analogWrite(thirdLed, brightness );

  delay(10); // 10ms for each step change means 2.55 secs to fade up or down
}
```

3. Storing and Retrieving Numeric Values in Program Memory

- Program with Program Memory:

```
#include <avr/pgmspace.h>  // needed for PROGMEM

// table of exponential values
// generated for values of i from 0 to 255 -> x=round( pow( 2.0, i/32.0) - 1);

const byte table[]PROGMEM = {
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   2,   2,   2,   2,   2,
   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   3,   3,   3,   3,   3,   3,
   3,   3,   3,   3,   3,   3,   4,   4,   4,   4,   4,   4,   4,   4,   4,   5,
   5,   5,   5,   5,   5,   5,   5,   6,   6,   6,   6,   6,   6,   6,   7,   7,
   7,   7,   7,   8,   8,   8,   8,   8,   9,   9,   9,   9,   9,  10,  10,  10,
  10,  11,  11,  11,  11,  12,  12,  12,  12,  13,  13,  13,  14,  14,  14,  15,
  15,  15,  16,  16,  16,  17,  17,  18,  18,  18,  19,  19,  20,  20,  21,  21,
  22,  22,  23,  23,  24,  24,  25,  25,  26,  26,  27,  28,  28,  29,  30,  30,
  31,  32,  32,  33,  34,  35,  35,  36,  37,  38,  39,  40,  40,  41,  42,  43,
  44,  45,  46,  47,  48,  49,  51,  52,  53,  54,  55,  56,  58,  59,  60,  62,
  63,  64,  66,  67,  69,  70,  72,  73,  75,  77,  78,  80,  82,  84,  86,  88,
  90,  91,  94,  96,  98, 100, 102, 104, 107, 109, 111, 114, 116, 119, 122, 124,
 127, 130, 133, 136, 139, 142, 145, 148, 151, 155, 158, 161, 165, 169, 172, 176,
 180, 184, 188, 192, 196, 201, 205, 210, 214, 219, 224, 229, 234, 239, 244, 250
};

const int rawLedPin  = 3;           // this LED is fed with raw values
const int adjustedLedPin = 5;       // this LED is driven from table
int brightness = 0;
int increment = 1;

void setup()
{
  // pins driven by analogWrite do not need to be declared as outputs
}
```

3. Storing and Retrieving Numeric Values in Program Memory

```
void loop()
{
  if (brightness > 254)
  {
     increment = -1; // count down after reaching 255
  }
  else if (brightness < 1)
  {
    increment =  1; // count up after dropping back down to 0
  }
  brightness = brightness + increment; // increment (or decrement sign is minus)

  // write the brightness value to the LEDs
  analogWrite(rawLedPin, brightness);  // this is the raw value
  int adjustedBrightness = pgm_read_byte(&table[brightness]);  // adjusted value
  analogWrite(adjustedLedPin, adjustedBrightness);

  delay(10); // 10ms for each step change means 2.55 secs to fade up or down
}
```

3. Storing and Retrieving Numeric Values in Program Memory

- When you need to use a complex expression to calculate a range of values that regularly repeat, it is often better to precalculate the values and include them in a table of values (usually as an array) in the code. This saves the time needed to calculate the values repeatedly when the code runs.
- The disadvantage concerns the memory needed to place these values in RAM. RAM is limited on Arduino and the much larger program memory space can be used to store constant values. This is particularly helpful for sketches that have large arrays of numbers. At the top of the sketch, the table is defined with the following expression:

```
const byte table[]PROGMEM = {
  0, . . .
```

- PROGMEM tells the compiler that the values are to be stored in program memory rather than RAM.

3. Storing and Retrieving Numeric Values in Program Memory

- The low-level definitions needed to use PROGMEM are contained in a file named pgmspace.h and the sketch includes this as follows:

```
#include <avr/pgmspace.h>
```

- To adjust the brightness to make the fade look uniform, this recipe adds the following lines to the LED output code:

```
int adjustedBrightness = pgm_read_byte(&table[brightness]);
analogWrite(adjustedLedPin, adjustedBrightness);
```

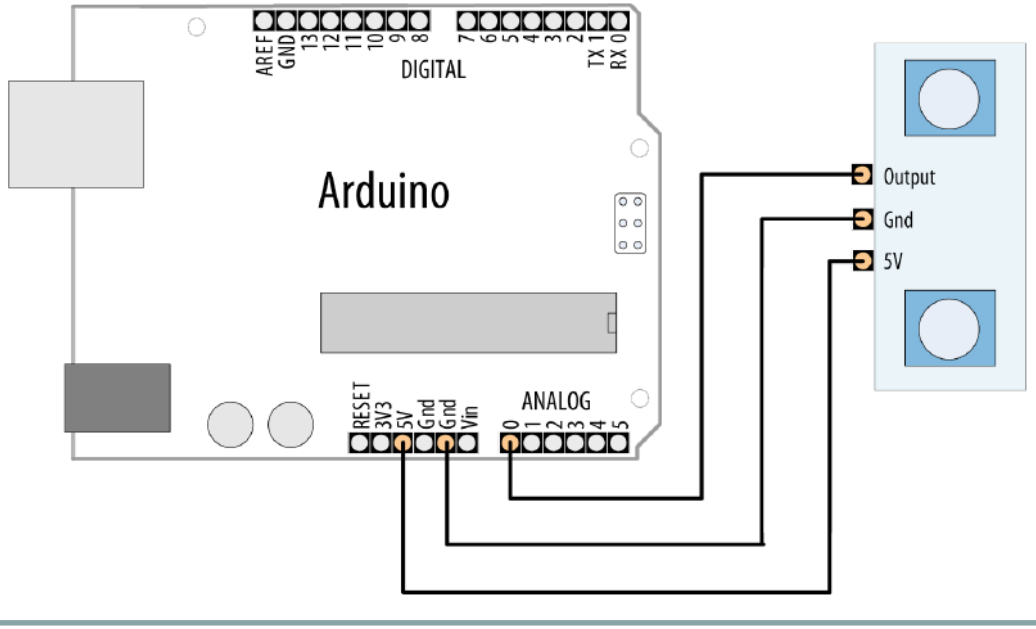
- The variable adjustedBrightness is set from a value read from program memory. The expression pgm_read_byte(&table[brightness]); means to return the address of the entry in the table array at the index position given by brightness. Each entry in the table is one byte, so another way to write this expression is:

```
pgm_read_byte(table + brightness);
```

- If it is not clear why &table[brightness] is equivalent to table + brightness, don't worry; use whichever expression makes more sense to you.

3. Storing and Retrieving Numeric Values in Program Memory

- Another example using the infrared sensor, which used a table for converting an infrared sensor reading into distance.
- Infrared (IR) sensors generally provide an analog output that can be measured using analogRead. They can have greater accuracy than ultrasonic sensors, albeit with a smaller range (a range of 10 centimeters to 1 or 2 meters is typical for IR sensors). This example uses an infrared sensor—the Sharp GP2Y0A02YK0F



3. Storing and Retrieving Numeric Values in Program Memory

- Original program with involvement RAM:

```
/* ir-distance sketch
 * prints distance and changes LED flash rate based on distance from IR sensor
 */

const int ledPin    = 13; // the pin connected to the LED to flash
const int sensorPin = 0;  // the analog pin connected to the sensor

const long referenceMv = 5000; // long int to prevent overflow when multiplied

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
}
```

3. Storing and Retrieving Numeric Values in Program Memory

- Original program with involvement RAM:

```
void loop()
{
  int val = analogRead(sensorPin);
  int mV = (val * referenceMv) / 1023;

  Serial.print(mV);
  Serial.print(",");
  int cm = getDistance(mV);
  Serial.println(cm);

  digitalWrite(ledPin, HIGH);
  delay(cm * 10 ); // each centimeter adds 10 milliseconds delay
  digitalWrite(ledPin, LOW);
  delay( cm * 10);

  delay(100);
}

// the following is used to interpolate the distance from a table
// table entries are distances in steps of 250 millivolts
const int TABLE_ENTRIES = 12;
const int firstElement = 250; // first entry is 250 mV
const int INTERVAL  = 250; // millivolts between each element
static int distance[TABLE_ENTRIES] = {150,140,130,100,60,50,40,35,30,25,20,15};

int getDistance(int mV)
{
   if( mV >  INTERVAL * TABLE_ENTRIES-1 )
      return distance[TABLE_ENTRIES-1];
   else
   {
      int index = mV / INTERVAL;
      float frac = (mV % 250) / (float)INTERVAL;
      return distance[index] - ((distance[index] - distance[index+1]) * frac);
   }
}
```

3. Storing and Retrieving Numeric Values in Program Memory

- Program to use a table in program memory instead of RAM

```
/* ir-distance_Progmem sketch
 * prints distance & changes LED flash rate depending on distance from IR sensor
 * uses progmem for table
 */

#include <avr/pgmspace.h> // needed when using Progmem

// table entries are distances in steps of 250 millivolts
const int TABLE_ENTRIES = 12;
const int firstElement = 250; // first entry is 250 mV
const int interval  = 250; // millivolts between each element
// the following is the definition of the table in Program Memory
const int distanceP[TABLE_ENTRIES] PROGMEM = { 150,140,130,100,60,50,
40,35,30,25,20,15 };

// This function reads from Program Memory at the given index
int getTableEntry(int index)
{
  int value =  pgm_read_word(&distanceP[index]);
  return value;
}
```

3. Storing and Retrieving Numeric Values in Program Memory

- The remaining code is similar to original program on RAM, except that the getTableEntry function is used to get the value from program memory instead of accessing a table in RAM. Here is the revised getDistance function from function below:

```
int getDistance(int mV)
{
   if( mV >  interval * TABLE_ENTRIES )
      return getTableEntry(TABLE_ENTRIES-1); // the minimum distance
   else
   {
      int index = mV / interval;
      float frac = (mV % 250) / (float)interval;
      return getTableEntry(index) - ((getTableEntry(index) -
getTableEntry(index+1)) * frac);
   }
}
```

- You have lots of strings and they are consuming too much RAM. You want to move string constants, such as menu prompts or debugging statements, out of RAM and into program memory.

3. Storing and Retrieving Strings in Program Memory

- This sketch creates a string in program memory and prints its value to the Serial Monitor using the F("text") expression introduced in Arduino 1.0. The technique for printing the amount of free RAM.

