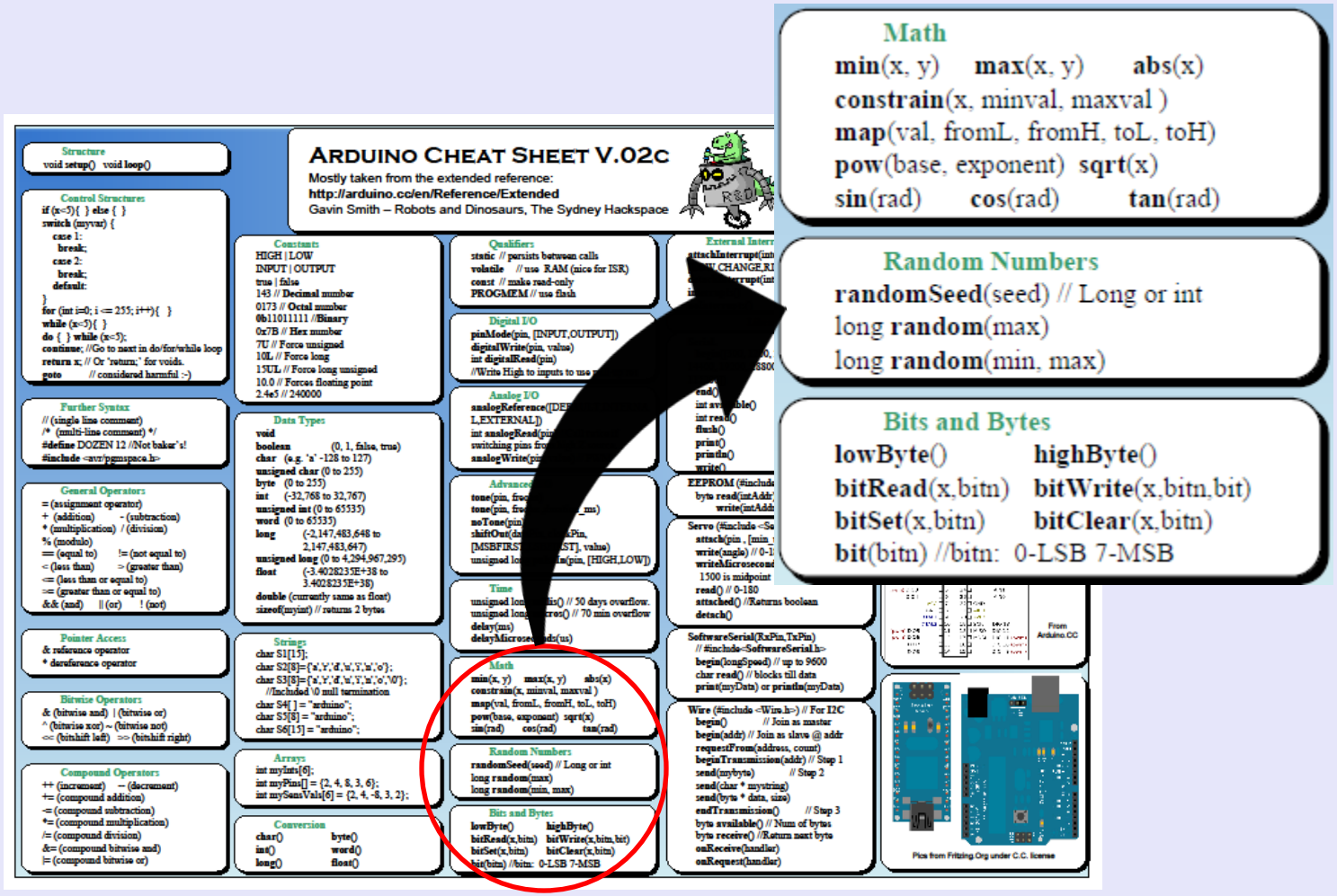
Embedded Sytems
Using Mathematical Operators

Using Mathematical Operators



Finding the Remainder After Dividing Two Values:

- **Problem:**
  - **You want to find the remainder after you divide two values.**

- **Solution:**
  - **Use the % symbol (the modulus operator) to get the remainder:**

Finding the Remainder After Dividing Two Values:

```
int myValue0 =  20 % 10;  // get the modulus(remainder) of 20 divided by 10
int myValue1 =  21 % 10;  // get the modulus(remainder) of 21 divided by 10
```

myValue0 equals 0 (20 divided by 10 has a remainder of 0). myValue1 equals 1 (21 divided by 10 has a remainder of 1).

