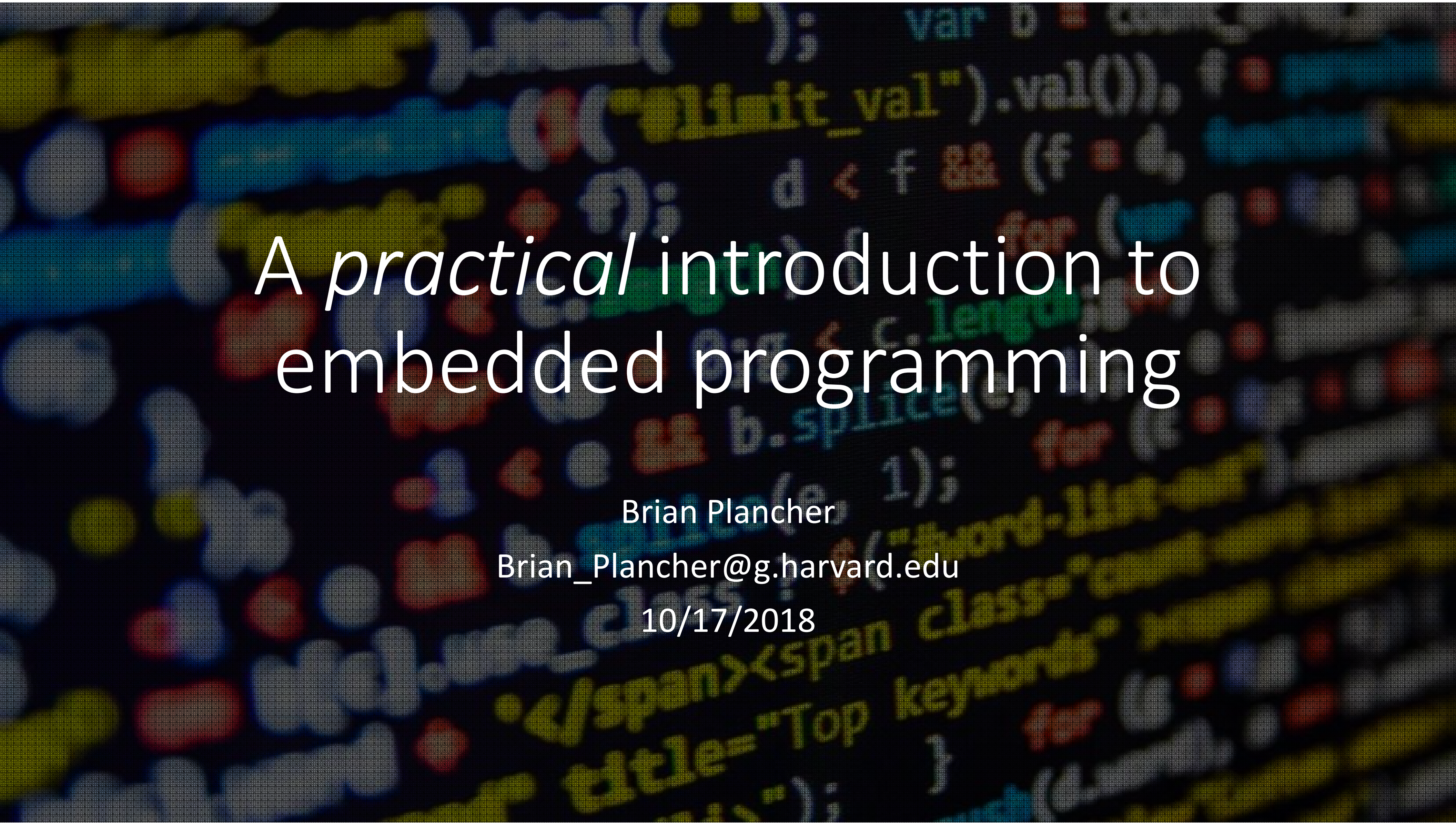


A *practical* introduction to embedded programming

Brian Plancher

Brian_Plancher@g.harvard.edu

10/17/2018

This week's task is simple:

1. Since the boards you made 2 weeks ago are perfect and are still in perfect shape and are totally programmable…
2. And since you already know how to code in C…
3. Write some custom code to test a function on your board!... You did make sure that you can programmatically change the button and/or LED right (aka they are connected to PAx)?

This week’s task is simple:

1. Since the boards you made 2 weeks ago are perfect
   and ar
   tally
   progra
2. And si
   de in C…
3. Write
   tion on your
   board!… You did make sure that you can programmatically change the button and/or LED right (aka they are connected to PAx)?

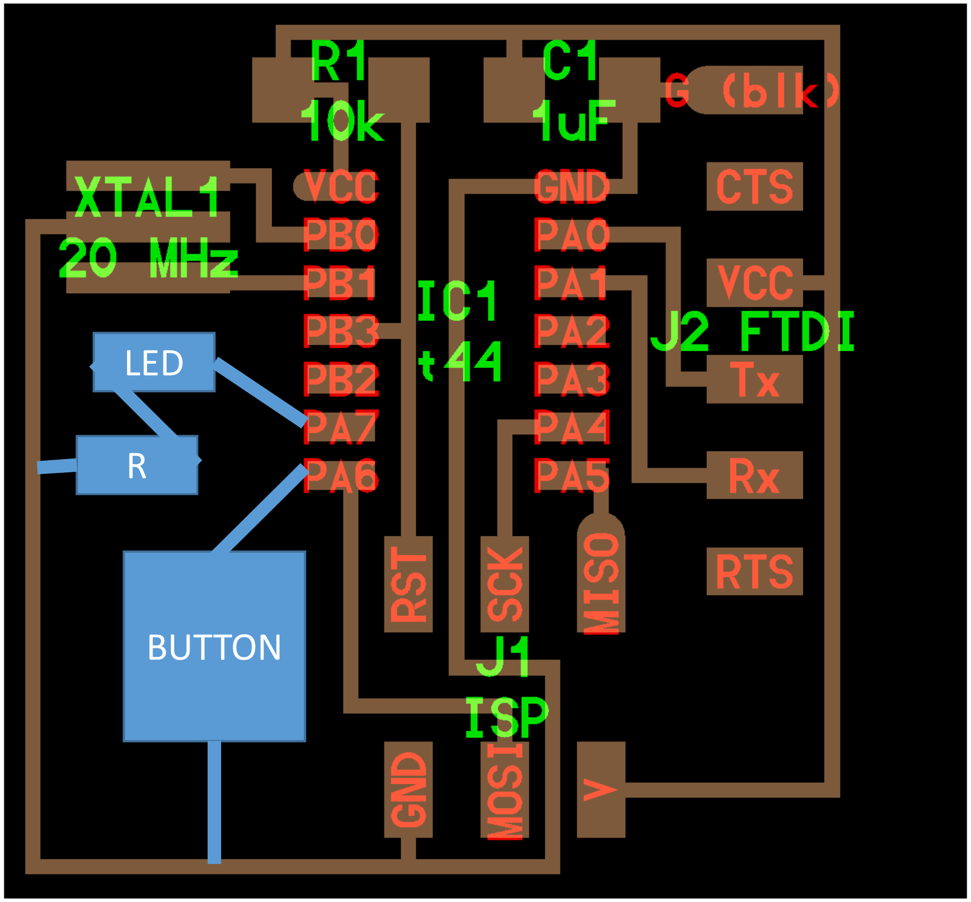
So as I said two weeks ago… if you are feeling like…

I HAVE NO IDEA
WHAT I'M DOING

RELAX
WE GOT THIS
memegen.com

One quick aside on boards before we talk about coding…

If you are goin to end up re-doing your board this is a really solid way to do it:



Now onto coding in AVR-C!

So if your first thought is: “What are codes”

Now onto coding in AVR-C!

So if your first thought is: “What are codes”