```
 #include <avr/pgmspace.h> // for progmem

//create a string of 20 characters in progmem
const prog_uchar myText[] PROGMEM = "arduino duemilanove ";

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.print(memoryFree());  // print the free memory
  Serial.print(' ');           // print a space

  printP(myText);              // print the string
  delay(1000);
}
```

3. Storing and Retrieving Strings in Program Memory

```
 // function to print a PROGMEM string
void printP(const prog_uchar *str)
{
char c;

   while((c = pgm_read_byte(str++)))
     Serial.write(c);
}

// variables created by the build process when compiling the sketch
extern int __bss_end;
extern void *__brkval;

// function to return the amount of free RAM
int memoryFree(){
  int freeValue;

  if((int)__brkval == 0) freeValue = ((int)&freeValue) - ((int)&__bss_end);
  else freeValue = ((int)&freeValue) - ((int)__brkval);
  return freeValue;
}
```

3. Using #define and const Instead of Integers

- If you want to minimize RAM usage by telling the compiler that the value is constant and can be optimized, you can use const to declare values that are constant throughout the sketch. For example, instead of:

```
int ledPin=13;
```

Use

```
const int ledPin=13;
```

- We often want to use a constant value in different areas of code. Just writing the number is a really bad idea. If you later want to change the value used, it's difficult to work out which numbers scattered throughout the code also need to be changed. It is best to use named references. Here are three different ways to define a value that is a constant:

```
int ledPin =  13;          // a variable, but this wastes RAM
const int ledPin =  13;    // a const does not use RAM
#define ledPin 13          // using a #define
                           // the preprocessor replaces ledPin with 13

pinMode(ledPin, OUTPUT);
```

3. Using #define and const Instead of Integers

- Although the first two expressions look similar, the term const tells the compiler not to treat ledPin as an ordinary variable. Unlike the ordinary int, no RAM is reserved to hold the value for the const, as it is guaranteed not to change. The compiler will produce exactly the same code as if you had written:

```
pinMode(13, OUTPUT);
```

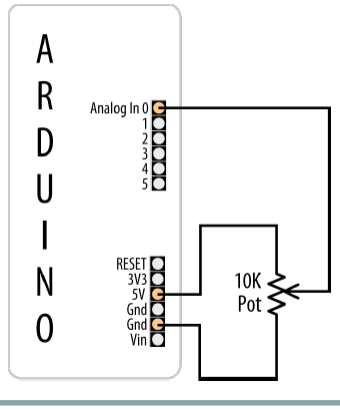
- You will sometimes see #define used to define constants in older Arduino code, but const is a better choice than #define. This is because a const variable has a type, which enables the compiler to verify and report if the variable is being used in ways not appropriate for that type. The compiler will also respect C rules for the scope of a const variable. A #define value will affect all the code in the sketch, which may be more than you intended. Another benefit of const is that it uses familiar syntax—#define does not use the equals sign, and no semicolon is used at the end.

3. Using Conditional Compilations

- You want to have different versions of your code that can be selectively compiled. For example, you may need code to work differently when debugging or when running with different boards, you can use the conditional statements aimed at the preprocessor to control how your sketch is built.
- This example includes the SPI.h library file that is only available for and needed with Arduino versions released after 0018:

```
#if ARDUINO > 18
#include <SPI.h>        // needed for Arduino versions later than 0018
#endif
```

- Another example read the voltage on an analog pin, potentiometer (pot) or a device or sensor that provides a voltage between 0 and 5 volts.



3. Using Conditional Compilations

- Original Program without Conditional Compilations:

```
/*
 Pot sketch
 blink an LED at a rate set by the position of a potentiometer
*/

const int potPin = 0;    // select the input pin for the potentiometer
const int ledPin = 13;   // select the pin for the LED
int val = 0;             // variable to store the value coming from the sensor

void setup()
{
  pinMode(ledPin, OUTPUT);  // declare the ledPin as an OUTPUT
}

void loop() {
  val = analogRead(potPin);    // read the voltage on the pot
  digitalWrite(ledPin, HIGH); // turn the ledPin on
  delay(val);                 // blink rate set by pot value (in milliseconds)
  digitalWrite(ledPin, LOW);  // turn the ledPin off
  delay(val);                 // turn led off for same period as it was turned on
}
```

3. Using Conditional Compilations

- Program displays some debug statements only if DEBUG is defined:

```
/*
 Pot_Debug sketch
 blink an LED at a rate set by the position of a potentiometer
 Uses Serial port for debug if DEBUG is defined
 */

const int potPin = 0;    // select the input pin for the potentiometer
const int ledPin = 13;   // select the pin for the LED
int val = 0;             // variable to store the value coming from the sensor

#define DEBUG

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);     // declare the ledPin as an OUTPUT
}

void loop() {
  val = analogRead(potPin);    // read the voltage on the pot
  digitalWrite(ledPin, HIGH);  // turn the ledPin on
  delay(val);                  // blink rate set by pot value
  digitalWrite(ledPin, LOW);   // turn the ledPin off
  delay(val);                  // turn LED off for same period as it was turned on
#if defined DEBUG
  Serial.println(val);
#endif
}
```

3. Using Conditional Compilations

- This recipe uses the preprocessor used at the beginning of the compile process to change what code is compiled. The first example tests if the value of the constant ARDUINO is greater than 18, and if so, the file SPI.h is included. The value of the ARDUINO constant is defined in the build process and corresponds to the Arduino release version.
- The syntax for this expression is not the same as that used for writing a sketch. Expressions that begin with the # symbol are processed before the code is compiled—see this chapter's introduction section for more on the pre-processor. You have already come across #include:

```
#include <library.h>
```

- The < > brackets tell the compiler to look for the file in the location for standard libraries:

```
#include "header.h"
```

- The compiler will also look in the sketch folder.

3. Using Conditional Compilations

- You can have a conditional compile based on the controller chip selected in the IDE. For example, the following code will produce different code when compiled for a Mega board that reads the additional analog pins that it has:

```
/*
 * ConditionalCompile sketch
 * This sketch recognizes the controller chip using conditional defines
 */


int numberOfSensors;
int val = 0;               // variable to store the value coming from the sensor

void setup()
{
   Serial.begin(9600);

#if defined(__AVR_ATmega1280__)   // defined when selecting Mega in the IDE
  numberOfSensors = 16;           // the number of analog inputs on the Mega
#else                             // if not Mega then assume a standard board
  numberOfSensors = 6;            // analog inputs on a standard Arduino board
#endif

 Serial.print("The number of sensors is ");
 Serial.println(numberOfSensors);
}

void loop() {
  for(int sensor = 0; sensor < numberOfSensors; sensor++)
  {
     val = analogRead(sensor);    // read the sensor value
     Serial.println(val);         // display the value
  }
  Serial.println();
  delay(1000);        // delay a second between readings
}
```

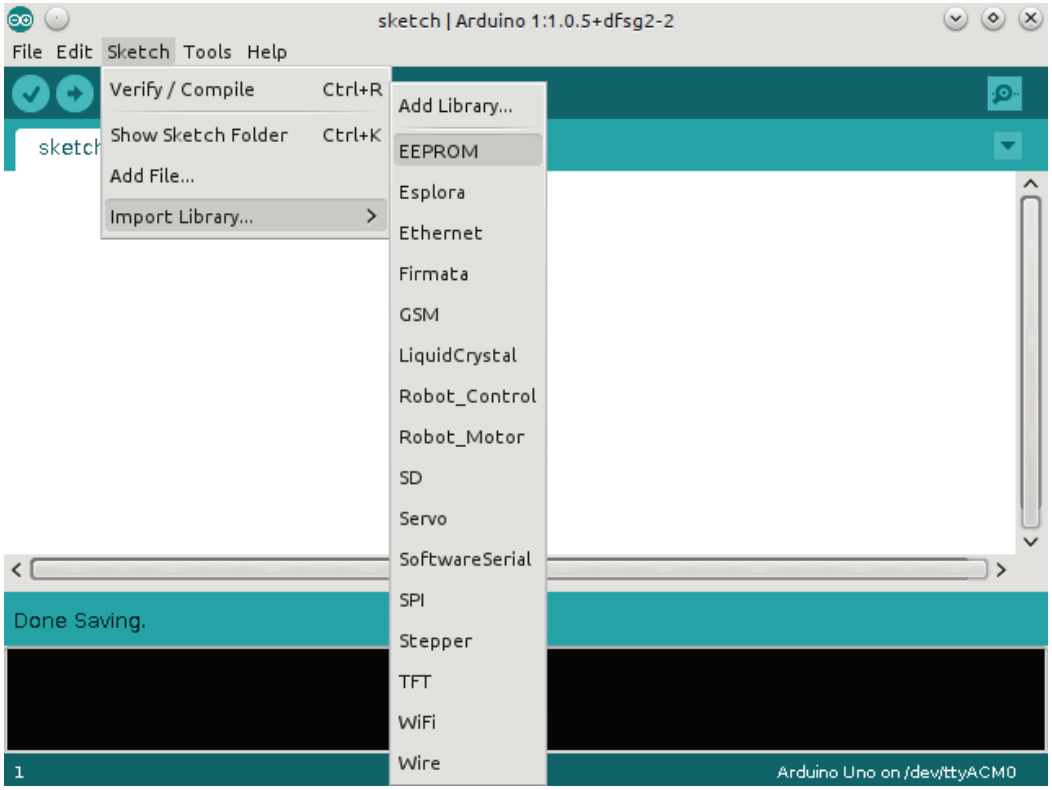
4. Storing Data in Permanent EEPROM Memory

- The EEPROM memory is a slightly different memory technology, supporting more write cycles. EEPROM memory on ATmega microcontrollers support at least 100,000 writes and can be read and written to byte by byte.
- This is the memory that will contain long-term settings and is not overwritten by each flash. Updating your sketch won't overwrite your variable.
- The EEPROM size varies for each microcontroller. The ATmega8 and ATmega168 found in early versions of the Arduino both have 512 bytes of EEPROM, and the ATmega328 in the Uno has 1,024 bytes. The ATmega1280 and ATmega2560 used in the different versions of the Arduino Mega both have 4 KB of EEPROM.

4. Storing Data in Permanent EEPROM Memory

- The EEPROM library is a collection of routines that can access the internal EEPROM memory, reading and writing bytes. The EEPROM library can be imported by manually writing the include statement:

```
#include <EEPROM.h>
```



4. EEPROM - Reading and Writing Bytes

- The entire EEPROM library consists of two functions: read() and write(). These two functions can read and write bytes from specific memory locations. The read() function reads data from a specifi ed address adr, expressed as an int, and returns data as a byte.