Here is a similar example, but by using 2 with the modulus operator, the result can be used to check if a value is odd or even:

```
int myValue;
//... code here to set the value of myValue
if (myValue % 2 == 0)
{
  Serial.println("The value is even");
}
else
{
  Serial.println("The value is odd");
}
```

Determining the Absolute Value:

- **Problem:**
  - You want to get the absolute value of a number.

- **Solution:**
  - abs(x) computes the absolute value of x. The following example takes the absolute value of the difference between readings on two analog input ports (see Chapter 5 for more on analogRead()):

Determining the Absolute Value:

```
int x = analogRead(0);
int y = analogRead(1);

if (abs(x-y) > 10)
{
  Serial.println("The analog values differ by more than 10");
}
```

Discussion

`abs(x-y);` returns the absolute value of the difference between x and y. It is used for integer (and long integer) values. To return the absolute value of floating-point values, see Recipe 2.3.

Constraining a Number to a Range of Values:

- **Problem:**
  - You want to ensure that a value is always within some lower and upper limit.
- **Solution:**
  - constrain(x, min, max) returns a value that is within the bounds of min and max:

```
myConstrainedValue = constrain(myValue, 100, 200);
```

Constraining a Number to a Range of Values:

Discussion

myConstrainedValue is set to a value that will always be greater than or equal to 100 and less than or equal to 200. If myValue is less than 100, the result will be 100; if it is more than 200, it will be set to 200.

Table 3-1 shows some example output values using a min of 100 and a max of 200.

Table 3-1. Output from constrain with min = 100 and max = 200

myValue (the input value)	constrain(myValue, 100, 200)
99	100
100	100
150	150
200	200
201	200

Finding the Minimum or Maximum of Some Values:

- **Problem:**
  - You want to find the minimum or maximum of two or more values.
- **Solution:**
- min(x,y) returns the smaller of two numbers. max(x,y) returns the larger of two numbers:

```
myValue = analogRead(0);
myMinValue = min(myValue, 200); // myMinValue will be the smaller of myVal or 200
myMaxValue = max(myValue, 100); // myMaxValue will be the larger of myVal or 100
```

Finding the Minimum or Maximum of Some Values:

Discussion

Table 3-2 shows some example output values using a `min` of 200. The table shows that the output is the same as the input (`myValue`) until the value becomes greater than 200.

Table 3-2. Output from min(myValue, 200)

myValue (the input value)	min(myValue, 200)
99	99
100	100
150	150
200	200
201	200

Finding the Minimum or Maximum of Some Values:

Table 3-3 shows the output using a max of 100. The table shows that the output is the same as the input (myValue) when the value is greater than or equal to 100.

Table 3-3. Output from max(myValue, 100)

myValue (the input value)	max(myValue, 100)
99	100
100	100
150	150
200	200
201	201

Raising a Number to a Power:

- **Problem:**
  - You want to raise a number to a power.

- **Solution:**
  - pow(x, y) returns the value of x raised to the power of y:

```
myValue = pow(3,2);
// This calculates 3^2, so myValue will equal 9.
```

Raising a Number to a Power:

Discussion

The pow function can operate on integer or floating-point values and it returns the result as a floating-point value:

```
Serial.print(pow(3,2)); // this prints 9.00
int z = pow(3,2);
Serial.println(z);      // this prints 9
```

The first output is 9.00 and the second is 9; they are not exactly the same because the first print displays the output as a floating-point number and the second treats the value as an integer before printing, and therefore displays without the decimal point.

Here is an example of raising a number to a fractional power:

```
float s = pow(2, 1.0 / 12); // the twelfth root of two
```

The twelfth root of two is the same as 2 to the power of 0.083333. The resultant value, s, is 1.05946 (this is the ratio of the frequency of two adjacent notes on a piano).

Taking the Square Root:

- **Problem:**
  - You want to calculate the square root of a number.

- **Solution**
  - The sqrt(x) function returns the square root of x:

```
Serial.print( sqrt(9) ); // this prints 3.00
```

Rounding Floating-Point Numbers Up and Down:

- **Problem:**
  - You want the next smallest or largest integer value of a floating-point number (floor or ceil).