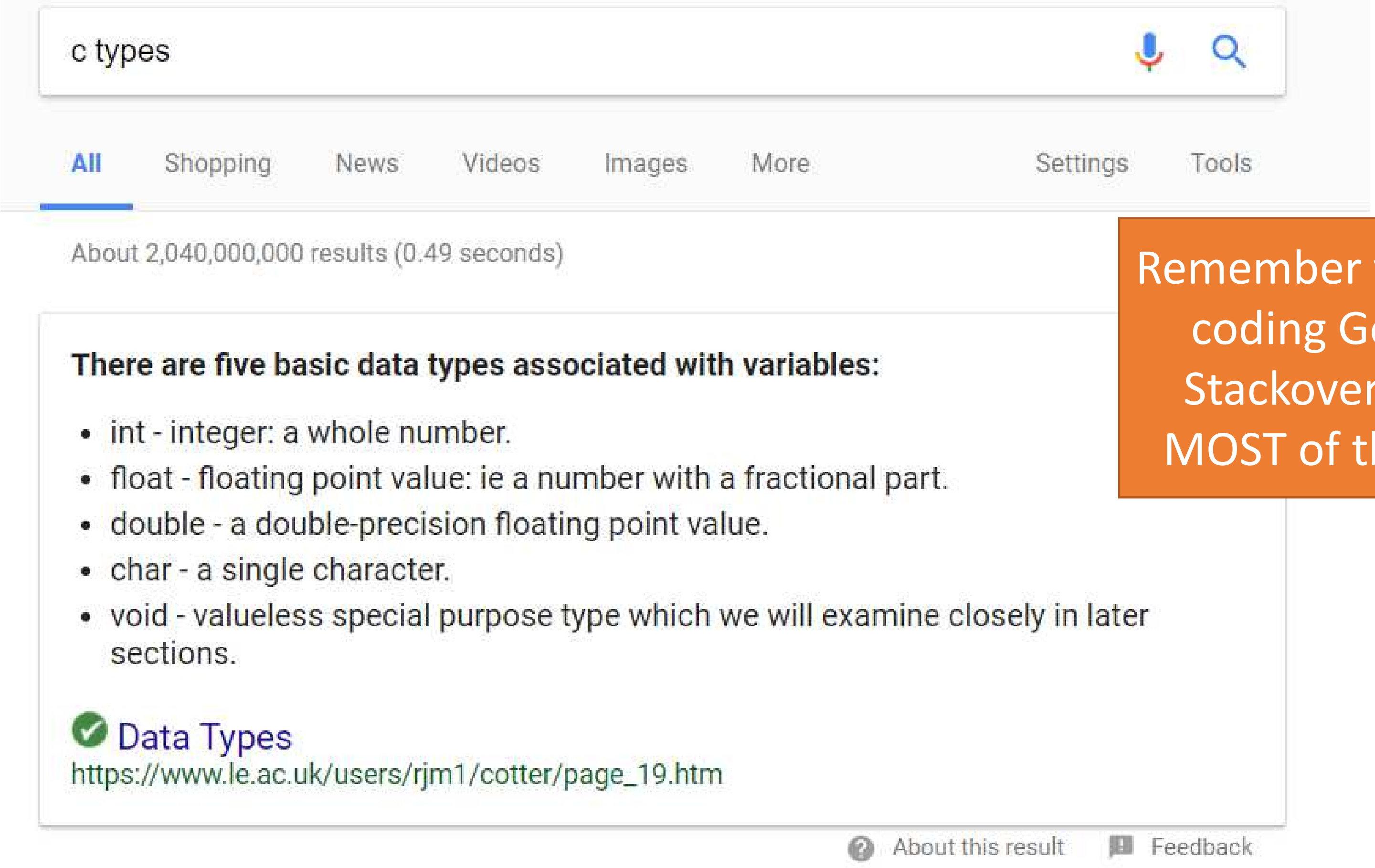
In short, **computer code is a human-readable language which tells the computer what to do**. The beauty of coding languages is that someone else wrote a *compiler* which translates the human readable words into 1s and 0s for the computer. The rules of a coding language are the assumptions the compiler makes during translation to ensure it gets it right!

Now onto coding in AVR-C!

So if your first thought is: “What is AVR-C? I feel like I should start with A…”

C is at this point the foundational language upon which most modern languages are based (or designed to be improvements on). AVR-C is a set of specific extensions to C to allow you to program your Attinys.

There are 5 basic datatypes you can use in C



About 2,040,000,000 results (0.49 seconds)

There are five basic data types associated with variables:

- int - integer: a whole number.
- float - floating point value: ie a number with a fractional part.
- double - a double-precision floating point value.
- char - a single character.
- void - valueless special purpose type which we will examine closely in later sections.

✓ Data Types
https://www.le.ac.uk/users/rjm1/cotter/page_19.htm

Remember for all things coding Google and Stackoverflow have MOST of the answers

You assign Variables (aka specific named instances of a type) to hold data

```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
```

You assign Variables (aka specific named instances of a type) to hold data

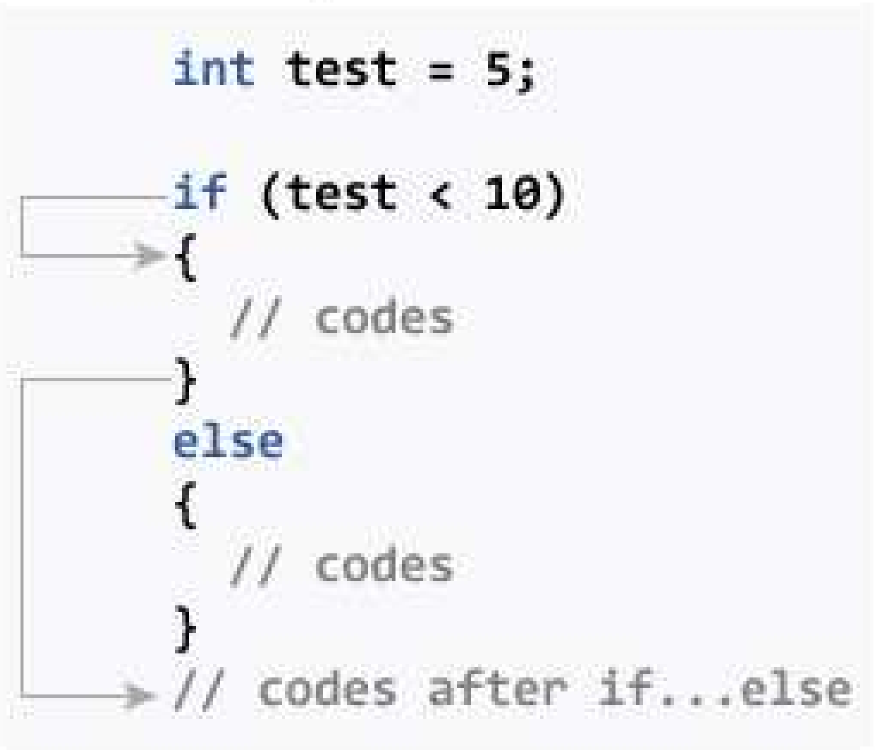
```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
```

Almost everything ends in semicolons in C!

Don’t forget them!

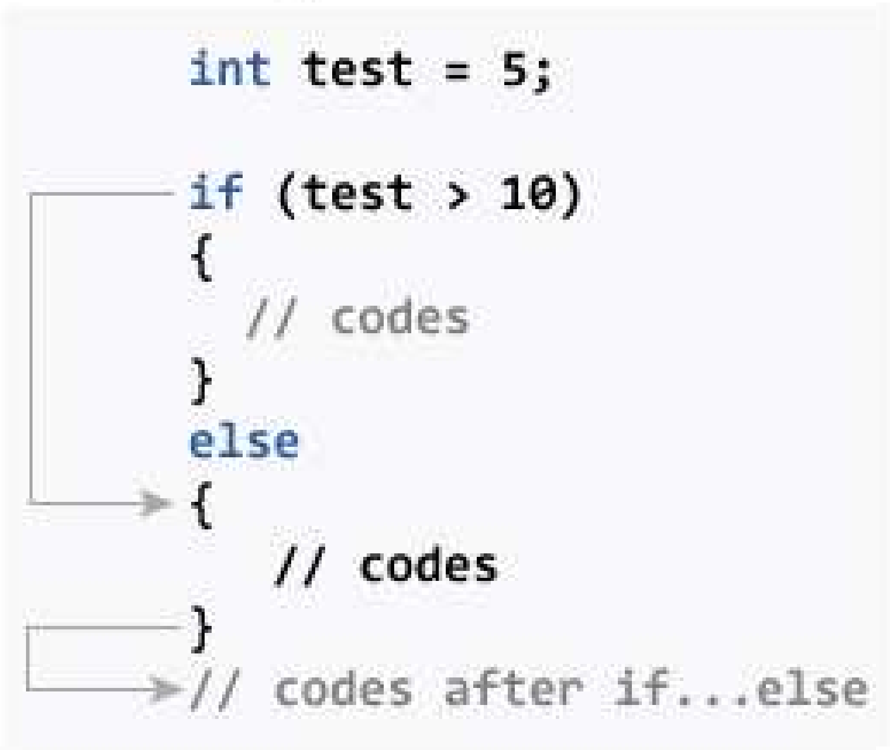
You can then use conditional statements to make decisions about what to do with data

Test expression is true



```
int test = 5;

if (test < 10)
{
  // codes
}
else
{
  // codes
}
// codes after if...else
```

Test expression is false



```
int test = 5;

if (test > 10)
{
  // codes
}
else
{
  // codes
}
// codes after if...else
```

You can then use conditional statements to make decisions about what to do with data

```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
int above_drinking_age;
If (age > 21){
   above_drinking_age = 1;
} else {
   above_drinking_age = 0;
}
```

You can then use conditional statements to make decisions about what to do with data

```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
int above_drinking_age;
If (age > 21){
   above_drinking_age = 1;
} else {
   above_drinking_age = 0;
}
```

All if and else statements need the {} around them!