```
EEPROM.read(adr);
```

- The write() function writes a byte contained in data to a specific address adr. This function does not return any values:

```
EEPROM.write(adr, data);
```

- The Arduino compiler automatically sets the correct start memory location. It doesn’t matter if you use an Uno, Mega2560, or Mini; the compiler “translates” the correct address. Reading at memory location 0 read from the fi rst byte of EEPROM.

4. EEPROM - Reading and Writing Bytes

```
byte value;
void setup()
{
  // initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    // wait for serial port to connect. Needed for Leonardo only
  }
  value = EEPROM.read(0);
  Serial.print("Value at position 0:");
  Serial.print(value, DEC);
  Serial.println();
}
void loop(){}
```

- In this program, the Arduino reads the first byte of EEPROM memory and displays it over the serial interface. Yes, it is that simple. Writing a byte into memory is just as straightforward:

4. EEPROM - Reading and Writing Bytes

```
void setup()
{
  EEPROM.write(0, value);
}

void loop() {}
```

- Writing a byte erases the byte in memory before rewriting, and this takes some time. Each write takes approximately 3.3 ms for each byte. Writing the entire contents of a 512-byte EEPROM device takes a little more than 1 1/2 seconds.

4. EEPROM - Reading and Writing Bits

- Bits are used when using true/false values. In some applications there will be relatively few (or sometimes none at all), and in others, you will use Boolean variables extensively. An Arduino cannot write individual bits to EEPROM; to store bits, they must first be stored in a byte. There are two possibilities.
- If you have a single bit to store, the easiest way is just to code it as a byte, even if you use 1 out of 8 bits.
- If you have several bits to store, you might want to try storing them all in 1 byte to save space, for example, a notifi cation LED that the user can program as he wants. If this is an RGB LED, the user can choose a mix of any primary colors for notifi cation. This can be coded into 3 bits; 1 for red, 1 for green, and 1 for blue. A logical 1 means the color is present, and a logical 0 means the color is not present.

4. EEPROM - Reading and Writing Bits

- You can define this as follows:

```
// primary colors
#define BLUE  4  // 100
#define GREEN 2  // 010
#define RED   1  // 001
```

- Did you note that RED was defined as 1, and has the number 001 next to it? Arduinos, like all computer systems, store data as binary—a collection of ones and zeros. It is critical to understand binary when performing bitwise calculations.
- Binary is a base-two system; that is to say that each digit can take one of two possible values—0 or 1. The rightmost fi gure corresponds to 2^0, the number to ts left corresponds to 2^1, the next one to 2^2 , and so on. In this example, I have used three specific values: 1, 2, and 4. I did not use 3 since in binary, 3 is written as 011, and I wanted each color to be assigned to a bit.
- There are five more bits that could be coded into this byte. Each bit couldindicate another behavior; maybe the LED should blink? Or maybe a warning beep? You can make this decision.

4. EEPROM - Reading and Writing Bits

- Also, another important part of bitwise calculations is AND and OR. In binary logic, a result is TRUE if one value AND the second value are both TRUE. TRUE and TRUE would result in TRUE, but TRUE and FALSE would result in FALSE. A result is TRUE if one value OR another value is TRUE. 1 OR 1 is TRUE, as is 1 OR 0, but 0 OR 0 is FALSE.
- Let's imagine you want a cyan light to be lit up if something occurs. Cyan is a mix of green and blue. In English, you would say that you want green and blue, but in computer logic, you would say that you want GREEN or BLUE. A logical OR is true if one of the two values being compared is true. In this case, GREEN (010) is compared to BLUE (100), and the answer becomes 110.

4. EEPROM - Reading and Writing Bits

- So, the result, called CYAN, is 110, but now that you have encoded that, how can you get the data out of it? This time, you will be using a logical AND. A logical AND is true if the both the values being compared are true. So, CYAN AND BLUE? CYAN has a value of 110, and the value of BLUE is 100. The leftmost bit is 1 in both, so that will return as a 1. The second bit is 1 in CYAN and 0 in BLUE. It returns 0. The third bit is 0 in both values; it also returns 0. The result is 100. You can now say that BLUE is present in CYAN because the result was not zero.
- Now, time to try that again with RED. The value of CYAN is 110, and RED is 001. The first two bits are 1 in CYAN and 0 in RED. They return 0. The third bit is 0 in CYAN and 1 in RED. The logical AND process returns 000. There is no RED in CYAN because CYAN AND RED returns 0.

4. EEPROM - Reading and Writing Bits

- To read boolean data, read the byte containing the data from EEPROM and then perform a logical AND with the reference value. To create boolean data, you must take an empty variable (initialized as 0) and then perform logical OR operations with reference values. What happens if you want to update an existing value? You already know how to set a bit, using a logical OR, but to clear a bit, you must use a logical NOT AND. NOT inverts a status; if it was previously TRUE, it will become FALSE. By inverting the reference, you keep every bit that is set except the one you want to clear. To toggle a bit, simply use the logical XOR to invert its status. XOR, short for Exclusive OR, will be true if and only if one of the inputs is TRUE; if they are both TRUE, then the result will be FALSE.

4. EEPROM - Reading and Writing Bits

- Table below is table of logical operators, showing the effect of each.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0
		OR	AND	XOR	NOT

- Following is a short example of how to perform bitwise operations. A bitwise OR is performed using the | symbol:

```
value |= RED; // Bitwise OR. Sets the BLUE bit
```

To perform a bitwise AND, use the & symbol:

```
vavalue &= ~GREEN; // Bitwise AND. Clears the RED bit (AND NOT RED)
```

And finally, to perform an exclusive OR, use the ^ symbol:

```
value ^= BLUE; // Bitwise XOR. Toggles the GREEN bit
```

4. EEPROM - Reading and Writing Strings

- Strings are generally an array of char values and as such can be easily stored and recalled. In Arduino, it's possible to use a char array as a string, or you can use the String data type for more robust data manipulation, at the cost of program size. With character arrays, you can recall the entire allocated memory and print it out as required.

Suppose you need to store a string, defined as such:

```
char myString[20];
```

You can also set a string to a specific value when you declare it. Note that while this array can contain up to 20 elements, not all of them have data.

```
char myString[20] = "Hello, world!";
```

You can store information in EEPROM like this:

```
int i;
for (i = 0; i < sizeof(myString); i++)
{
  EEPROM.write(i, myString[i]);
}
```

4. EEPROM - Reading and Writing Strings

- This routine will write the contents of the string to EEPROM memory, one byte at a time. Even if the string is only 5 bytes long, it will store the contents of the entire array. That is, if you declare a char array of 20 elements and only have valid data in the fi rst 5 bytes, you'll still be writing 20 bytes to EEPROM.
- You could make a more optimized routine that automatically stops when it receives a null character: the end of a C string, but because this routine writes to EEPROM memory that is not often (if ever) changed, there is no point to over complexifying the program. Reading a string is just as easy and the operation is the same; it will take 1 byte from EEPROM and place it into the string, and repeat for each byte in the string.

```
int i;
for (i = 0; i < sizeof(myString); i++)
  {
    myString[i] = EEPROM.read(i);
  }
```

4. EEPROM - Reading and Writing Other Values

- If the EEPROM can only read and write bytes, how can you save the contents of an integer or a floating point number? At first it might seem impossible, but remember that in computers, everything is just 1s and 0s. Even a floating-point number is written in memory as binary, it just occupies a larger number of bytes. Just like with strings, it is possible to write just about anything in EEPROM memory, by reading and writing 1 byte at a time.
- Before beginning, you must know exactly what sort of data you need to read and write. For example, on all Arduinos except the Due, an int is written as 2 bytes. By using techniques known as shifts and masks, it is possible to “extract” bytes of data. Shifting takes a binary number and “shifts” data to the left or to the right by a certain number of bits. Masking makes it possible to perform bitwise operations on a portion of a binary number.

4. EEPROM - Reading and Writing Other Values

- Take the following example:

```
byte value;
void setup()
{
  // initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    // wait for serial port to connect. Needed for Leonardo only
  }
  value = EEPROM.read(0);
  Serial.print("Value at position 0:");
  Serial.print(value, DEC);
  Serial.println();
}
void loop(){}
```

4. EEPROM - Reading and Writing Other Values

- In this example, an int is to be saved into EEPROM. It contains two bytes: the low byte and the high byte. The terminology “low” and “high” bytes is used when a number is stored on several bytes; the low byte contains the least significant part of the number, and the high byte contains the most significant part of the number.
- First, the lowest byte is extracted. It simply takes the number and performs a bitwise AND with 0xFF. The 0x in front of the letters tells the Arduino IDE that this is a hexadecimal number. Just like binary, hexadecimal is another way of printing a number. Instead of using only two values per figure, hexadecimal uses 16. 0xFF is the hexadecimal representation of 255, the largest number that a byte can hold. Then, the same value is shifted right 8 bits, and again, an AND is performed. This is an elegant solution that can work for integers but will not work for more complex numbers, like a fl oating-point. You cannot perform shifts with a floating-point, more advanced techniques are required.

4. EEPROM - Reading and Writing Other Values

- Other example: program that would greet the user, ask for name and age and store the responses in EEPROM.
- The Arduino should fi rst check its EEPROM memory. If no information is found, it will ask the user some questions and then store that information into nonvolatile memory. If the information is found, it will tell the user what information it has and then delete the contents of its memory.