- **Solution:**
  - floor(x) returns the largest integer value that is not greater than x. ceil(x) returns the smallest integer value that is not less than x.

Rounding Floating-Point Numbers Up and Down:

Discussion

These functions are used for rounding floating-point numbers; use `floor(x)` to get the largest integer that is not greater than `x`. Use `ceil` to get the smallest integer that is greater than `x`.

Here is some example output using `floor`:

```
Serial.println( floor(1) );    // this prints  1.00
Serial.println( floor(0) );    // this prints  0.00
Serial.println( floor(.1) );   // this prints  0.00
Serial.println( floor(-1) );   // this prints -1.00
Serial.println( floor(-1.1) ); // this prints -2.00
```

Rounding Floating-Point Numbers Up and Down:

Here is some example output using ceil:

```
Serial.println( ceil(1) );     // this prints  1.00
Serial.println( ceil(1.1) );   // this prints  2.00
Serial.println( ceil(0) );     // this prints  0.00
Serial.println( ceil(.1) );    // this prints  1.00
Serial.println( ceil(-1) );    // this prints -1.00
Serial.println( ceil(-1.1) );  // this prints -1.00
```

You can round to the nearest integer as follows:

```
if (floatValue  > 0.0)
  result = floor(floatValue + 0.5);
else
  result = ceil(num - 0.5);
```

Using Trigonometric Functions:

- **Problem:**
  - You want to get the sine, cosine, or tangent of an angle given in radians or degrees.

- **Solution:**
  - sin(x) returns the sine of angle x.
  - cos(x) returns the cosine of angle x.
  - tan(x) returns the tangent of angle x.

Using Trigonometric Functions:

Discussion

Angles are specified in radians and the result is a floating-point number (see Recipe 2.3). The following example illustrates the trig functions:

```
float deg = 30;                 // angle in degrees
float rad  = deg * PI / 180;    // convert to radians
Serial.println(rad);            // print the radians
Serial.println (sin(rad));      // print the sine
Serial.println (cos(rad));      // print the cosine
```

This converts the angle into radians and prints the sine and cosine. Here is the output with annotation added:

```
0.52   30 degrees is 0.5235988 radians, print only shows two decimal places
0.50   sine of 30 degrees is .5000000, displayed here to two decimal places
0.87   cosine is .8660254, which rounds up to 0.87
```

Although the sketch calculates these values using the full precision of floating-point numbers, the `Serial.print` routine shows the values of floating-point numbers to two decimal places.

The conversion from radians to degrees and back again is textbook trigonometry. PI is the familiar constant for π (3.14159265...). PI and 180 are both constants, and Arduino provides some precalculated constants you can use to perform degree/radian conversions:

```
rad = deg * DEG_TO_RAD;  // a way to convert degrees to radians
deg = rad * RAD_TO_DEG;  // a way to convert radians to degrees
```

Generating Random Numbers:

- **Problem:**
  - **You want to get a random number, either ranging from zero up to a specified maximum or constrained between a minimum and maximum value you provide.**
- **Solution:**
  - **Use the random function to return a random number. Calling random with a single parameter sets the upper bound; the values returned will range from zero to one less than the upper bound:**

Generating Random Numbers:

```
random(max);      // returns a random number between 0 and max -1
```

Calling random with two parameters sets the lower and upper bounds; the values returned will range from the lower bound (inclusive) to one less than the upper bound:

```
random(min, max); // returns a random number between min and max -1
```

Discussion

Although there appears to be no obvious pattern to the numbers returned, the values are not truly random. Exactly the same sequence will repeat each time the sketch starts. In many applications, this does not matter. But if you need a different sequence each time your sketch starts, use the function randomSeed(seed) with a different seed value each time (if you use the same seed value, you'll get the same sequence). This function starts the random number generator at some arbitrary place based on the seed parameter you pass:

```
randomSeed(1234);  // change the starting sequence of random numbers.
```

Generating Random Numbers:

```
// Random
// demonstrates generating random numbers

int randNumber;

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

  // Print random numbers with no seed value
  Serial.println("Print 20 random numbers between 0 and 9");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(10);
    Serial.print(randNumber);
    Serial.print(" ");
  }
  Serial.println();
  Serial.println("Print 20 random numbers between 2 and 9");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(2,10);
    Serial.print(randNumber);
    Serial.print(" ");
  }
```