You can create functions to encapsulate some operate which you use a lot

```
int checkID(int age){
  If (age > 21){
     return 1;
  } else {
     return 0;
  }
}
```

```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
int above_drinking_age = checkID(my_age);
```

You can create functions to encapsulate some operate which you use a lot

```
int checkID(int age){
 If (age < 21){
   return 1;
 } else {
   return 0;
 }
}
```

```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
int above_drinking_age = checkID(my_age);
```

When you **call** a function you need to pass in the variables which it will use

You can create functions to encapsulate some operate which you use a lot

```
int checkID(int age){
  if (age < 21){
    return 1;
  } else {
    return 0;
  }
}
```

```
int my_age = 27;
char first_initial = 'B';
char last_initial = 'P';
int above_drinking_age = checkID(my_age);
```

You also need to specify the **return type** for the function and then make sure to return the appropriate thing

When you **call** a function you need to pass in the variables which it will use

Finally you use loops to repetitively call the same set of actions

```
int class_ages[3];
```

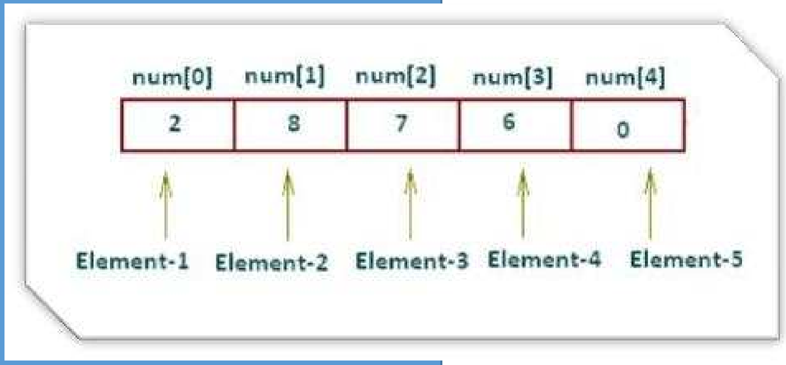
This is an **ARRAY** which is a list of some type. In this case it is 3 ints.

Finally you use loops to repetitively call the same set of actions

```
int class_ages[3];
class_ages[0] = 17;
class_ages[1] = 21;
class_ages[2] = 54;
```

This is an **ARRAY** which is a list of some type. In this case it is 3 ints.

It is zero-index!



Finally you use loops to repetitively call the same set of actions

```
int class_ages[3];
class_ages[0] = 17;
class_ages[1] = 21;
class_ages[2] = 54;
int index = 0;
while (index < 3){
  if (checkID(class_ages[index])){
    letIntoBar();
  }
  index = index + 1;
}
```

We can use a **WHILE LOOP** to iterate until we hit the condition

Finally you use loops to repetitively call the same set of actions

```
int class_ages[3];
class_ages[0] = 17;
class_ages[1] = 21;
class_ages[2] = 54;
int index = 0;
while (index < 3){
  if (checkID(class_ages[index])){
    letIntoBar();
  }
  index++;
}
```

We can use a **WHILE LOOP** to iterate until we hit the condition

We can shorthand
index = index + 1;
to:
index+=1;
or:
Index++;

Finally you use loops to repetitively call the same set of actions

```
int class_ages[3];
class_ages[0] = 17;
class_ages[1] = 21;
class_ages[2] = 54;
int index = 0;
while (index < 3){
  if (checkID(class_ages[index])){
    letIntoBar();
  }
  index++;
}
```

We can use a **WHILE LOOP** to iterate until we hit the condition

We can shorthand
index = index + 1;
to:
index+=1;
or:
Index++;

DON'T FORGET THE ++

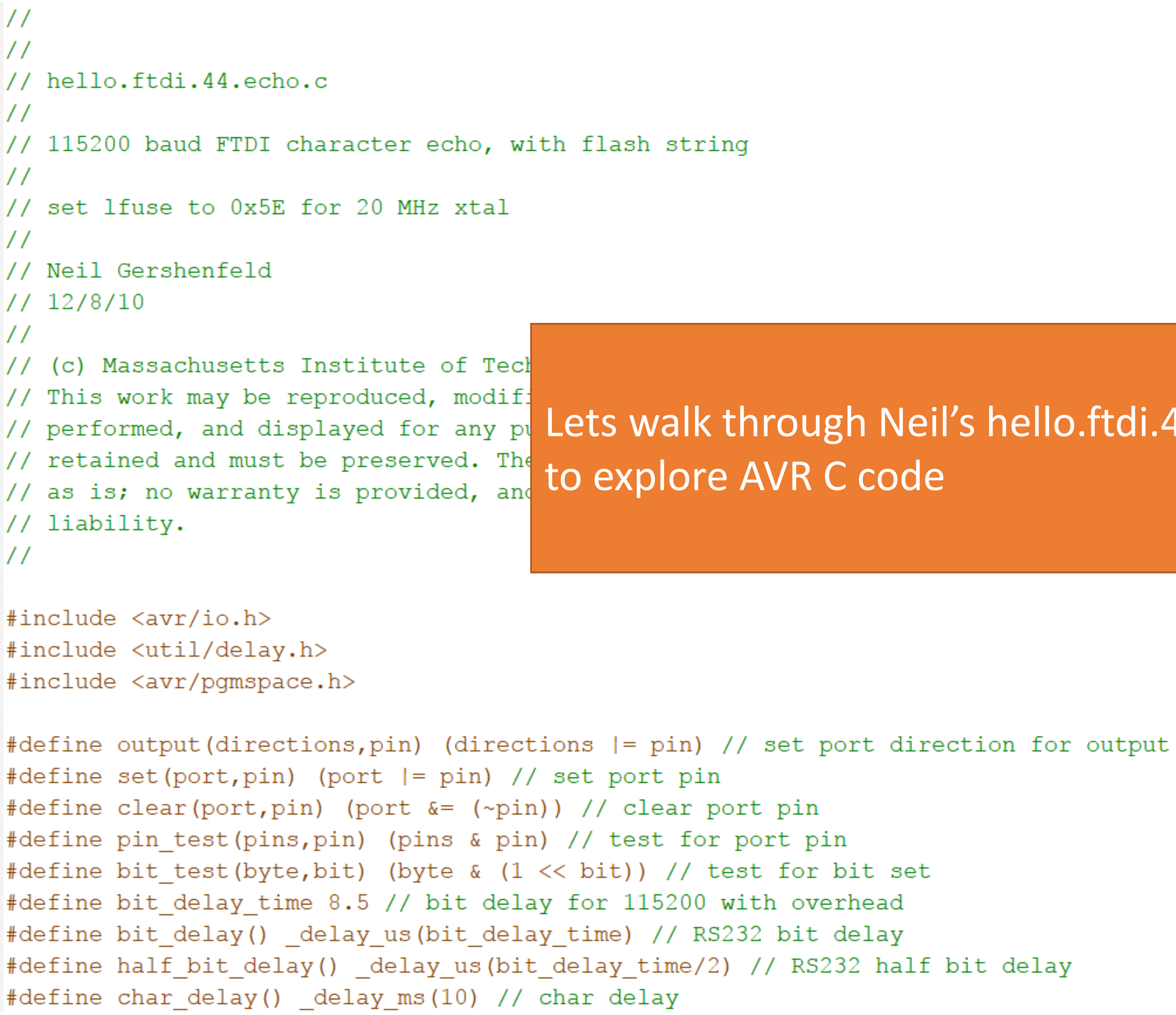
Finally you use loops to repetitively call the same set of actions

```
int class_ages[3];
class_ages[0] = 17;
class_ages[1] = 21;
class_ages[2] = 54;
for (int index = 0; index < 3; index++){
   if (checkID(class_ages[index])){
      letIntoBar();
   }
}
```

We can use a **FOR LOOP** to shorthand the while loop and make sure we don’t forget the ++

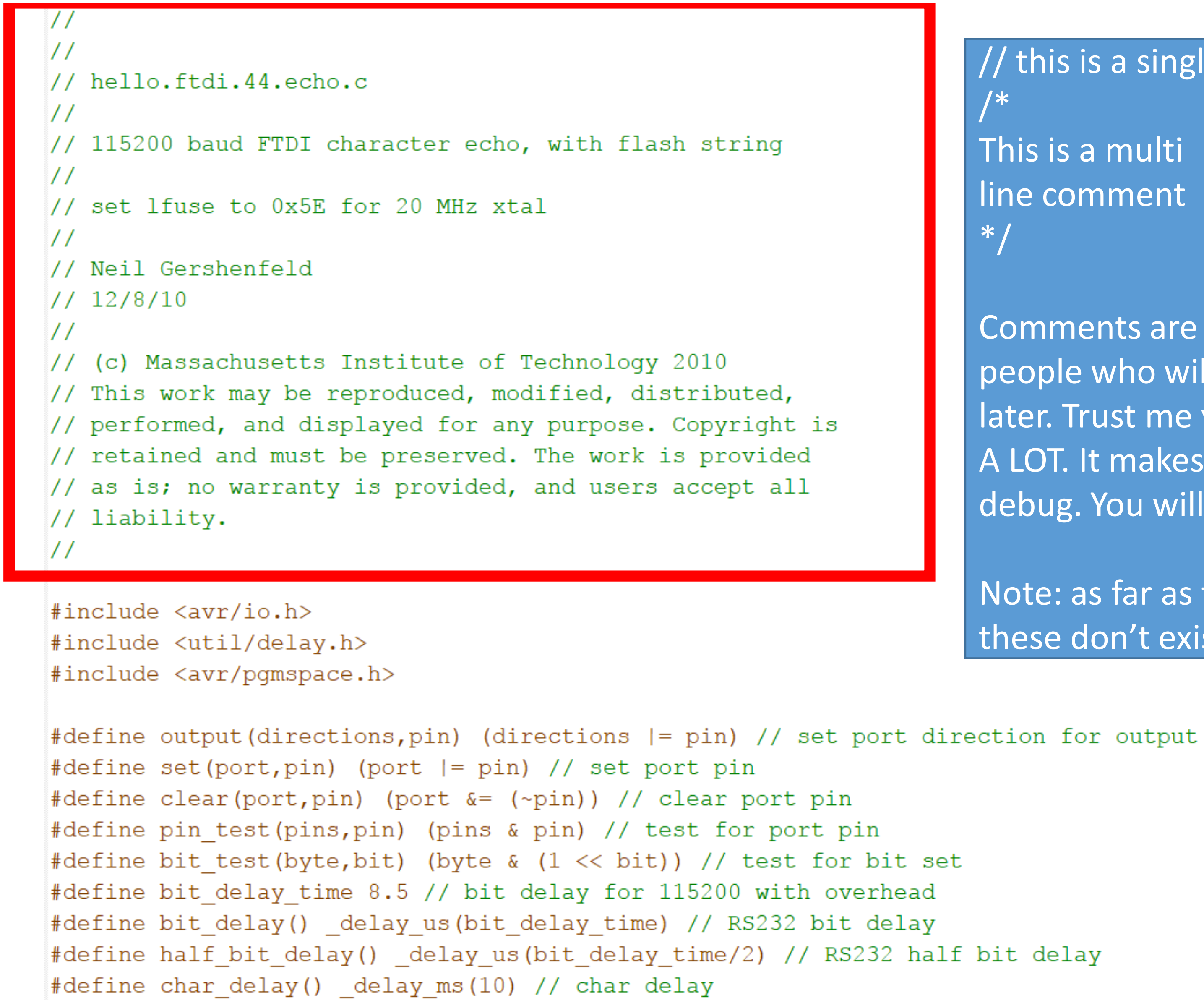
And that is programming in C in a nutshell

