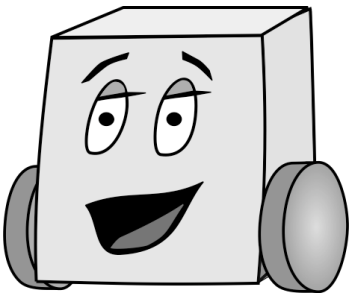

E11: Autonomous Vehicles

Fall 2010

Harris & Lape with Keeter & Ong

PS 3: Creating Gold Codes

A Gold code is a sequence of 0's and 1's commonly used in communications systems such as cell phones and the Global Positioning System (GPS). Gold codes are named for Dr. Robert Gold, who published the idea in 1967. They are popular because they are easy to generate in hardware and they are in some sense *maximally* different from one another: this way, many different Gold codes can be transmitted at the same time without confusing one for another.

In the E11 Design Competition, the beacons will be transmitting various different Gold codes using flashing LEDs. In this problem set, you will write a program to generate 5-bit Gold codes (as if you were building a beacon). In the fourth problem set, you will write a program to search for -- and identify -- Gold codes through the phototransistor.

Homework 3: Gold Code Generation ~ *and checking*

For this assignment, you will write a single Mudduino file, named `ps31.pde`, in which the important pieces will be in several functions *outside of* `setup` and `loop`.

You will want to test your code – however, you may do that however you are comfortable. We discussed some helpful functions for printing arrays and for getting single integers and arrays of integers from the keyboard in class. Feel free to use those!

However, we will be concerned with the correct operation of the following eight functions. In fact, to test your functions we will paste them into our own code (with our own testing scaffolding – based on the examples in class) and then run them. So, for this assignment, it is these eight functions that matter.

You will want to refer to Prof. Harris's Gold Code notes from the past few lectures to put these functions into context: by the end of this assignment, you should be able to verify the figure (Figure 5) at the end of lecture 4's notes: it shows that – regardless of wrapping-around – the correlation between any two different gold codes is low (no bigger than 9) relative to the maximum correlation possible between the *same* two gold codes. After all, if they're the same, the maximum correlation is 31! (but not 31 factorial)

Here are the eight functions you should implement in ps31.pde:

Function #0

This doesn't count as one of the 8 functions! But I'd ***strongly*** advise you to start with the code on the E11 programming hints page, which provides a way to input and output arrays easily!

Try it out!

Function #1

For this part, write (***and test***) a function whose signature (first line) is

```
void XOR_arrays( int A[], int B[], int C[], int LEN )
{
  // your code here
}
```

This **XOR_arrays** function should take in three arrays of equal length, named A, B, and C. It should also take in an integer named LEN, which is the length of all three arrays.

This function does not return anything, but it should place into array C (destroying any of C's existing contents) the result of XORing the individual elements of arrays A and B.

Be sure to check your function before moving on!

Function #2

For this part, write (***and test***) a function whose signature (first line) is

```
void shift_taps35_dest( int A[], int LEN )
{
  // your code here
}
```

This **shift_taps35_dest** function should take in one array named A whose length is LEN, which is the second input.

This function does not return anything, but it should destructively change the array A so that each element has shifted one place to the right. In addition, the new zeroth element should be the XOR of the ***original*** values of the third and fifth element of A. Note that these are actually A[2] and A[4]!!

Be sure to check your function before moving on!

Function #3

For this part, write (***and test***) a function whose signature is

```
void shift_taps2345_dest( int A[], int LEN )
{
  // your code here
}
```

This **shift_taps2345_dest** function should take in one array named A whose length is LEN, which is the second input.

This function does not return anything, but it should destructively change the array A so that each element has shifted one place to the right. In addition, the new zeroth element should be the XOR of the ***original*** values of the second, third, fourth, and fifth element of A. Note that the indices will be one smaller!

Be sure to check your function before moving on!

Function #4

For this part, write (***and test***) a function whose signature is

```
void wrap_dest( int A[], int LEN )
{
  // your code here
}
```

This **wrap_dest** function should take in one array named A whose length is LEN, which is the second input.

This function does not return anything, but it should destructively change the array A so that each element has shifted one place to the right. In addition, the new zeroth element should be the old value of the last element of A.

Be sure to check your function before moving on!

Function #5

For this part, write (***and test***) a function whose signature is

```
void mlsrs_taps35( int A[], int LEN, int MLSRS[], int MLSRS_LEN )
{
  // your code here
}
```

This **mlsrs_taps35** function should take in one array named A – this will represent the ***seed*** of the shift-register sequence. The length of A is LEN, which is the second input.

The third input to **mlsrs_taps35** should be an array named MLSRS, which stands for maximum-length shift-register sequence. The fourth input should be **`MLSRS_LEN`**, which should be the length of MLSRS.

You will probably only call this function with the second input having the value of 5 and the fourth input having the value of 31. However, it will be general-purpose enough to handle other values… .