```
1   #include <EEPROM.h>
2
3   #define EEPROM_DATAPOS 0
4   #define EEPROM_AGEPOS 1
5   #define EEPROM_NAMEPOS 2
6   #define EEPROM_CONTROL 42
7
8   char myName[] = {"Arduino"};
9   char userName[64];
10  char userAge[32];
11  unsigned char age;
12  int i;
13  byte myValue = 0;
14
```

4. EEPROM - Reading and Writing Other Values

```
15  void setup()
16  {
17    // Configure the serial port:
18    Serial.begin(9600);
19
20    // Does the EEPROM have any information?
21    myValue = EEPROM.read(EEPROM_DATAPOS);
22
23    if (myValue == 42)
24    {
25      // Get the user's name
26      for (i = 0; i < sizeof(userName); i++)
27      {
28        userName[i] = EEPROM.read(EEPROM_NAMEPOS + i);
29      }
30
31      // Get the user's age
32      age = EEPROM.read(EEPROM_AGEPOS);
33
34      // Print out what we know of the user
35      Serial.println("I know you!");
36      Serial.print("Your name is ");
37      Serial.print(userName);
38      Serial.print(" and you are ");
39      Serial.print(age);
40      Serial.println(" years old.");
41
42      // Write zero back to the control number
43      EEPROM.write(EEPROM_DATAPOS, 0);
44    }
45    else
46    {
47      // Welcome the user
48      Serial.println("Hello! What is your name?");
49
50      // Wait until serial data is available
51      while(!Serial.available())
52      // Wait for all the data to arrive
53      delay(200);
54
55      // Read in serial data, one byte at a time
56      Serial.readBytes(userName, Serial.available());
57
58      // Say hello to the user
59      Serial.print("Hello, ");
60      Serial.print(userName);
61      Serial.print(". My name is ");
62      Serial.print(myName);
63      Serial.println("\n");
```

4. EEPROM - Reading and Writing Other Values

```
64
65      // Save the user's name to EEPROM
66      for (i = 0; i < sizeof(userName); i++)
67      {
68        EEPROM.write(EEPROM_NAMEPOS + i, userName[i]);
69      }
70
71      // Ask for user's age
72      Serial.print("How old are you, ");
73      Serial.print(userName);
74      Serial.println("?");
75
76      // Wait until serial data is available
77      while(!Serial.available())
78      // Wait for all the data to arrive
79      delay(200);
80      age = Serial.parseInt();
```

4. EEPROM - Reading and Writing Other Values

```
81
82      // Print out the user's age
83      Serial.print("Oh, you are ");
84      Serial.print(age);
85      Serial.println("?");
86      Serial.print("I am ");
87      Serial.print(millis());
88      Serial.println(" microseconds old. Well, my sketch is.");
89
90      // Now save this to EEPROM memory
91      EEPROM.write(EEPROM_AGEPOS, age);
92
93      // Since we have all the information we need, and it has been
94      //saved, write a control number to EEPROM
95      EEPROM.write(EEPROM_DATAPOS, EEPROM_CONTROL);
96    }
97
98  }
99
100 void loop()
101 {
102   // put your main code here, to run repeatedly:
103 }
```

4. EEPROM - Reading and Writing Other Values

- On line 11, the user's age is now stored in an unsigned char. Originally this was stored in an int, but this presents a problem for EEPROM memory. Remember that in Chapter 4 you saw that int values stored from –32768 to 32767.
- You won't need all those numbers; humans don't (yet) live that long, and in any case, negative numbers aren't necessary. The problem isn't the range; it is the size of the container. On most Arduinos, an int is coded on 2 bytes (in the Due it occupies 4 bytes). If you release your program as open source, you will have no way of knowing which Arduino will be used. In addition, an int for an age is a bad idea; it isn't optimal. An unsigned char is always 1 byte and can handle numbers from 0 all the way to 255. This will be easier to write to an EEPROM.

4. EEPROM - Reading and Writing Other Values

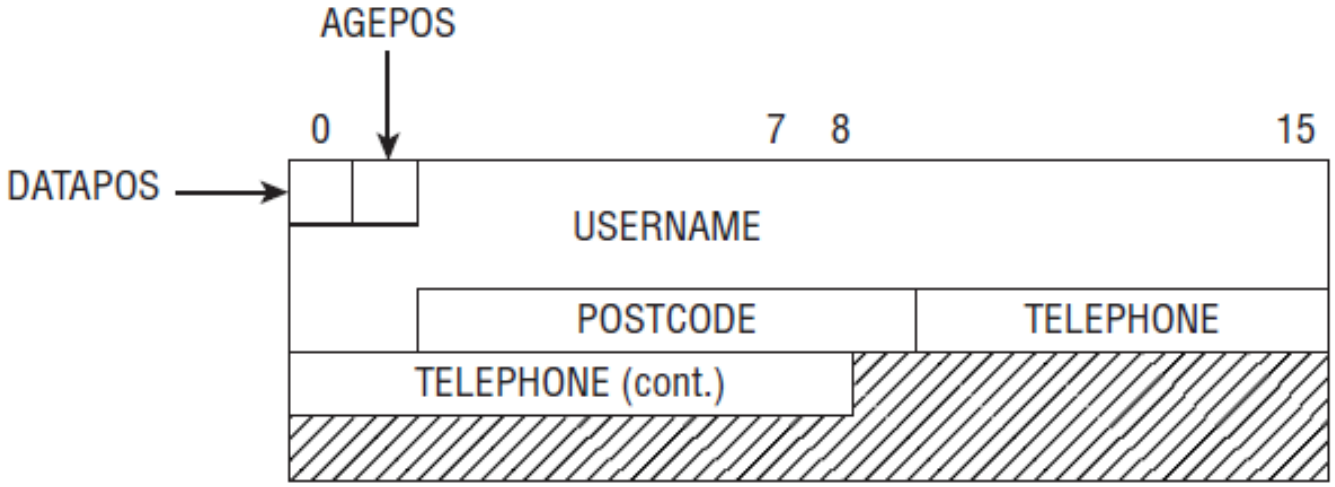
- On line 21, the sketch reads data from the EEPROM. The exact location is defined by EEPROM_DATAPOS. Of course, the function could have been called directly with the number 0 (and this is exactly what the compiler is going to do), but adding a #define makes the code more readable and also allows the developer to change memory location without worrying about forgetting a call.
- This makes everything neater. This sketch shows the persistence of non-volatile memory, and as such, it has to have a way of ignoring any data stored. To do this, a “control” byte is allocated. The Arduino reads a value in the EEPROM. If it receives the number 42, it presumes that the EEPROM contains valid information and attempts to read that data. If the Arduino reads any other number, it asks the user for information, writes that data to EEPROM, and then writes the control byte.

4. EEPROM - Reading and Writing Other Values

- Assuming that no valid EEPROM data has been found, the sketch is close to what was already present in the previous chapter. On lines 50 and 76, the serial call has been changed. At the end of the previous example, I asked you to try and find a better way of listening for serial communication. This is one way of waiting for serial data.
- On line 91, the sketch saves the contents of the variable age to EEPROM using a single function call: EEPROM.write(). However, on line 65, the string username is saved 1 byte at a time. The entire string memory is written to EEPROM, but you could tweak the code to write only what is needed. What would you write?
- This brings the question: How do you organize memory? It is up to you, the engineer and creator, to decide how the memory will be partitioned. This example used position 0 as the control byte, position 1 as the age, and 20 bytes from position 2 onward as a string containing your name. Don’t hesitate to use a spreadsheet or some paper notes to map out your memory, to know what will go where.

4. EEPROM - Reading and Writing Other Values

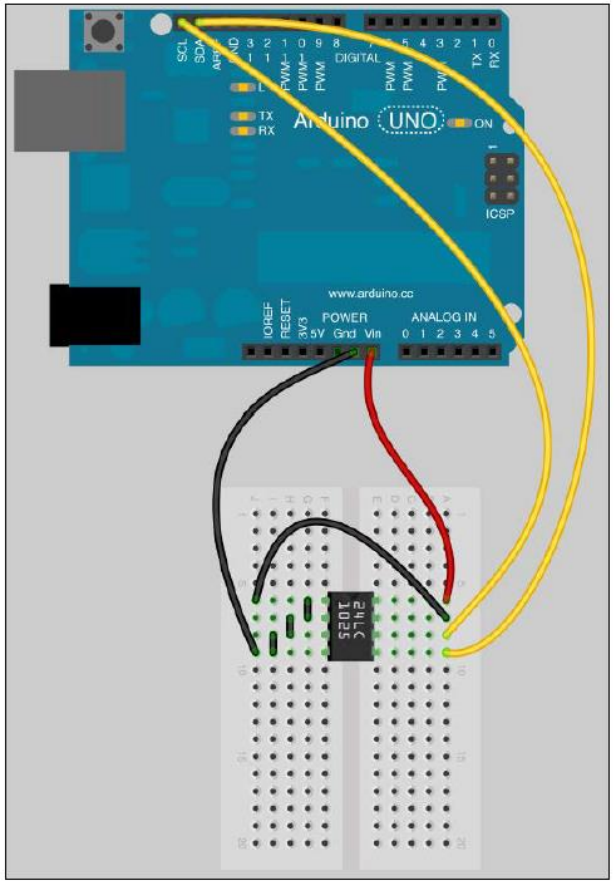
- Memory organization”

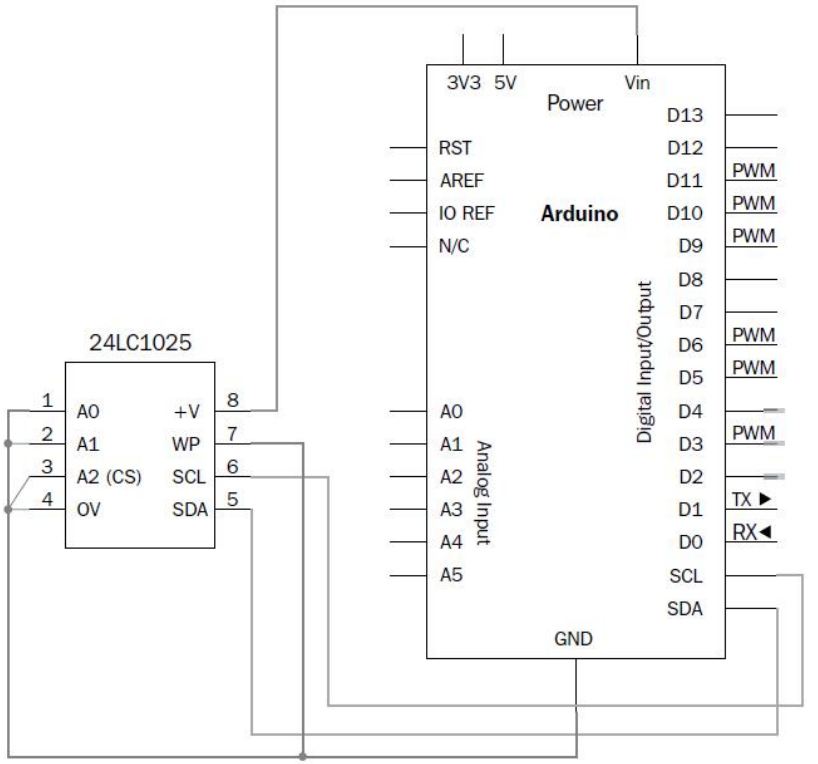


- Keep in mind that #define statements are easier to change rather than looking through your code if you need to change something.

5. External EEPROM

- There are a lot of cheap EEPROM components available in electronics markets. For example, to use the classic 24LC1025, an EEPROM implementing I2C for read/write operations and providing 1024k bytes of memory space.





5. External EEPROM

- A0, A1, and A2 are chip address inputs. +V and 0V are 5V and ground. WP is the write protect pin. If it is wired to ground, we can write to the EEPROM. If it is wired to 5V, we cannot. SCL and SDA are the two wires involved in the I2C communication and are wired to SDA / SCL. SDA stands for Serial Data Line and SCL stands for Serial Clock Line.
- Be careful about the SDA/SCL pins. The following depends on your board:
  - The Arduino UNO before R3 and Ethernet's I2C pins are A4 (SDA) and A5 (SCL)
  - Mega2560, pins 20 (SDA) and 21 (SCL)
  - Leonardo, pin 2 (SDA) and pin 3 (SCL)
  - Due Pins, pins 20 (SDA) and 21 (SCL) and also another one SDA1 and SCL1.

5. External EEPROM

- Reading and writing to the EEPROM The underlying library that we can use for I2C purposes is Wire. You can find it directly in the Arduino core. This library takes care of the raw bits, but we have to look at it more closely. The Wire library takes care of many things for us.