NOT BAD



```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
// Neil Gershenfeld
// 12/8/10
//
// (c) Massachusetts Institute of Tech
// This work may be reproduced, modifi
// performed, and displayed for any pu
// retained and must be preserved. The
// as is; no warranty is provided, and
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direction for output
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

Lets walk through Neil’s hello.ftdi.44.echo.c to explore AVR C code

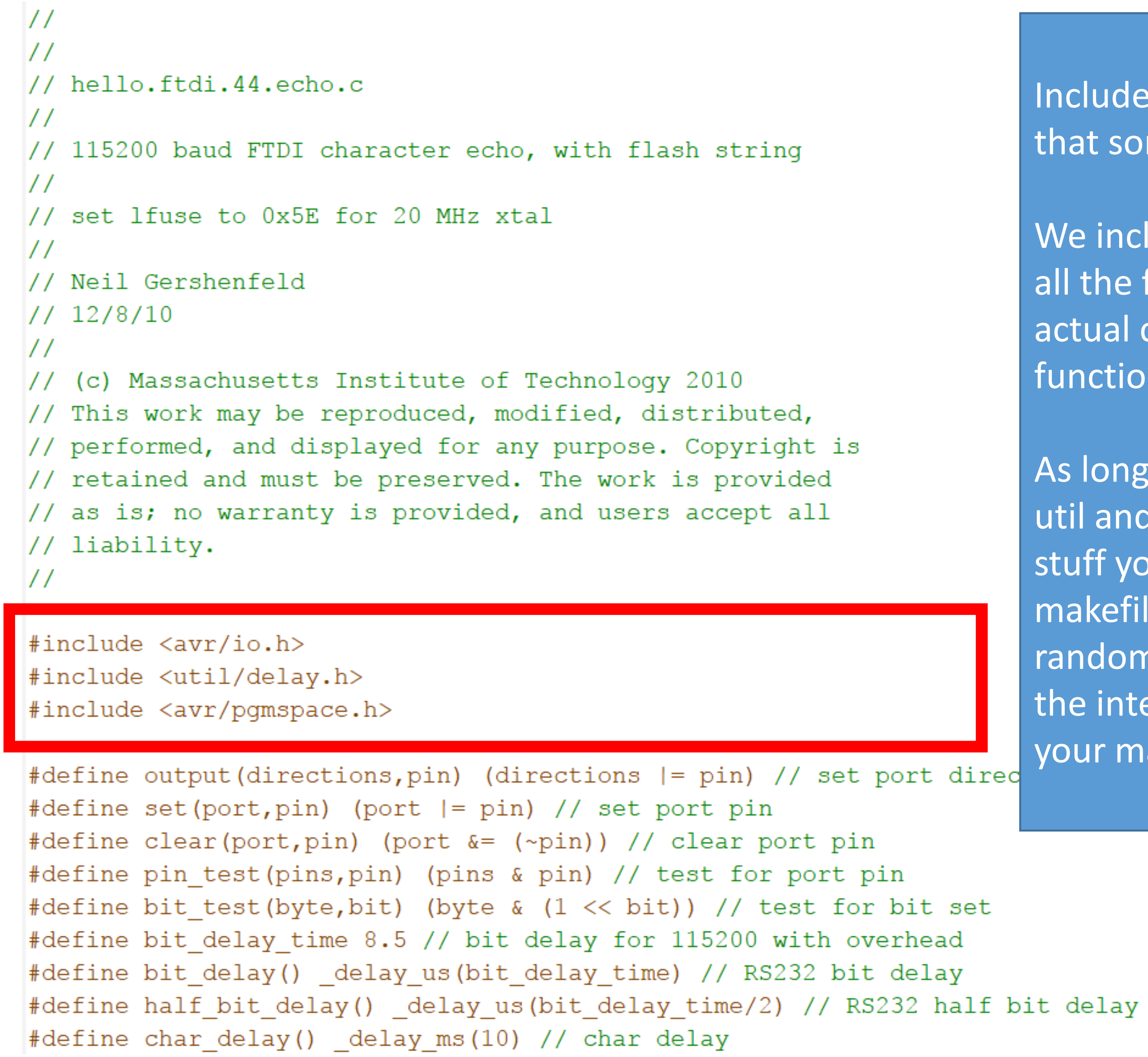


```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
// Neil Gershenfeld
// 12/8/10
//
// (c) Massachusetts Institute of Technology 2010
// This work may be reproduced, modified, distributed,
// performed, and displayed for any purpose. Copyright is
// retained and must be preserved. The work is provided
// as is; no warranty is provided, and users accept all
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direction for output
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

// this is a single line comment
/*
This is a multi
line comment
*/

Comments are for YOU and for other people who will read your code later. Trust me you want to comment A LOT. It makes it much easier to debug. You will be happy later!

Note: as far as the program knows these don't exist.