Generating Random Numbers:

```
// Print random numbers with the same seed value each time
randomSeed(1234);
Serial.println();
Serial.println("Print 20 random numbers between 0 and 9 after constant seed ");
for(int i=0; i < 20; i++)
{
  randNumber = random(10);
  Serial.print(randNumber);
  Serial.print(" ");
}

// Print random numbers with a different seed value each time
randomSeed(analogRead(0));  // read from an analog port with nothing connected
```

Generating Random Numbers:

```
  Serial.println();
  Serial.println("Print 20 random numbers between 0 and 9 after floating seed ");
  for(int i=0; i < 20; i++)
  {
    randNumber = random(10);
    Serial.print(randNumber);
    Serial.print(" ");
  }
  Serial.println();
  Serial.println();
}

void loop()
{
}
```

Generating Random Numbers:

Here is the output from this code:

```
Print 20 random numbers between 0 and 9
7 9 3 8 0 2 4 8 3 9 0 5 2 2 7 3 7 9 0 2
Print 20 random numbers between 2 and 9
9 3 7 7 2 7 5 8 2 9 3 4 2 5 4 3 5 7 5 7
Print 20 random numbers between 0 and 9 after constant seed
8 2 8 7 1 8 0 3 6 5 9 0 3 4 3 1 2 3 9 4
Print 20 random numbers between 0 and 9 after floating seed
0 9 7 4 4 7 7 4 4 9 1 6 0 2 3 1 5 9 1 1
```

If you press the reset button on your Arduino to restart the sketch, the first three lines of random numbers will be unchanged. Only the last line changes each time the sketch starts, because it sets the seed to a different value by reading it from an unconnected analog input port as a seed to the randomSeed function. If you are using analog port 0 for something else, change the argument to analogRead to an unused analog port.

Setting and Reading Bits:

- **Problem:**
  - You want to read or set a particular bit in a numeric variable.

- **Solution:**
  - Use the following functions:

    bitSet(x, bitPosition)

  - Sets (writes a 1 to) the given bitPosition of variable x

Setting and Reading Bits:

bitClear(x, bitPosition)

- **Clears (writes a 0 to) the given bitPosition of variable x**

bitRead(x, bitPosition)

- **Returns the value (as 0 or 1) of the bit at the given bitPosition of variable x**

bitWrite(x, bitPosition, value)

- **Sets the given value (as 0 or 1) of the bit at the given bitPosition of variable x**

bit(bitPosition)

- **Returns the value of the given bit position: bit(0) is 1, bit(1) is 2, bit(2) is 4, and so on.**
- **In all these functions, bitPosition 0 is the least significant (rightmost) bit**

Setting and Reading Bits:

```
// bitFunctions
// demonstrates using the bit functions


byte flags = 0; // these examples set, clear or read bits in a variable called flags.

// bitSet example
void setFlag( int flagNumber)
{
   bitSet(flags, flagNumber);
}

// bitClear example
void  clearFlag( int flagNumber)
{
   bitClear(flags, flagNumber);
}

// bitPosition example

int  getFlag( int flagNumber)
{
   return  bitRead(flags, flagNumber);
}

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

Setting and Reading Bits:

```
void loop()
{
     showFlags();
     setFlag(2);  // set some flags;
     setFlag(5);
     showFlags();
     clearFlag(2);
```

```
     showFlags();

     delay(10000); // wait a very long time
}

// reports flags that are set
void showFlags()
{
     for(int flag=0; flag < 8; flag++)
     {
       if (getFlag(flag) == true)
          Serial.print("* bit set for flag ");
       else
          Serial.print("bit clear for flag ");

       Serial.println(flag);
     }
     Serial.println();
}
```

Setting and Reading Bits:

This code will print the following:

```
bit clear for flag 0
bit clear for flag 1
bit clear for flag 2
bit clear for flag 3
bit clear for flag 4
bit clear for flag 5
bit clear for flag 6
bit clear for flag 7

bit clear for flag 0
bit clear for flag 1
* bit set for flag 2
bit clear for flag 3
bit clear for flag 4
* bit set for flag 5
bit clear for flag 6
bit clear for flag 7

bit clear for flag 0
bit clear for flag 1
bit clear for flag 2
bit clear for flag 3
bit clear for flag 4
* bit set for flag 5
bit clear for flag 6
bit clear for flag 7
```

Setting and Reading Bits:

The state of eight switches can be packed into a single 8-bit value instead of requiring eight bytes or integers. The example in this recipe's Solution shows how eight values can be individually set or cleared in a single byte.

The term *flag* is a programming term for values that store the state of some aspect of a program. In this sketch, the flag bits are read using bitRead, and they are set or cleared using bitSet or bitClear. These functions take two parameters: the first is the value to read or write (flags in this example), and the second is the bit position indicating where the read or write should take place. Bit position 0 is the least significant (rightmost) bit; position 1 is the second position from the right, and so on. So:

```
bitRead(2, 1); // returns 1 : 2 is binary 10 and bit in position 1 is 1
bitRead(4, 1); // returns 0 : 4 is binary 100 and bit in position 1 is 0
```

There is also a function called bit that returns the value of each bit position:

```
bit(0)  is equal to 1;
bit(1)  is equal to 2;
bit(2)  is equal to 4;
...
bit(7)  is equal to 128
```

Shifting Bits:

- **Problem:**
  - You need to perform bit operations that shift bits left or right in a byte, int, or long.

- **Solution:**
  - Use the << (bit-shift left) and >> (bit-shift right) operators to shift the bits of a value.

Shifting Bits:

Discussion

This fragment sets variable x equal to 6. It shifts the bits left by one and prints the new value (12). Then that value is shifted right two places (and in this example becomes equal to 3):

```
void setup()
{Serial.begin(9600);}
void loop()
{ int x=6;
  int result = x<<1; //6 shifted left once is 12
  Serial.println(result);
  result = result >>2;// 12 shifted right twice is 3
  Serial.println(result);
}
```

Shifting Bits:

Here is how this works: 6 shifted left one place equals 12, because the decimal number 6 is 0110 in binary. When the digits are shifted left, the value becomes 1100 (decimal 12). Shifting 1100 right two places becomes 0011 (decimal 3). You may notice that shifting a number left by *n* places is the same as multiplying the value by 2 raised to the power of *n*. Shifting a number right by *n* places is the same as dividing the value by 2 raised to the power of *n*. In other words, the following pairs of expressions are the same:

`x << 1` is the same as `x * 2`.
`x << 2` is the same as `x * 4`.
`x << 3` is the same as `x * 8`.
`x >> 1` is the same as `x / 2`.
`x >> 2` is the same as `x / 4`.
`x >> 3` is the same as `x / 8`.

Shifting Bits:

The Arduino controller chip can shift bits more efficiently than it can multiply and divide, and you may come across code that uses the bit shift to multiply and divide:

```
int c = (a << 1) + (b >> 2); //add (a times 2) plus ( b divided by 4)
```

The expression `(a << 1) + (b >> 2);` does not look much like `(a * 2) + (b / 4);`, but both expressions do the same thing. Indeed, the Arduino compiler is smart enough to recognize that multiplying an integer by a constant that is a power of two is identical to a shift and will produce the same machine code as the version using shift. The source code using arithmetic operators is easier for humans to read, so it is preferred when the intent is to multiply and divide.

Extracting High and Low Bytes in an int or long:

- **Problem:**
  - **You want to extract the high byte or low byte of an integer; for example, when sending integer values as bytes on a serial or other communication line.**

- **Solution:**
  - **Use lowByte(i) to get the least significant byte from an integer. Use highByte(i) to get the most significant byte from an integer.**

Extracting High and Low Bytes in an int or long:

The following sketch converts an integer value into low and high bytes:

```
//ByteOperators

int intValue = 258; // 258 in hexadecimal notation is 0x102


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

Extracting High and Low Bytes in an int or long:

```
void loop()
{
  int loWord,hiWord;
  byte loByte, hiByte;


  hiByte = highByte(intValue);
  loByte = lowByte(intValue);

  Serial.println(intValue,DEC);
  Serial.println(intValue,HEX);
  Serial.println(loByte,DEC);
  Serial.println(hiByte,DEC);

  delay(10000); // wait a very long time
}
```

Extracting High and Low Bytes in an int or long:

Discussion

The example sketch prints `intValue` followed by the low byte and high byte:

```
258     // the integer value to be converted
102     // the value in hexadecimal notation
2       // the low byte
1       // the high byte
```

To extract the byte values from a `long`, the 32-bit long value first gets broken into two 16-bit *words* that can then be converted into bytes as shown in the earlier code. At the time of this writing, the standard Arduino library did not have a function to perform this operation on a `long`, but you can add the following lines to your sketch to provide this:

```
#define highWord(w) ((w) >> 16)
#define lowWord(w) ((w) & 0xffff)
```

These are *macro expressions*: `hiWord` performs a 16-bit shift operation to produce a 16-bit value, and `lowWord` masks the lower 16 bits using the bitwise And operator (see Recipe 2.20).