```
#include <Wire.h>

void eepromWrite(byte address, byte source_addr,
  Wire.beginTransmission(address);
  Wire.write(source_addr);
  Wire.write(data);
  Wire.endTransmission();
}

byte eepromRead(int address, int source_addr) {
  Wire.beginTransmission(address);
  Wire.write(source_addr);
  Wire.endTransmission();

  Wire.requestFrom(address, 1);
  if(Wire.available())
    return Wire.read();
  else
    return 0xFF;
}
```

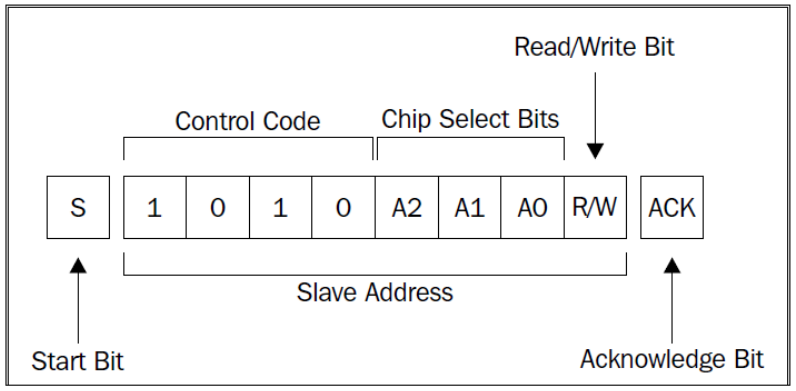
5. External EEPROM

```
void setup() {
  Wire.begin();
  Serial.begin(9600);

  for(int i = 0; i < 10; i++) {
    eepromWrite(B01010000, i, 'a'+i);
    delay(100);
  }

  Serial.println("Bytes written to external EEPROM !");
}

void loop() {
  for(int i = 0; i < 10; i++) {
    byte val = eepromRead(B01010000, i);
    Serial.print(i);
    Serial.print("\t");
    Serial.print(val);
    Serial.print("\n");
    delay(1000);
  }
}
```

5. External EEPROM

- We include the Wire library at first. Then we define 2 functions:
  - eepromWrite()
  - eepromRead()
- These functions write and read bytes to and from the external EEPROM using the Wire library.
- The Setup() function instantiates the Wire and the Serial communication. Then using a for loop, we write data to a specific address. This data is basically a character 'a' plus a number. This structure writes characters from a to a + 9 which means 'j'.
- This is an example to show how we can store things quickly, but of course we could have written more meaningful data.
- We then print a message to the Serial Monitor in order to tell the user that Arduino has finished writing to the EEPROM.
- In the loop() function, we then read the EEPROM. It is quite similar to the EEPROM library with the addresses using I2C message format as follows:

5. External EEPROM



- Wire library takes care of Start Bit and Acknowledge Bit. The control code is fixed and you can change the Chip Select Bits by wiring A0, A1, and A2 pins to ground or +V. That means there are 8 possibilities of addresses from 0 to 7. 1010000 1010001… until 1010111. 1010000 binary means 0x50 in hexadecimal, and 1010111 means 0x57.
- In our case, we wired A0, A1, and A2 to ground, then the EEPROM address on the I2C bus is 0x50. We could use more than one on the I2C bus, but only if we need more storage capacity. Indeed, we would have to address the different devices inside our firmware.

5. External EEPROM

- Wire library takes care of Start Bit and Acknowledge Bit. The control code is fixed and you can change the Chip Select Bits by wiring A0, A1, and A2 pins to ground or +V. That means there are 8 possibilities of addresses from 0 to 7. 1010000 1010001… until 1010111. 1010000 binary means 0x50 in hexadecimal, and 1010111 means 0x57
- In our case, we wired A0, A1, and A2 to ground, then the EEPROM address on the I2C bus is 0x50. We could use more than one on the I2C bus, but only if we need more storage capacity. Indeed, we would have to address the different devices inside our firmware.

5. External EEPROM

DS28E07 **1024-Bit, 1-Wire EEPROM**

General Description

The DS28E07 is a 1024-bit, 1-Wire® EEPROM chip organized as four memory pages of 256 bits each. Data is written to an 8-byte scratchpad, verified, and then copied to the EEPROM memory. As a special feature, the four user memory pages can individually be write protected or put in EPROM-emulation mode, where bits can only be changed from a 1 to a 0 state. Each device has its own guaranteed unique 64-bit ROM identification number (ROM ID) that is factory programmed into the chip. The communication follows the 1-Wire protocol with the ROM ID acting as node address in the case of a multiple-device 1-Wire network.

Applications

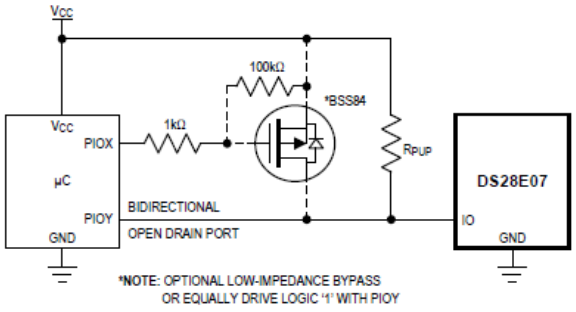
- Accessory/PCB Identification
- Medical Sensor Calibration Data Storage
- Analog Sensor Calibration Including IEEE P1451.4 Smart Sensors
- Ink and Toner Print Cartridge Identification
- After-Market Management of Consumables

Benefits and Features

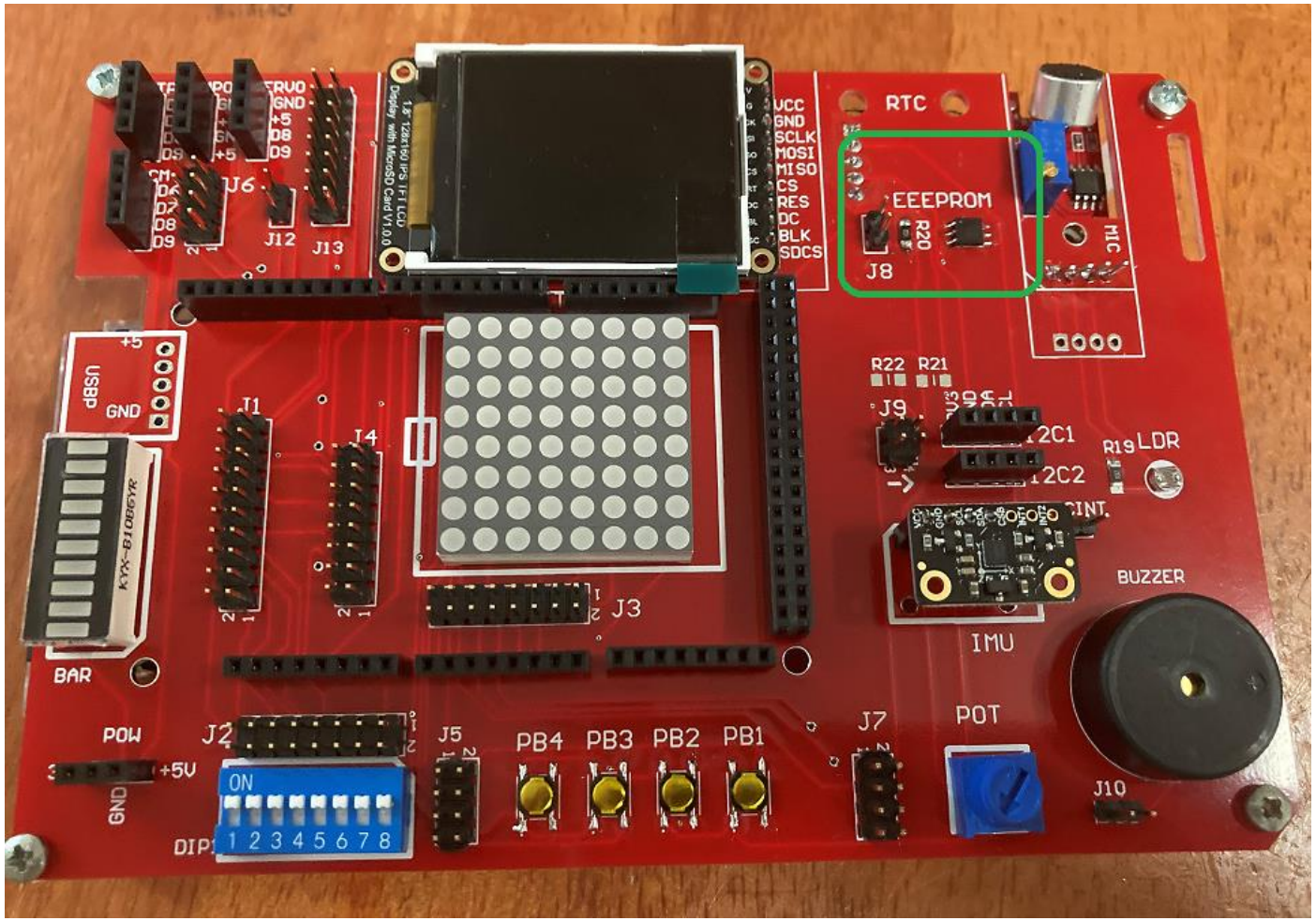
- Partitioning of Memory Provides Greater Flexibility in Programming User Data
  - 1024 Bits of EEPROM Memory Organized as Four Pages of 256 Bits
  - Individual Memory Pages Can Be Permanently Write Protected or Put in EPROM-Emulation Mode (Write to 0)
- Advanced 1-Wire Protocol Minimizes Interface to Just Single IO Reducing Required Pin Count and Enhancing Reliability
  - Unique Factory-Programmed, Unalterable 64-Bit Identification Number
  - Switchpoint Hysteresis and Filtering to Optimize Performance in the Presence of Noise
  - Communicates to Host with a Single Digital Signal at 15.4kbps or 125kbps Using 1-Wire Protocol
  - Reads and Writes over a Wide Voltage Range from 3.0V to 5.25V from -40°C to +85°C
  - ±8kV HBM ESD Protection (typ) for IO Pin

Ordering Information ***appears at end of data sheet.***

Typical Application Circuit



5. External EEPROM



5. External EEPROM

ArduinoTeachingBoard_EEPROM_final.ino