This function does not return anything, but it should destructively change the array MLSRS so that each element is the value of the "last column" of the shift-register sequence, seeded with the array A and using the taps 3 and 5. You may want to re-read the description of this process in Lecture 4's notes.

You may alter the array A in the process of creating the MLSRS – so, don't worry that A will get shifted throughout that process. Note that you'll want to use shift_taps35_dest here!

Also, your function should print out the shift register sequence as it computes the MLSRS – this will result in a table similar to the Table 2 in Lecture 4's notes.

Be sure to check your function before moving on!

Function #6

For this part, write (***and test***) a function whose signature is

```
void mlsrs_taps2345( int A[], int LEN, int MLSRS[], int MLSRS_LEN )
{
  // your code here
}
```

This **mlsrs_taps2345** function should be basically the same as the previous function, except that it should use the taps 2, 3, 4, and 5.

Again, your function should print out the shift register sequence as it computes the MLSRS – this will result in a table different to the Table 2 in Lecture 4's notes, but you will be able to check your MLSRS by comparing to the value in those notes.

Check your function by generating the two gold codes that appear in Lecture 4's notes. You'll want to use XOR_arrays for this!

As a reminder, here are those two gold codes, taken from those notes:

A Gold code is produced by XORing two different maximal length shift register sequences produced with different taps.

In this class, we will use the taps we have just studied (one sequence using $1 + x^3 + x^5$, and the other using $1 + x^2 + x^3 + x^4 + x^5$). If both sequences are produced using seeds of 00001, the Gold code is

```
    10000100101100111111000110111010 (1 + x^3 + x^5 seed 00001)
xor 10000101101010001111011111001001 (1+x^2+x^3+x^4+x^5 seed 00001)
    00000001000110110000011001110011
```

Let us call this 31-bit sequence GC($1 + x^3 + x^5$, $1 + x^2 + x^3 + x^4 + x^5$, 00001).

An entire family of Gold codes is produced by shifting the second sequence. For example another Gold code is produced by shifting the second sequence right by 1 (which is equivalent to seeding with 00011):

```
    10000100101100111111000110111010 (1 + x^3 + x^5 seed 00001)
xor 11000010110101000111011111100100 (1+x^2+x^3+x^4+x^5 seed 00011)
    01000110011001111001010010111110
```

We will call this sequence GC($1 + x^3 + x^5$, $1 + x^2 + x^3 + x^4 + x^5$, 00011).

Congratulations! You've generated the Gold codes your robot will be using… .

Function #7

For this part, write (***and test***) a function whose signature is

```
int correlate( int A[], int B[], int LEN )
{
  // your code here
}
```

This **correlate** function should take in two arrays of equal length (A, B, and LEN, respectively). Then, correlate should return the value of the correlation between the bits in arrays A and B as described in Lecture 4's notes. To summarize, for each corresponding pair of elements in A and B, you should ***add one*** to the final correlation value if those elements are equal and you should ***subtract one*** if they are unequal

Be sure to check your function before moving on! Lecture 4's notes have example data on which you can test!

Function #8

For this part, write (***and test***) a function whose signature is

```
void correlateGoldCodes( int GC1[], int GC2[], int LEN )
{
  // your code here
}
```

This **correlateGoldCodes** function should take in two arrays, GC1 and GC2, of equal length, which is the third input, LEN.

This function does not need to return anything. Rather, it should do the following:

- Compute the correlation between GC1 and each possible shift of GC2
- Print out the shift value and the correlation at each shift

It is OK to change the contents of one or both of the gold-code arrays (this helps with the shifting).

Here is one example of appropriate output – this is simply a textual version of the graph in Figure 5 of Lecture 4's notes:

```
Correlation of shift 0 is -1
Correlation of shift 1 is -1
Correlation of shift 2 is 7
Correlation of shift 3 is 7
Correlation of shift 4 is -1
Correlation of shift 5 is -9
Correlation of shift 6 is -1
Correlation of shift 7 is 7
Correlation of shift 8 is -1
Correlation of shift 9 is -9
Correlation of shift 10 is -9
Correlation of shift 11 is -1
Correlation of shift 12 is 7
Correlation of shift 13 is -9
Correlation of shift 14 is -1
Correlation of shift 15 is 7
Correlation of shift 16 is 7
Correlation of shift 17 is 7
Correlation of shift 18 is -9
Correlation of shift 19 is -9
```

```
Correlation of shift 20 is 7
Correlation of shift 21 is 7
Correlation of shift 22 is -1
Correlation of shift 23 is -9
Correlation of shift 24 is -9
Correlation of shift 25 is 7
Correlation of shift 26 is 7
Correlation of shift 27 is -9
Correlation of shift 28 is -9
Correlation of shift 29 is 7
Correlation of shift 30 is 7
```

Congratulations! You have now verified (perhaps not quite *proven*) the property that makes Gold Codes so useful.

In the next problem set, you'll use your correlation and XOR functions to determine the flash-patterns of LED-based Gold Code transmissions.

Deliverables

You are responsible for turning in one Arduino file:

- `ps31.pde`