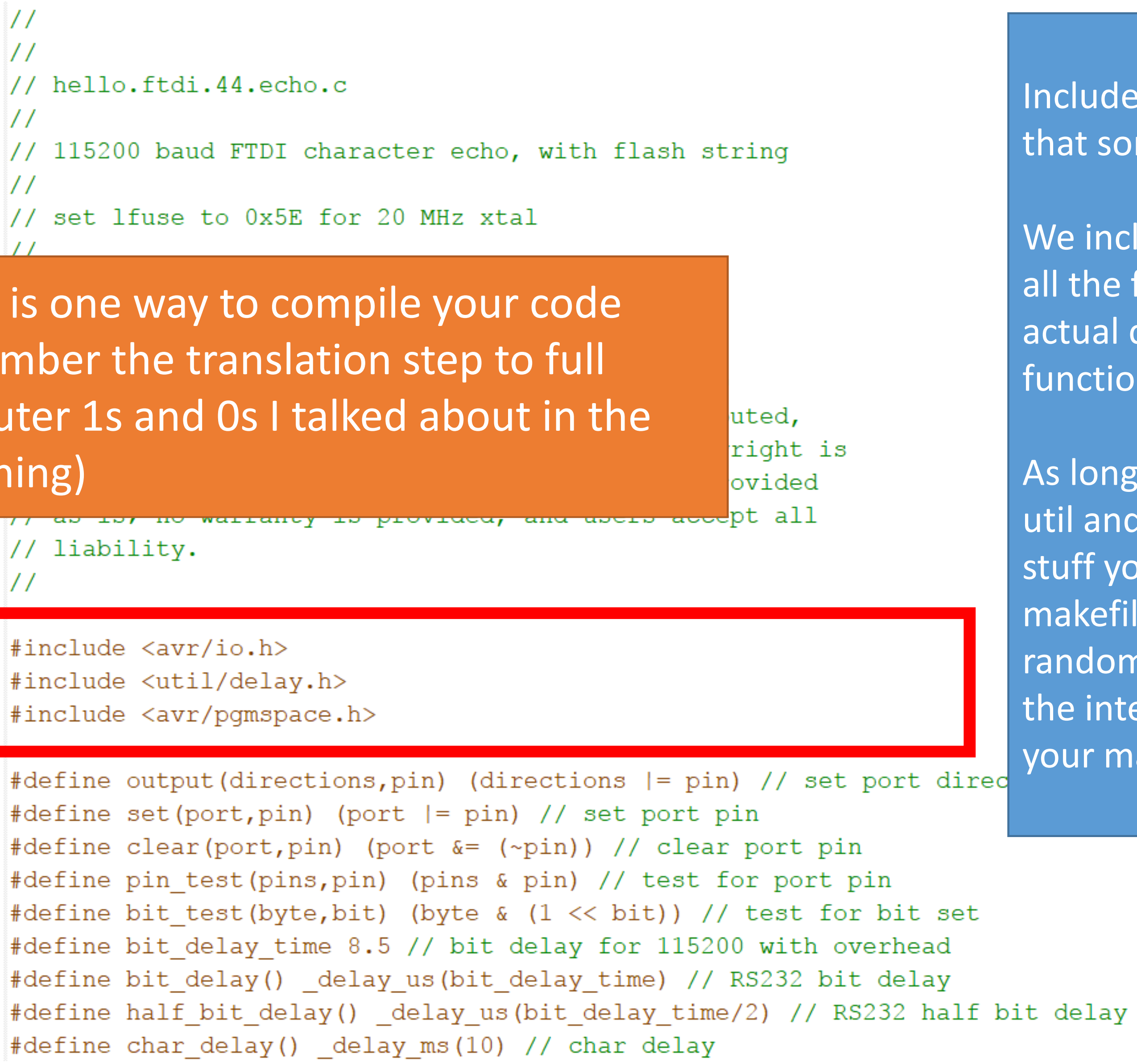


```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
// Neil Gershenfeld
// 12/8/10
//
// (c) Massachusetts Institute of Technology 2010
// This work may be reproduced, modified, distributed,
// performed, and displayed for any purpose. Copyright is
// retained and must be preserved. The work is provided
// as is; no warranty is provided, and users accept all
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direc
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

Includes are how you reuse code that someone else wrote.

We include .h files as they describe all the functions we need. Note: the actual code implementing those functions resides in a .c file.

As long as you are using only avr and util and other basic c programming stuff you won’t need to change your makefile. If you end up using random stuff from somewhere on the internet you will need to update your makefile to include that code.



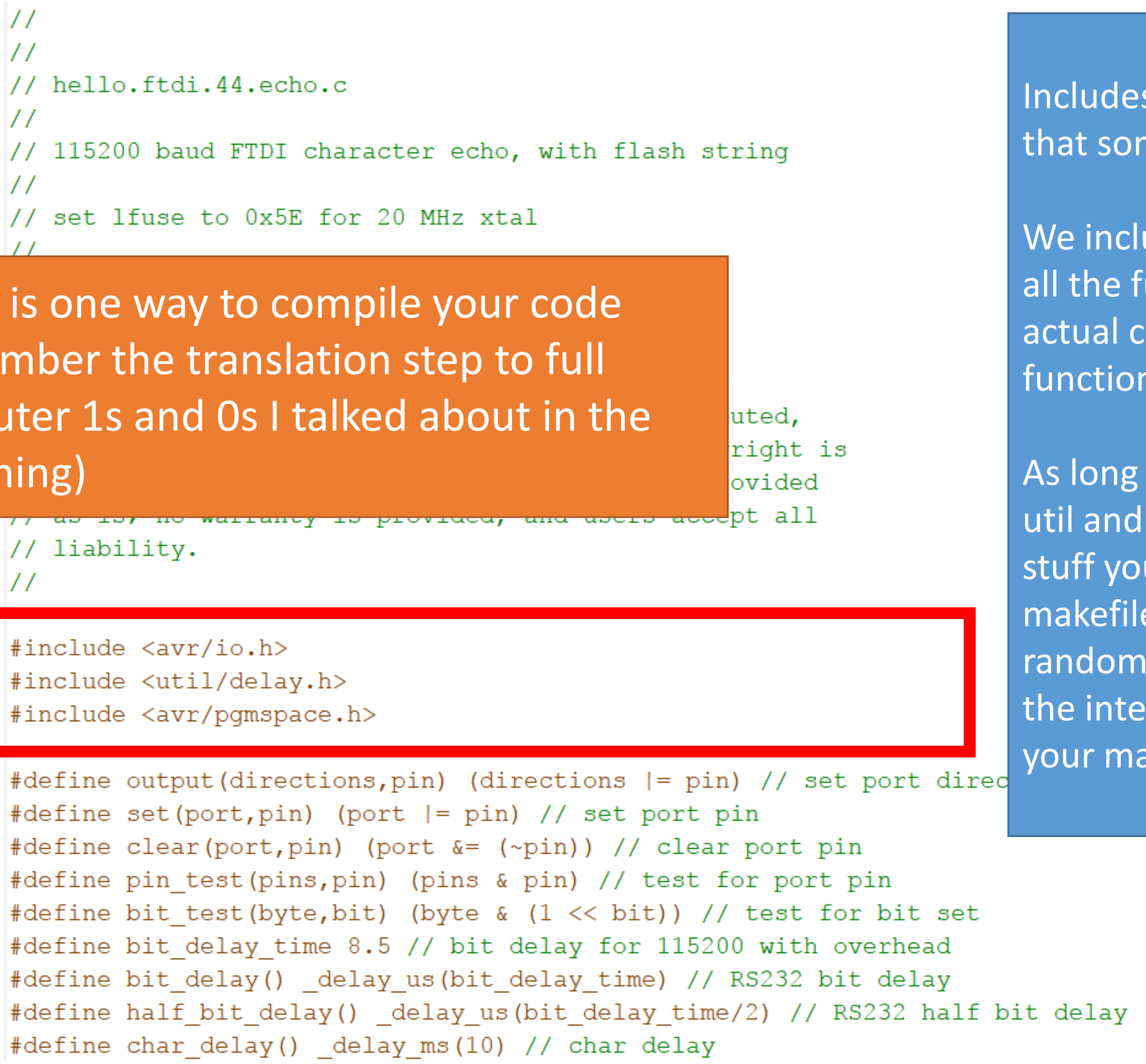
```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
                                                      uted,
                                                      right is
                                                      ovided
// as is; no warranty is provided, and users accept all
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direc
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

MAKE is one way to compile your code (remember the translation step to full computer 1s and 0s I talked about in the beginning)

Includes are how you reuse code that someone else wrote.

We include .h files as they describe all the functions we need. Note: the actual code implementing those functions resides in a .c file.

As long as you are using only avr and util and other basic c programming stuff you won’t need to change your makefile. If you end up using random stuff from somewhere on the internet you will need to update your makefile to include that code.



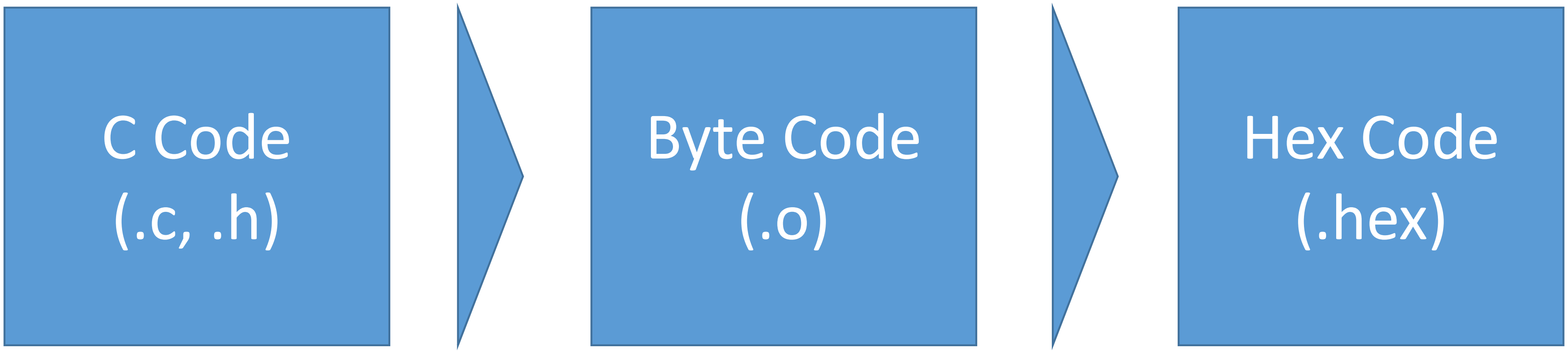
```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
                                                uted,
                                                right is
                                                ovided
// as is; no warranty is provided, and users accept all
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direc
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

MAKE is one way to compile your code (remember the translation step to full computer 1s and 0s I talked about in the beginning)

Includes are how you reuse code that someone else wrote.

We include .h files as they describe all the functions we need. Note: the actual code implementing those functions resides in a .c file.

As long as you are using only avr and util and other basic c programming stuff you won't need to change your makefile. If you end up using random stuff from somewhere on the internet you will need to update your makefile to include that code.



C Code
(.c, .h)

Byte Code
(.o)

Hex Code
(.hex)

Compiler does this for you automagically (by MAKE)! So all you have to do is write code that obeys the rules of C (and AVR)!

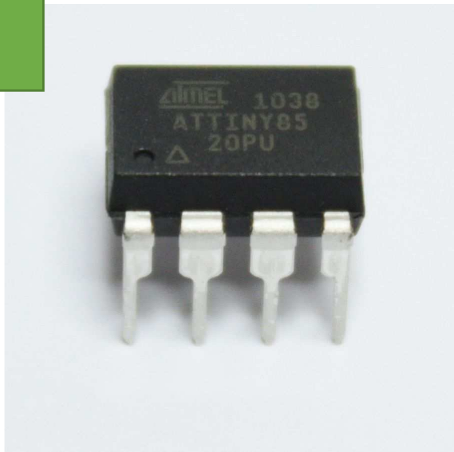
C Code
(.c, .h)

Byte Code
(.o)

Hex Code
(.hex)

Lets pause and take a look at the MAKEFILE
(aka the instructions to MAKE)

automagically (by MAKE)!
So all you have to do is
write code that obeys the
rules of C (and AVR)!