```
#include <OneWire.h>

// Define the one wire bus and the address of the DS28E07 chip
OneWire ds(8);  // One wire bus is on digital pin 2
byte addr[] = {0x2D, 0x3B , 0xE8 , 0x67 , 0x40 , 0x0 , 0x0 , 0x1}; //insert the Address of EEPROM value

void setup() {
  Serial.begin(9600);
  while (!Serial) {}

  Serial.println("--------------------------------------------------------------------------------");

  Serial.println("Reading from EEPROM (FROM MEMORY)");       //Reading from eeprom after power-cycle
  String EEPROMString = readFromEEPROM(8);
  EEPROMString.remove(7,12);            //trim undefined characters
  Serial.println(EEPROMString);

  Serial.println("Getting Family code / Address of EEPROM:");
  searchFunction();

  // Write a test string to the EEPROM
  Serial.println("Writing to EEPROM");
  String testString = "NewLine ";
  writeToEEPROM(testString);

  // Read the string back from the EEPROM and print it to the serial monitor
  Serial.println("Reading from EEPROM");
  String readString = readFromEEPROM(testString.length());
  readString.remove(7,12);
  Serial.println(readString);
}

void loop() {
}

void searchFunction() {
  byte i;
  byte addr[8];

  if (!ds.search(addr)) {
    ds.reset_search();
    delay(1000);
  }

  for (i = 0; i < 8; i++) {
    Serial.print(addr[i], HEX);
    Serial.print(" ");
  }
  Serial.print("\n");
}
```

5. External EEPROM

```
52  // Function to write a string to the DS28E07 EEPROM
53  void writeToEEPROM(String data) {
54    // Convert the input string to a char array
55    char dataChars[30];
56    data.toCharArray(dataChars, 30);
57
58    // Write the data to the EEPROM
59    ds.reset(); //reset device
60    ds.select(addr);//select address of device to talk to
61    ds.write(0x0F,1);   // Write ScratchPad       Configuration/mode bytes can be in datasheet
62    ds.write(0x01,1);   //TA1 addresses           TA (Target Address) can also be found in the datasheet
63    ds.write(0x09,1);   //TA2 addresses
64    for ( int i = 0; i < data.length(); i++) {
65      ds.write(dataChars[i],1);
66    }
67    ds.reset();
68    ds.select(addr);
69    ds.write(0x0F, 1);  // Copy ScratchPad
70  }
71
72  // Function to read a string from the DS28E07 EEPROM
73  String readFromEEPROM(int length) {
74    byte dataBytes[30];
75    ds.reset();
76    ds.select(addr);
77    ds.write(0xAA);  // Read Scratchpad
78    for (int i = 0; i < length+4; i++) {
79      dataBytes[i] = ds.read();
80    }
81    return String((char*)dataBytes).substring(3);
82  }
```

5. External EEPROM

6. SD Card

- SD, short for Secure Digital, is an evolution over the previous MultiMediaCard standard. The SD Card Association manages the format, specifi cations, and evolutions, and uses a trademarked logo to enforce compatibility. If your device has the same logo as the one on your SD card, you know that they will be compatible.
- The card capacity is only one factor. To use a card's capacity, the system normally needs to use a fi lesystem. A fi lesystem is a way of preparing the space on a physical storage medium (SD-card, floppy, or hard drive) to allow fi les and folders to be stored in a hierarchal way. SD cards can be used to transfer data between devices and operating systems with different specifications. From this variety of formats, FAT has emerged as the most common fi le system.

6. SD Card

- FAT, short for File Allocation Table, has been used since the early days of PCs. It has undergone several changes over the years. The original FAT specification, FAT8, is no longer in use. FAT16 uses 16 bits to define sector entries (a method of storing fi le information) and is limited to 2 gigabyte partitions. FAT32 was released after this, and storage space was theoretically increased to 2 terabytes; although in practice, few systems used it beyond 32 gigabytes.
- Newer systems use the exFAT filesystem, a new but incompatible file system that allows huge storage capacity; in theory, up to 64 zettabytes. For comparison, in 2013, the entire World Wide Web was estimated at 4 zettabytes. FAT32 has been surpassed technically by several file systems, including exFAT and NTFS, but still remains in use for its simplicity. NTFS adds several interesting features such as journaling, linking, and quotas; features that are not required by a digital camera. The code required to interact with a FAT32 file system is extremely small, making it ideal for embedded systems.

6. SD Card

- The Arduino SD library can work with SD and SD-HC cards, all the way up to 32 gigabytes. This limitation is mainly due to the filesystem; Arduinos can use FAT16 and FAT32 filesystems but cannot use the newer, proprietary exFAT.
- SD-XC cards are normally formatted with exFAT, but some people have reported using SD-XC cards formatted to FAT-32.
- An Arduino can work with any speed classes of SD-cards, but data throughput will be limited when writing with an Arduino. You may want to buy a faster card if you transfer data to and from a PC.
- Communications to and from the SD card are done via SPI. The SS pin (SPI Slave Select) must be left untouched. The SD library will not work if the SS pin is not configured as an output.
- Numerous shields exist and do not always use the same pin to initialize the SD card. The chip select pin can change from one design to another; consult the shield documentation to know which pin to use when initializing the SD card reader.

6. SD Card

- The Arduino language has an SD library built in. This library depends on three other internal libraries that handle card and filesystem-specific functions, but abstraction makes the library extremely easy to use.
- Importing the Library: To be able to use the SD library, you must first import it. This can be done either automatically in the Arduino IDE by going to the Sketch ⇨ Import Library ⇨ SD menu item, or manually with this:

```
#include <SD.h>
```

- Arduinos communicate with SD card controllers using the SPI protocol. Thus, you must also import that library

```
#include <SPI.h>
```

6. SD Card

- Connecting the card: as with many Arduino libraries, to initialize the library, you must call SD.begin().

```
result = SD.begin();
result = SD.begin(csPin);
```

- SD.begin() returns true if a card is detected and the library initialized; otherwise, it returns false. The optional csPin argument is used to configure which slave select pin should be used if your application does not use the default hardware SS pin. Most shields will use the default hardware pin.

```
// See if the card is present and can be initialized:
if (!SD.begin(chipSelectPin)) {
  Serial.println("Could not initialize SD card.");
  // End the sketch gracefully
  return;
}
Serial.println("SD Card initialized.");
```

6. SD Card

- Opening and Closing Files: The SD library can create, update, and delete files on a FAT16/32 filesystem. The SD library (and indeed most programming environments) does not differentiate between creating a fi le and opening a file. The system is told to open a file. If the files exists, it will be opened. If it does not exist, an entry is created, and a new blank file is opened. To open a file, call SD.open().

```
file = SD.open(filepath);
file = SD.open(filepath, mode);
```

- To open a file, you must first create a File object, and then use that object on subsequent file actions:

```
File myFile;
myFile = SD.open("data.dat", FILE_WRITE);
```

It is also possible to check beforehand if a file exists. To do this, use `SD.exists()`.

```
result = SD.exists(filename);
```

6. SD Card

- Reading files: to read a byte from a file, use the read() function of the File class

data = file.read();	This function returns 1 byte at a time (or −1 if no data is available) and automatically updates the pointer. If you do not want the pointer to be updated, you can call peek().
data = file.peek();	Its use is exactly the same as read(), returning 1 byte, but the pointer is not updated. Several calls to peek() returns the same byte. To know the value of the pointer (to know which byte is the next to be read), use position().
result = file.position();	This function does not take any parameters and returns an unsigned long indicating the current position within the fi le. It is also possible to set the position with seek().
result = file.seek(position);	This function attempts to set the fi le pointer to the value of position, defined as an unsigned long. To know the size of the current open fi le, use size(). It returns the fi le size in bytes as an unsigned long.
data = file.size();	To know if there are any more bytes available for reading, use available().
number = file.available();	This function returns the remaining bytes inside a fi le, as an int.

6. SD Card

- Writing files: Three functions are used to write data to a fi le. print() and println() are used in the same way as the Serial functions of the same name and write()places bytes at the pointer position in the file.
- print() and println() can be used to write formatted data: text and decimal numbers, as well as binary, hexadecimal, and octal representations using the optional base parameter. By specifying BIN as the base parameter, print will write binary notation. Using OCT and HEX, print will write octal and hexadecimal respectively. The difference between print() and println() is that println() automatically adds a new line character at the end. Both of these functions ignore the file pointer and append data to the end of the file.

```
file.print(data);
file.print(data, base);
file.println(data);
file.println(data, base);
```

6. SD Card

- The write() function is different. It can write data directly inside a file but will not insert data; it will overwrite any data present if not at the end of the file.

```
file.write(data);
file.write(buffer, len);
```

- The data parameter can be a byte, a char, or a string. The buffer parameter is a byte, array of char, or a String, and the len parameter indicates the number of bytes to be used.
- write(), print(), and println() also return the number of bytes written to the buffer, but reading this is optional.

6. SD Card

- Folder operations

result = SD.mkdir(folder);	To create a folder. This function returns true if the folder was created, or false if the operation did not succeed. It takes a string as a parameter and is the folder to be created(complete with forward slashes).
SD.mkdir("/data/sensors/temperature");	It can also create intermediate folders if required
result = SD.rmdir(folder);	To remove a folder. This deletes the folder from the fi lesystem but only on the condition that it is empty. The function returns true if the folder were deleted, or false if it did not complete the operation.
result = file.isDirectory();	This function takes no parameters and returns a boolean; true if the file is a folder, and false if the fi le is a regular file.

- Card operations : Data is buffered; that is to say that when the sketch is told to save data, that data is not necessarily written to the SD card immediately. Because SD cards have an embedded controller, write operations can be queued and the actual write can be performed a few seconds later. When the SD embedded controller receives multiple write operations, later write operations are often delayed until the card has fi nished current operations. To force all data to be written to a file, use flush().