Extracting High and Low Bytes in an int or long:

This code converts the 32-bit hex value 0x1020304 to its 16-bit constituent high and low values:

```
loword = lowWord(longValue);
hiword = highWord(longValue);
Serial.println(loword,DEC);
Serial.println(hiword,DEC);
```

This prints the following values:

```
772   // 772 is 0x0304 in hexadecimal
258   // 258 is 0x0102 in hexadecimal
```

Note that 772 in decimal is 0x0304 in hexadecimal, which is the low-order word (16 bits) of the `longValue 0x1020304`. You may recognize 258 from the first part of this recipe as the value produced by combining a high byte of 1 and a low byte of 2 (0x0102 in hexadecimal).

Forming an int or long from High and Low Bytes:

- **Problem:**
  - You want to create a 16-bit (int) or 32-bit (long) integer value from individual bytes; for example, when receiving integers as individual bytes over a serial communication link. This is the inverse operation of Recipe 3.14.

- **Solution:**
  - Use the word(h,l) function to convert two bytes into a single Arduino integer. Here is the code from Recipe 3.14 expanded to convert the individual high and low bytes back into an integer:

Forming an int or long from High and Low Bytes:

```
//ByteOperators

int intValue = 0x102;  // 258


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

void loop()
{
  int loWord,hiWord;
  byte loByte, hiByte;


  hiByte = highByte(intValue);
  loByte = lowByte(intValue);

  Serial.println(intValue,DEC);
  Serial.println(loByte,DEC);
  Serial.println(hiByte,DEC);

  loWord = word(hiByte, loByte);   // convert the bytes back into a word
  Serial.println(loWord,DEC);
  delay(10000); // wait a very long time
}
```

Forming an int or long from High and Low Bytes:

Arduino does not have a function to convert a 32-bit long value into two 16-bit words (at the time of this writing), but you can add your own makeLong() capability by adding the following line to the top of your sketch:

```
#define makeLong(hi, low)  ((hi) << 16 & (low))
```

This defines a command that will shift the high value 16 bits to the left and add it to the low value:

```
#define makeLong(hi, low)  (((long) hi) << 16 | (low))
#define highWord(w) ((w) >> 16)
#define lowWord(w) ((w) & 0xffff)

// declare a value to test
long longValue = 0x1020304;  // in decimal: 16909060
                             // in binary : 00000001 00000010 00000011 00000100
```

Forming an int or long from High and Low Bytes:

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

void loop()
{
  int loWord,hiWord;

  Serial.println(longValue,DEC);  // this prints  16909060
  loWord = lowWord(longValue);    // convert long to two words
  hiWord = highWord(longValue);
  Serial.println(loWord,DEC);     // print the value 772
  Serial.println(hiWord,DEC);     // print the value 258
  longValue = makeLong( hiWord, loWord);  // convert the words back to a long
  Serial.println(longValue,DEC);  // this again prints  16909060

  delay(10000); // wait a very long time
}
```

The output is:

```
16909060
772
258
16909060
```

PORT Pin and register details

ATmega328 datasheet, pp. 76-94

Figure 13-1. I/O Pin Equivalent Schematic

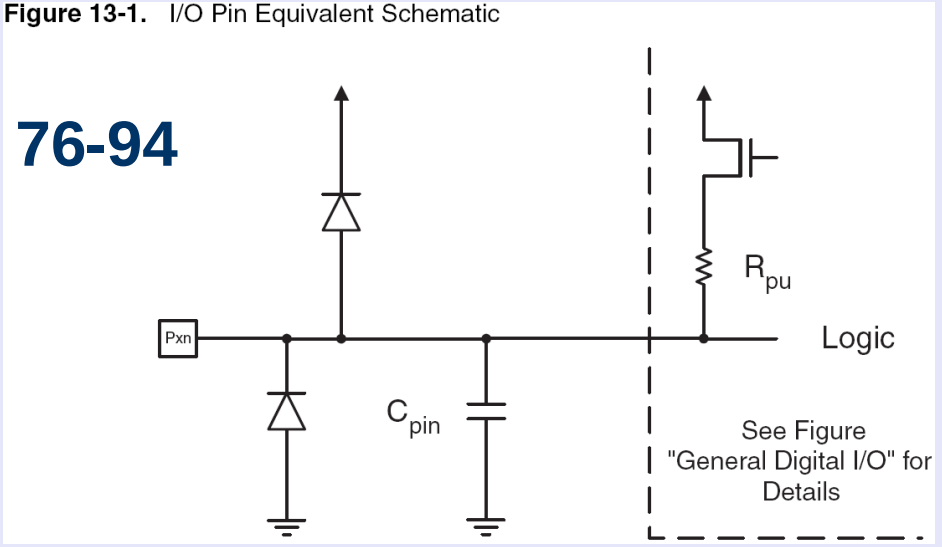
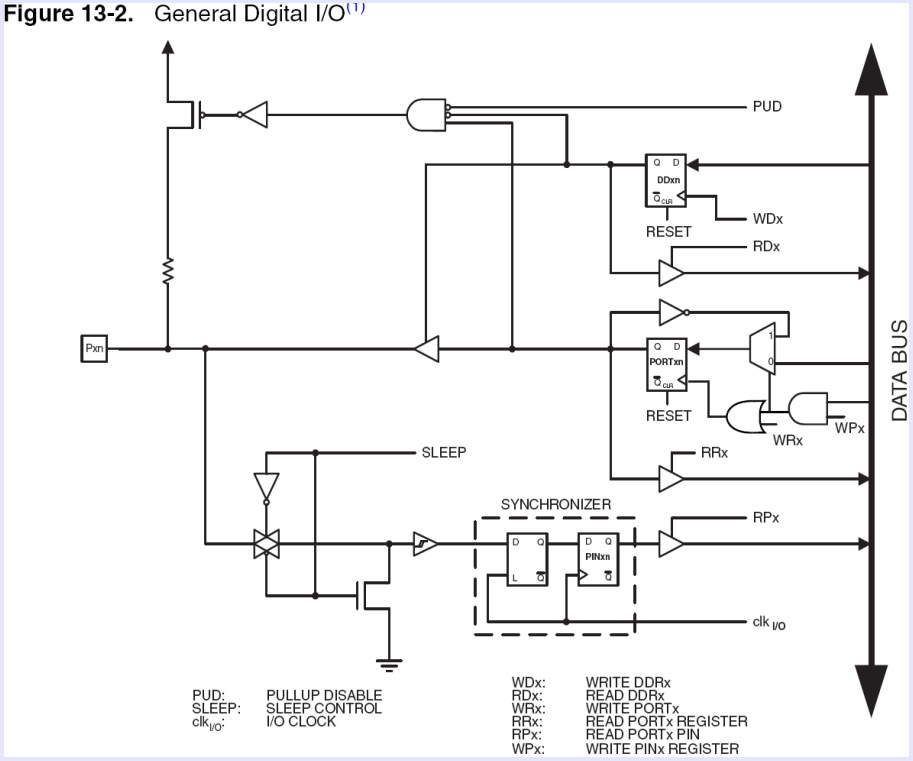


Figure 13-2. General Digital I/O[1]



PORTD – The Port D Data Register

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRD – The Port D Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PIND – The Port D Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Example 1

- Make Arduino pins 3, 5, and 7 (PD3, PD5, and PD7) to be outputs

- Arduino approach

```
pinMode(3, OUTPUT);
pinMode(5, OUTPUT);
pinMode(7, OUTPUT);
```

Or if me106.h is used:

```
pinMode(PIN_D3, OUTPUT);
pinMode(PIN_D5, OUTPUT);
pinMode(PIN_D7, OUTPUT);
```

- Alternate approach

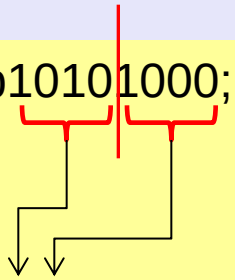
```
DDRD = 0b10101000;

or

DDRD = 0xA8;

or

DDRD | = 1<<PD7 | 1<<PD5 | 1<<PD3;
```



Arduino and the AVR