```
PROJECT=hello.ftdi.44.echo
SOURCES=$(PROJECT).c
MMCU=attiny44
F_CPU = 20000000

CFLAGS=-mmcu=$(MMCU) -Wall -Os -DF_CPU=$(F_CPU)

$(PROJECT).hex: $(PROJECT).out
   avr-objcopy -O ihex $(PROJECT).out $(PROJECT).c.hex;\
   avr-size --mcu=$(MMCU) --format=avr $(PROJECT).out

$(PROJECT).out: $(SOURCES)
   avr-gcc $(CFLAGS) -I./ -o $(PROJECT).out $(SOURCES)

program-usbtiny: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U flash:w:$(PROJECT).c.hex

program-usbtiny-fuses: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U lfuse:w:0x5E:m
```

The file to make

```
PROJECT=hello.ftdi.44.echo
SOURCES=$(PROJECT).c
MMCU=attiny44
F_CPU = 20000000

CFLAGS=-mmcu=$(MMCU) -Wall -Os -DF_CPU=$(F_CPU)

$(PROJECT).hex: $(PROJECT).out
   avr-objcopy -O ihex $(PROJECT).out $(PROJECT).c.hex;\
   avr-size --mcu=$(MMCU) --format=avr $(PROJECT).out

$(PROJECT).out: $(SOURCES)
   avr-gcc $(CFLAGS) -I./ -o $(PROJECT).out $(SOURCES)

program-usbtiny: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U flash:w:$(PROJECT).c.hex

program-usbtiny-fuses: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U lfuse:w:0x5E:m
```

What board you are making it for

```
PROJECT=hello.ftdi.44.echo
SOURCES=$(PROJECT).c
MMCU=attiny44
F_CPU = 20000000

CFLAGS=-mmcu=$(MMCU) -Wall -Os -DF_CPU=$(F_CPU)

$(PROJECT).hex: $(PROJECT).out
   avr-objcopy -O ihex $(PROJECT).out $(PROJECT).c.hex;\
   avr-size --mcu=$(MMCU) --format=avr $(PROJECT).out

$(PROJECT).out: $(SOURCES)
   avr-gcc $(CFLAGS) -I./ -o $(PROJECT).out $(SOURCES)

program-usbtiny: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U flash:w:$(PROJECT).c.hex

program-usbtiny-fuses: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U lfuse:w:0x5E:m
```

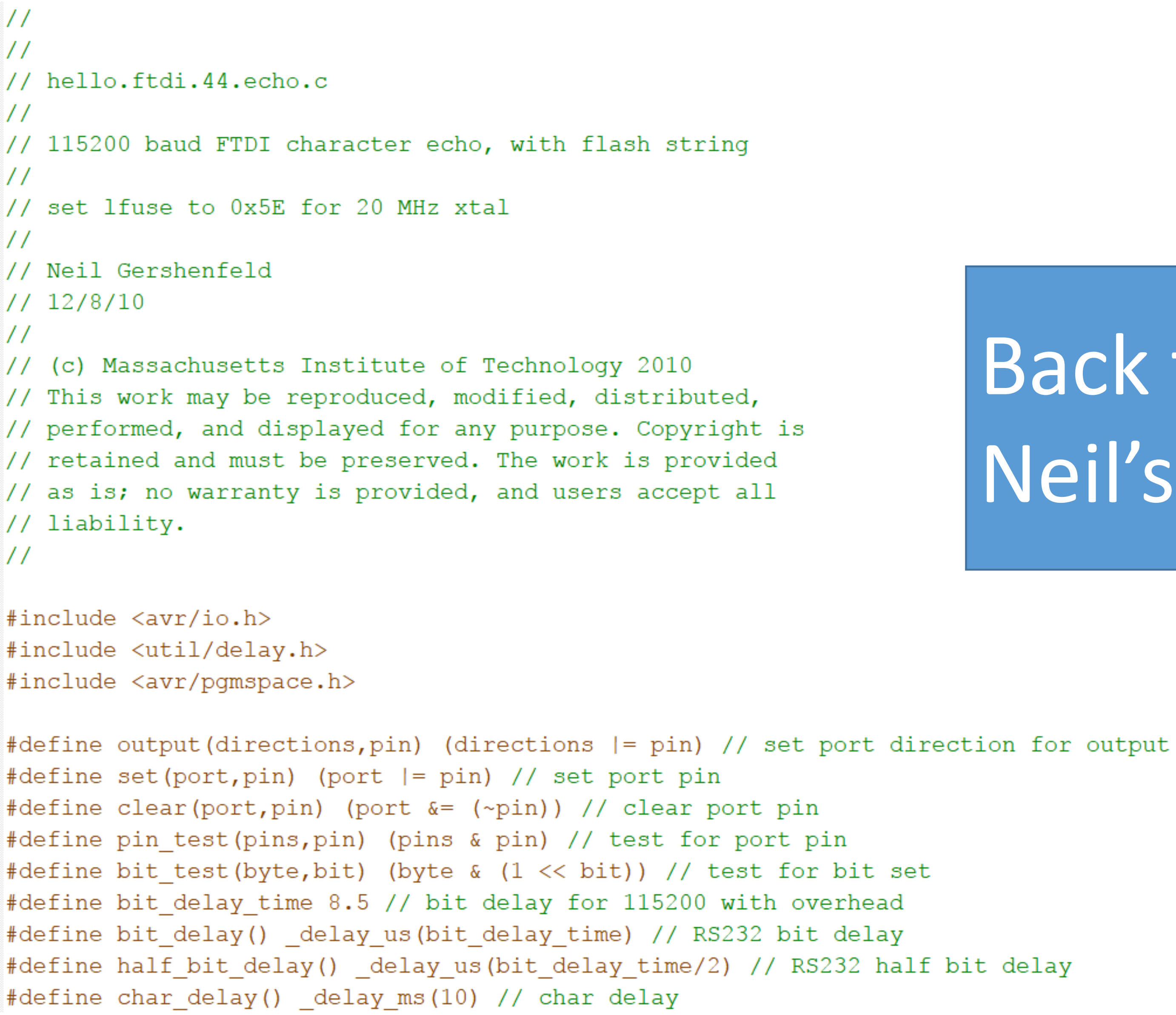
Compiler flags (don’t worry about it)

```
PROJECT=hello.ftdi.44.echo
SOURCES=$(PROJECT).c
MMCU=attiny44
F_CPU = 20000000

CFLAGS=-mmcu=$(MMCU) -Wall -Os -DF_CPU=$(F_CPU)

$(PROJECT).hex: $(PROJECT).out
   avr-objcopy -O ihex $(PROJECT).out $(PROJECT).c.hex;\
   avr-size --mcu=$(MMCU) --format=avr $(PROJECT).out

$(PROJECT).out: $(SOURCES)
   avr-gcc $(CFLAGS) -I./ -o $(PROJECT).out $(SOURCES)

program-usbtiny: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U flash:w:$(PROJECT).c.hex

program-usbtiny-fuses: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U lfuse:w:0x5E:m
```

Tells the compiler to make a .o and a .hex file using avr (and automatically links in the standard c library things)

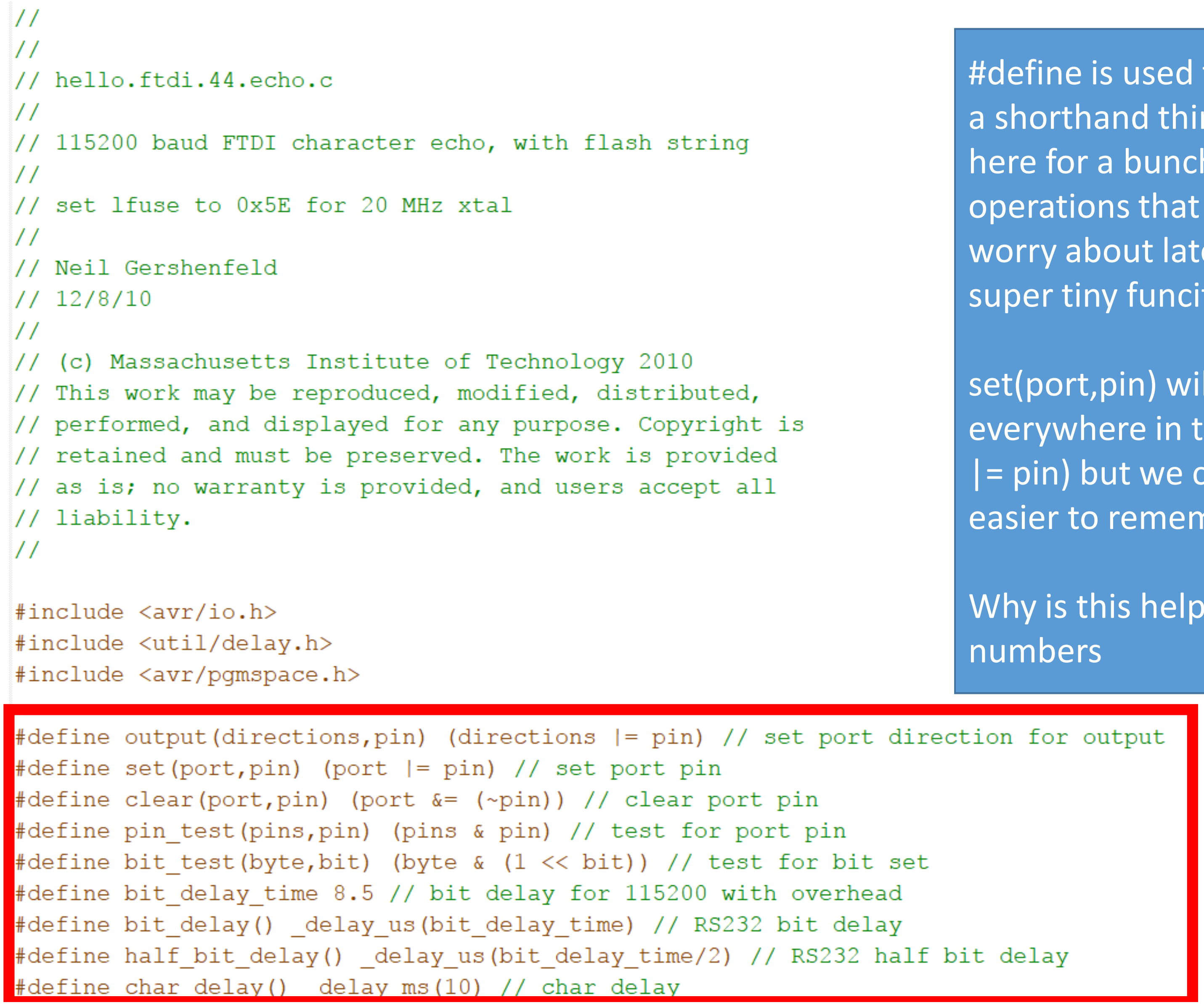
```
PROJECT=hello.ftdi.44.echo
SOURCES=$(PROJECT).c
MMCU=attiny44
F_CPU = 20000000