```
flush(file);
```

6. SD Card

- Advanced Usage: The SD library actually makes use of three internal libraries: Sd2Card, SdVolume, and SdFile. All the functions present in the SD library are wrapper functions that call different functions in these three libraries. The SD library follows the Arduino philosophy, making it easy to do advanced functions. However, you can still use these three libraries if you need access to even more advanced functions.

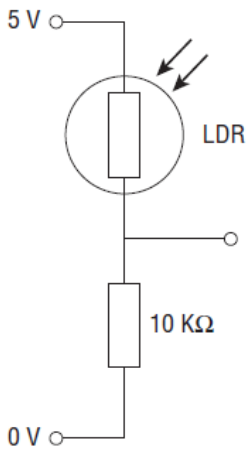
```
Sd2Card card;
SdVolume volume;
SdFile root;
```

- There are numerous functions, and these functions are mainly out of the scope of this book, but there are a few that may be of interest. To get information about the card size, you can get data about the geometry of the SD card—that is, the number of clusters and the number of blocks per cluster.

```
unsigned long volumesize = volume.blocksPerCluster();
volumesize *= volume.clusterCount();
volumesize *= 512;
```

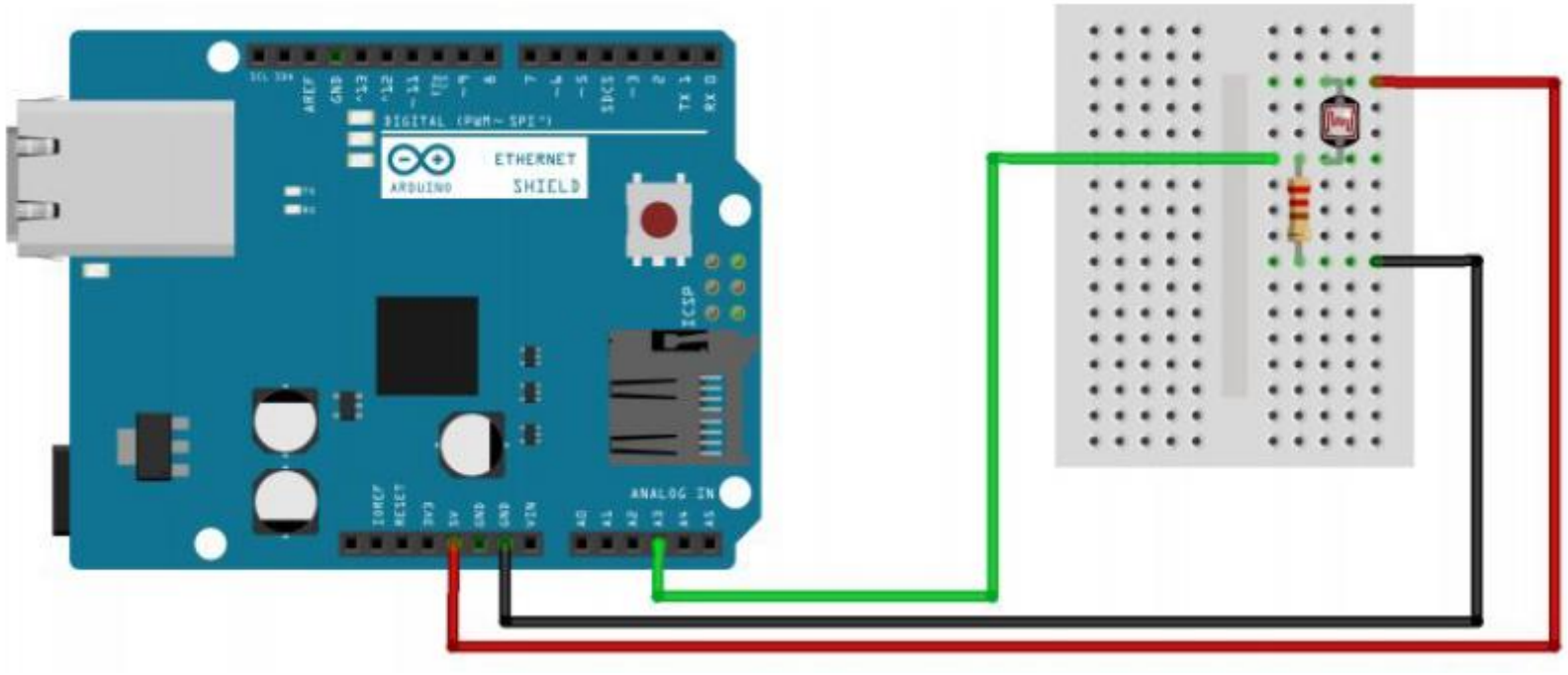
6. SD Card

- On SD cards, blocks are always 512 bytes. You can get the amount of blocks per cluster, and the amount of clusters on the card, giving you the card size, in bytes. More utility functions are listed in the example program: CardInfo. It is available in the Arduino IDE: Files ⇨ Examples ⇨ SD ⇨ CardInfo
- Example Program: For this application, you build a data-logging application. The aim is to understand how sunlight evolves during a day using LDR.



6. SD Card

- Example wiring with the Ardunio Ethernet Shield



6. SD Card

- Arduino Sketch:

```
1   #include <SD.h>
2   #include <SPI.h>
3   const int chipSelect = 4; // Change this as required
4
5   int light;
6   int lightPin = A3;
7   unsigned int iteration = 1;
8
9
10 void setup()
11 {
12   Serial.begin(9600);
13
14   Serial.print("Initializing SD card...");
15   // Chip Select pin needs to be set to output for the SD library
16   pinMode(10, OUTPUT);
17
18   // Attempt to initialize SD library
19   if (!SD.begin(chipSelect)) {
20     Serial.println("Card failed, or not present");
21     // don't do anything more:
22     return;
23   }
24   Serial.println("Card initialized.");
25 }
26
```

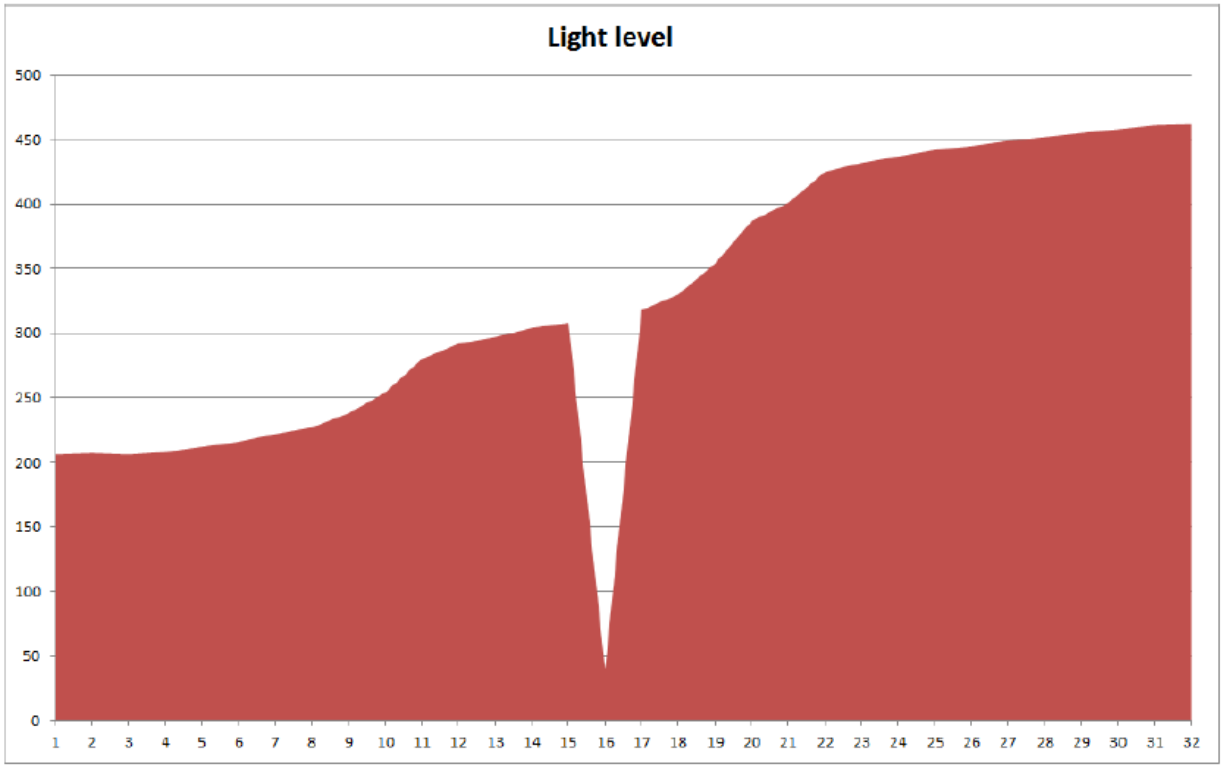
6. SD Card

- Arduino Sketch:

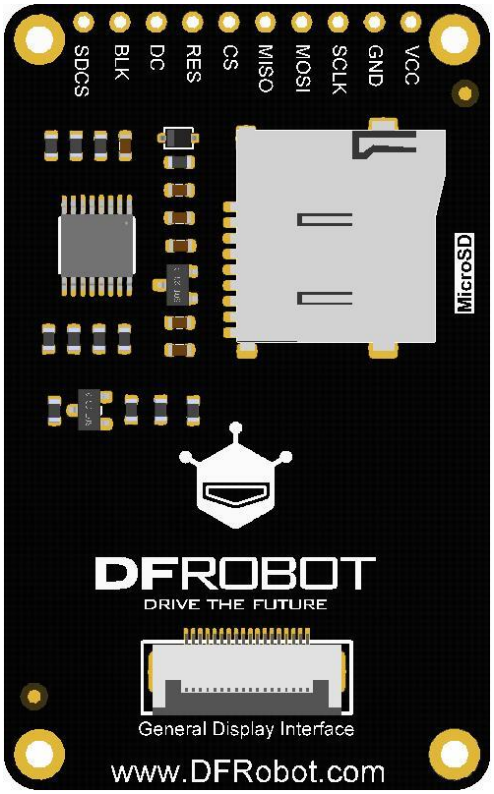
```
27 void loop()
28 {
29   // Get a light level reading
30   light = analogRead(lightPin);
31
32  // Open the SD data file
33  File dataFile = SD.open("light.txt", FILE_WRITE);
34
35  // Has the file been opened?
36  if (dataFile)
37  {
38     // Create a formatted string
39     String dataString = "";
40     dataString += String(iteration);
41     dataString += ",";
42     dataString += String(light);
43     dataString += ",";
44
45     // Print data to the serial port, and to the file
46     Serial.println(dataString);
47     dataFile.println(dataString);
48
49     // Close the file
50     dataFile.close();
51   }
52
53   // Increase the iteration number
54   iteration++;
55
56   // Sleep for one minute
57   delay(60 * 1000);
58 }
```

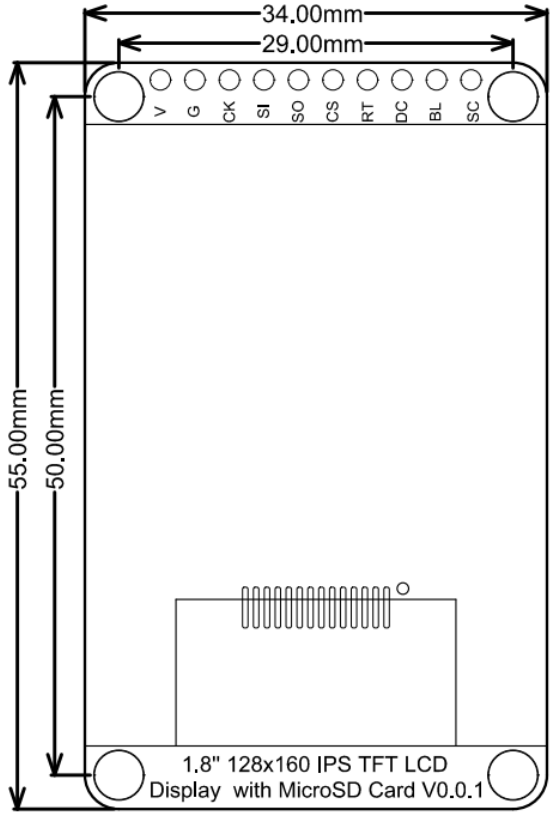
6. SD Card

- The result of this sketch creates a text fi le that can be imported into a spreadsheet. The results of a sunrise for real data logger is shown in Figure below. The ambient light level is already at 200 due to street lights, but something happened at the 16-minute mark—the visible light suddenly dropped down considerably, but only for a minute. This was probably the sensor being blocked—but it shows that surprises can happen in real-time. As a result, more measurements are required.



6. SD Card





Num	Label	Description
1	VCC	Power
2	GND	Ground
3	SCLK	Clock signal
4	MOSI	Receiving data
5	MISO	Transmitting data
6	CS	Screen Chip-select
7	RES	Reset
8	DC	Data/Command
9	BLK	Backlight
10	SDCS	SD card chip-select

- This ST7735S 1.8" TFT Display features a resolution of 128×160 and SPI (4-wire) communication. Integrated with an **SD card slot**.

6. SD Card

- https://www.arduinolibraries.info/libraries/sd-fat

SdFat

Provides access to SD memory cards.

Author	Bill Greiman
Website	https://github.com/greiman/SdFat
Category	Data Storage
License	MIT
Library Type	Contributed
Architectures	Any

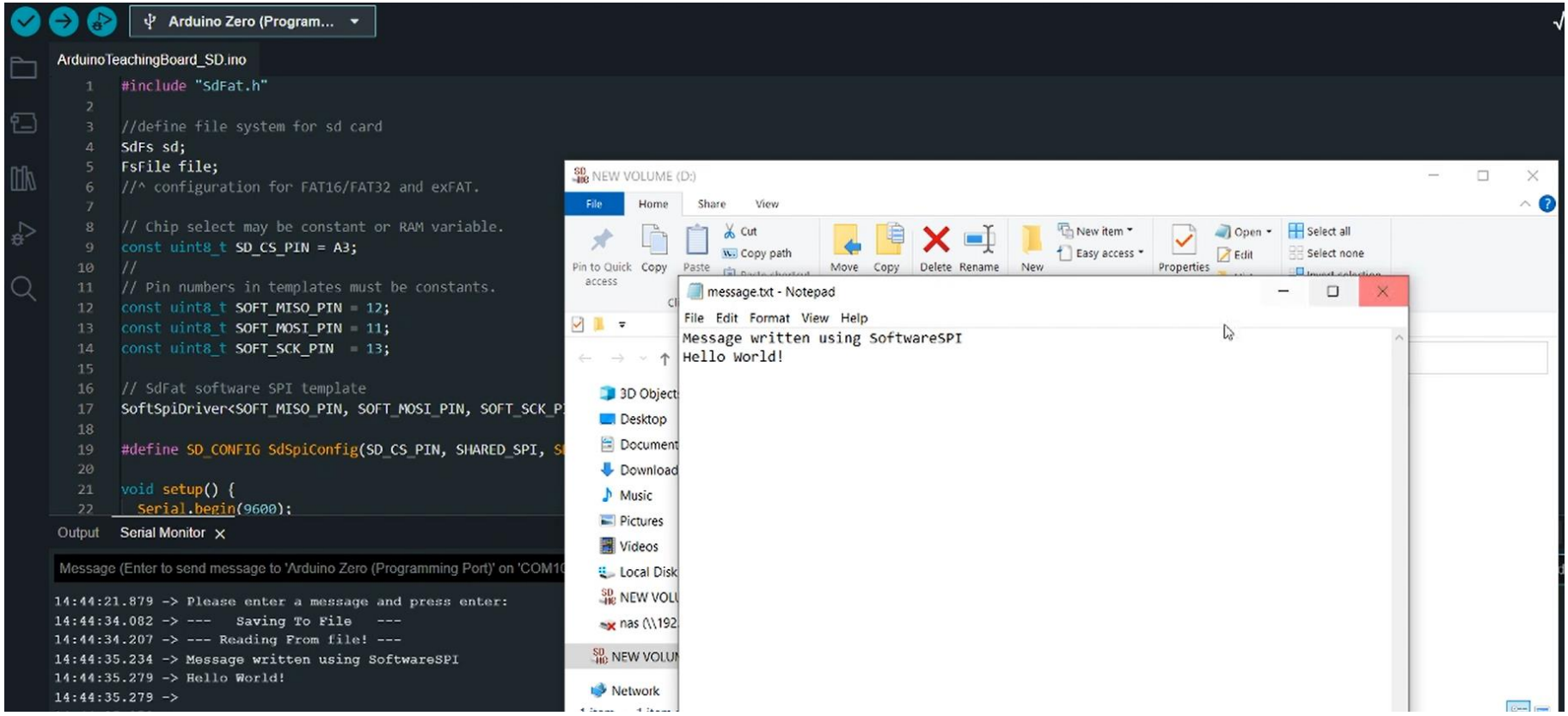
The SdFat library supports FAT16, FAT32, and exFAT file systems on Standard SD, SDHC, and SDXC cards.

6. SD Card

ArduinoTeachingBoard_SD.ino

```ino
#include "SdFat.h"

//define file system for sd card
SdFs sd;
FsFile file;
//^ configuration for FAT16/FAT32 and exFAT.

// Chip select may be constant or RAM variable.
const uint8_t SD_CS_PIN = A3;
//
// Pin numbers in templates must be constants.
const uint8_t SOFT_MISO_PIN = 12;
const uint8_t SOFT_MOSI_PIN = 11;
const uint8_t SOFT_SCK_PIN  = 13;

// SdFat software SPI template
SoftSpiDriver<SOFT_MISO_PIN, SOFT_MOSI_PIN, SOFT_SCK_PIN> softSpi;

#define SD_CONFIG SdSpiConfig(SD_CS_PIN, SHARED_SPI, SD_SCK_MHZ(0), &softSpi)

void setup() {
  Serial.begin(9600);

  if (!sd.begin(SD_CONFIG)) {
    Serial.println("SD card initialization failed!");
    sd.initErrorHalt();
    while (1);
  }

  // Open/create a file for writing
  if (!file.open("message.txt", O_RDWR | O_CREAT)) {
    sd.errorHalt(F("open failed"));
  }

  file.close(); //release file

  Serial.println("Please enter a message and press enter:");
}

void loop() {
  if (Serial.available()) {

    // Read the incoming message
    String message = Serial.readString();

    WriteSD(file, message);

    ReadSD(file);
  }
}

void WriteSD(File file, String message) {
  Serial.println("---   Saving To File   ---");

  file.open("message.txt", O_RDWR);
  file.rewind();                //Go to file position 0
  file.println("Message written using SoftwareSPI");
  file.println(message);
  file.close();
}

void ReadSD(File file) {
  Serial.println("--- Reading From file! ---");

  file.open("message.txt", O_RDWR);
  file.seek(0);                 //go to char 0
  String contents = file.readString();
  Serial.println(contents);
  file.close();
}
```

6. SD Card


Arduino Zero (Program...
ArduinoTeachingBoard_SD.ino
#include "SdFat.h"
//define file system for sd card
SdFs sd;
FsFile file;
//^ configuration for FAT16/FAT32 and exFAT.
// Chip select may be constant or RAM variable.
const uint8_t SD_CS_PIN = A3;
//
// Pin numbers in templates must be constants.
const uint8_t SOFT_MISO_PIN = 12;
const uint8_t SOFT_MOSI_PIN = 11;
const uint8_t SOFT_SCK_PIN = 13;
// SdFat software SPI template
#define SD_CONFIG SdSpiConfig(SD_CS_PIN, SHARED_SPI,
void setup() {
Serial.begin(9600);
Output
Serial Monitor
14:44:21.879 -> Please enter a message and press enter:
14:44:34.082 -> --- Saving To File ---
14:44:34.207 -> --- Reading From file! ---
14:44:35.234 -> Message written using SoftwareSPI
14:44:35.279 -> Hello World!
14:44:35.279 ->
NEW VOLUME (D:)
File
Home
Share
View
Pin to Quick access
Copy
Paste
Cut
Copy path
Move
Copy
Delete
Rename
New
New item
Easy access
Properties
Open
Edit
Select all
Select none
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk
NEW VOLU
nas (\\192
NEW VOLU
Network
message.txt - Notepad
File Edit Format View Help
Message written using SoftwareSPI
Hello World!