CFLAGS=-mmcu=$(MMCU) -Wall -Os -DF_CPU=$(F_CPU)

$(PROJECT).hex: $(PROJECT).out
   avr-objcopy -O ihex $(PROJECT).out $(PROJECT).c.hex;\
   avr-size --mcu=$(MMCU) --format=avr $(PROJECT).out

$(PROJECT).out: $(SOURCES)
   avr-gcc $(CFLAGS) -I./ -o $(PROJECT).out $(SOURCES)

program-usbtiny: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U flash:w:$(PROJECT).c.hex

program-usbtiny-fuses: $(PROJECT).hex
   avrdude -p t44 -P usb -c usbtiny -U lfuse:w:0x5E:m
```

Takes a .hex file and sends it to the avr using with a program or fuse command



```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
// Neil Gershenfeld
// 12/8/10
//
// (c) Massachusetts Institute of Technology 2010
// This work may be reproduced, modified, distributed,
// performed, and displayed for any purpose. Copyright is
// retained and must be preserved. The work is provided
// as is; no warranty is provided, and users accept all
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direction for output
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

Back to Neil’s code!



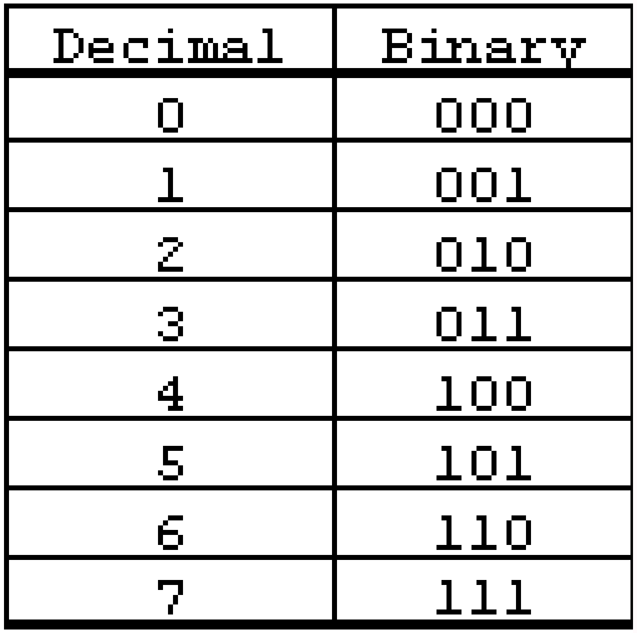
```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
// Neil Gershenfeld
// 12/8/10
//
// (c) Massachusetts Institute of Technology 2010
// This work may be reproduced, modified, distributed,
// performed, and displayed for any purpose. Copyright is
// retained and must be preserved. The work is provided
// as is; no warranty is provided, and users accept all
// liability.
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#define output(directions,pin) (directions |= pin) // set port direction for output
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit set
#define bit_delay_time 8.5 // bit delay for 115200 with overhead
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 half bit delay
#define char_delay() _delay_ms(10) // char delay
```

#define is used to make some word a shorthand thing. Neil uses them here for a bunch of quick bitwise operations that we won’t have to worry about later. Think of them as super tiny funcitons.

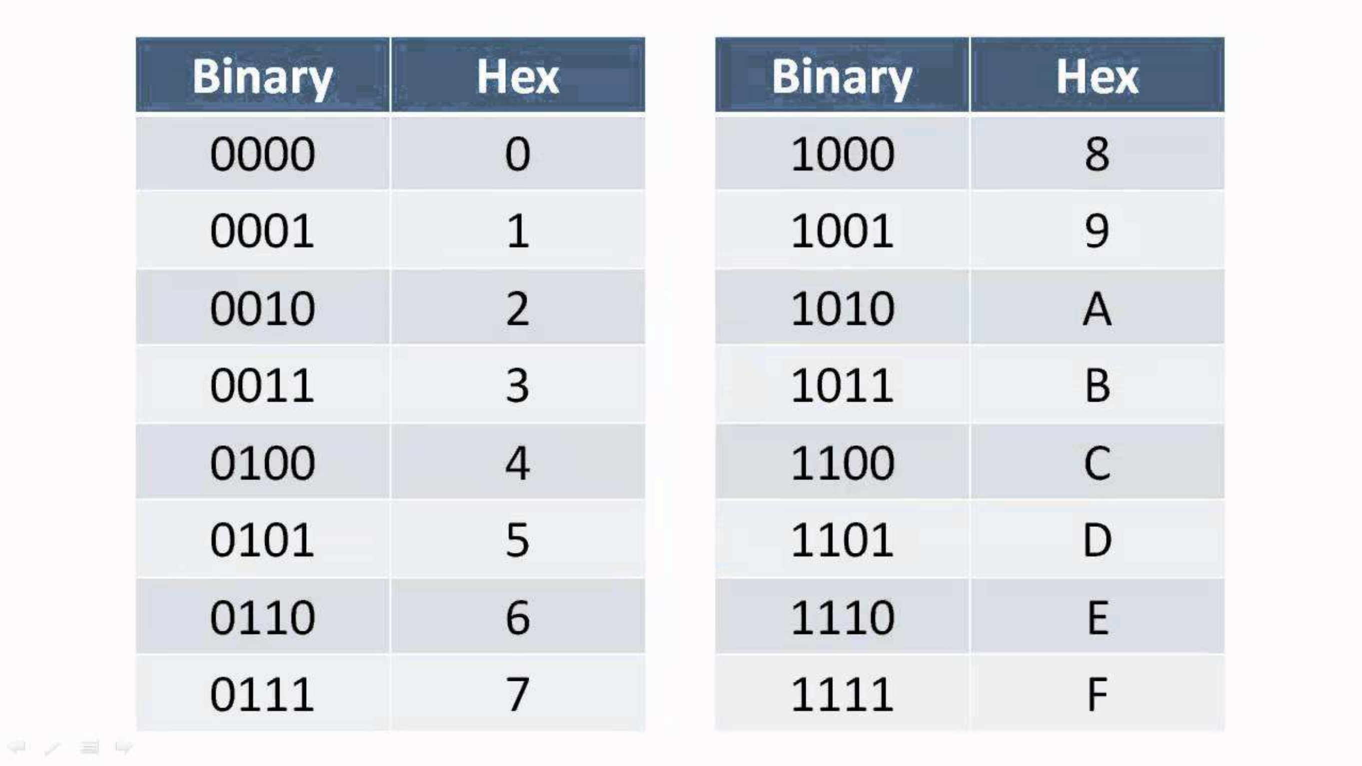
set(port,pin) will be replaced everywhere in the code with (port |= pin) but we can simply write the easier to remember set(port,pin)

Why is this helpful – lets talk binary numbers

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

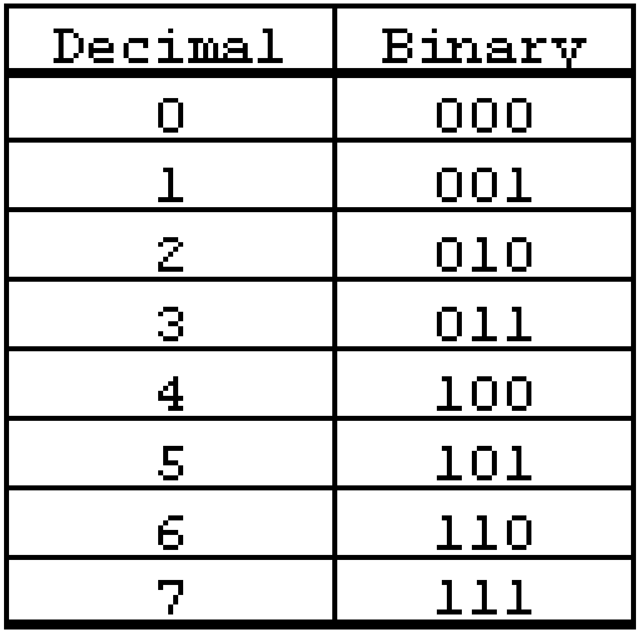


Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

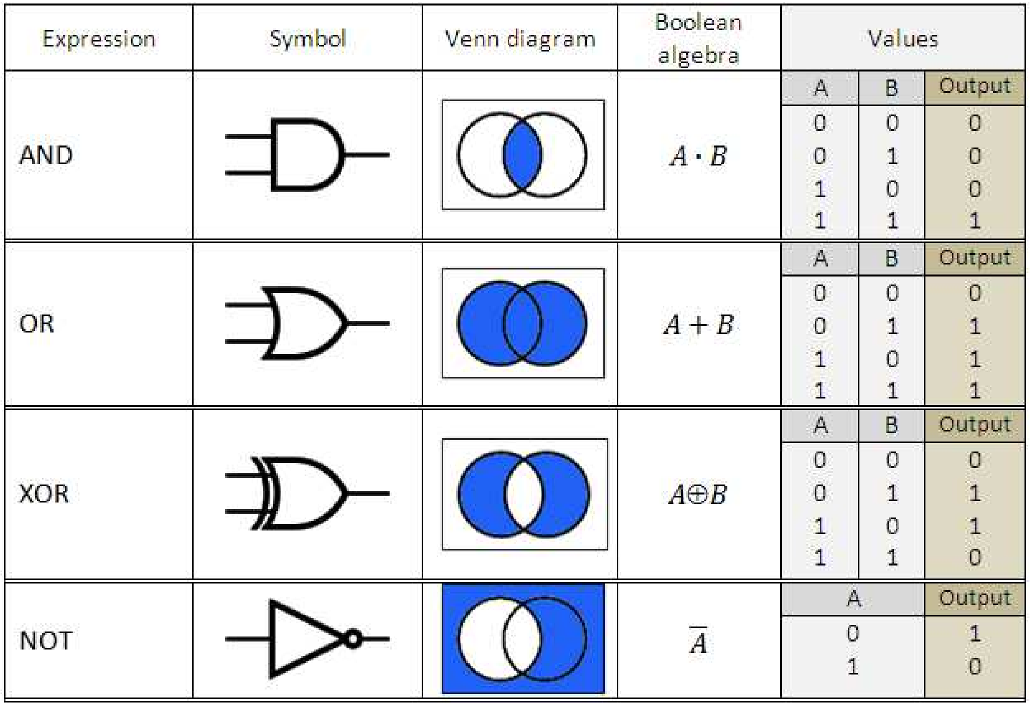


Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

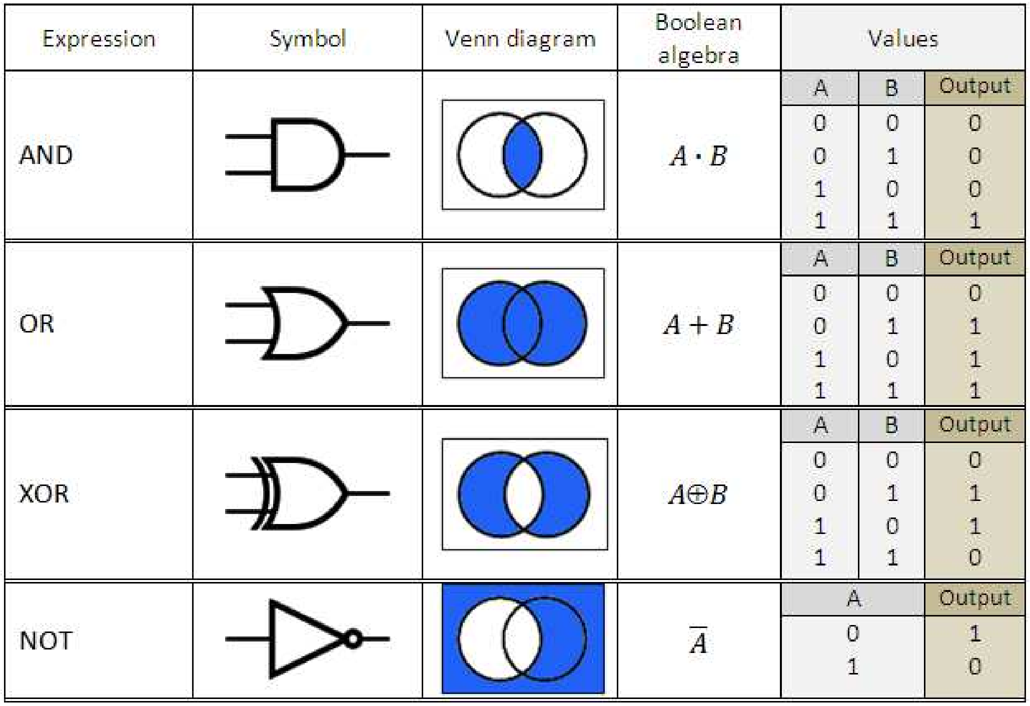
Binary	Hex
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F



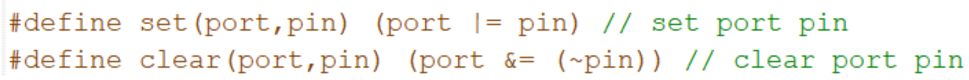
Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111



Expression	Symbol	Venn diagram	Boolean algebra	Values		
AND			$A \cdot B$	A	B	Output
				0	0	0
				0	1	0
				1	0	0
				1	1	1
OR			$A + B$	A	B	Output
				0	0	0
				0	1	1
				1	0	1
				1	1	1
XOR			$A \oplus B$	A	B	Output
				0	0	0
				0	1	1
				1	0	1
				1	1	0
NOT			$\overline{A}$	A		Output
				0		1
				1		0



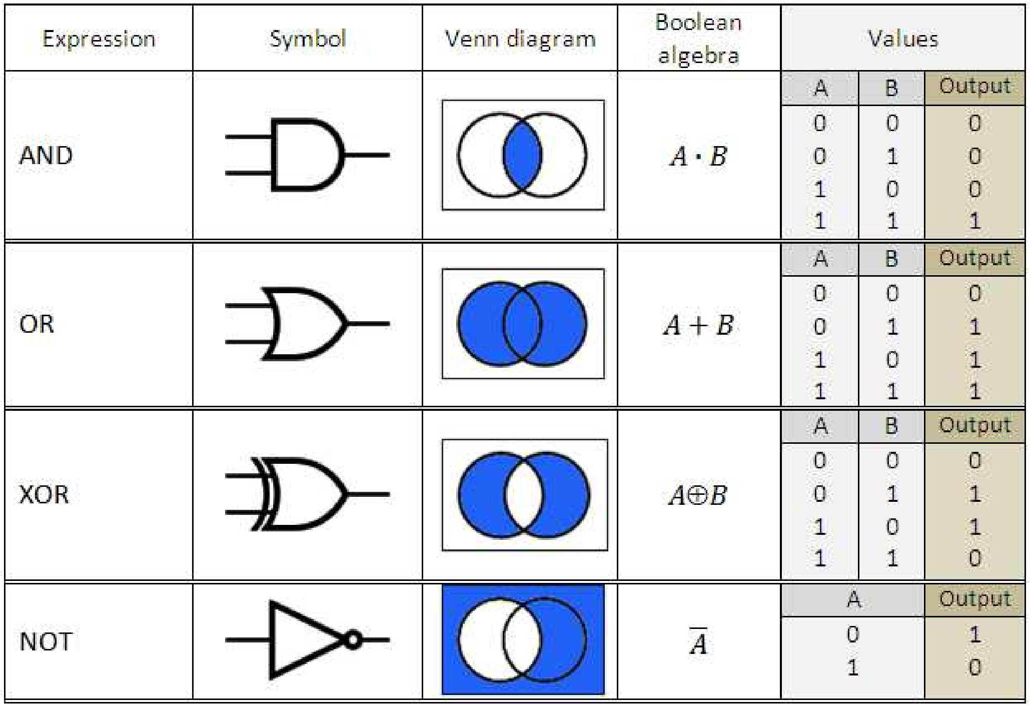
Expression	Symbol	Venn diagram	Boolean algebra	Values		
				A	B	Output
AND			$A \cdot B$	0	0	0
				0	1	0
				1	0	0
				1	1	1
				A	B	Output
OR			$A + B$	0	0	0
				0	1	1
				1	0	1
				1	1	1
				A	B	Output
XOR			$A \oplus B$	0	0	0
				0	1	1
				1	0	1
				1	1	0
				A		Output
NOT			$\overline{A}$	0		1
				1		0



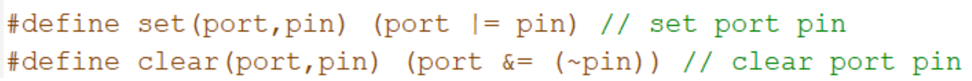
```
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
```

| is logical OR
& is logical AND
~ is logical NOT

So if we pick a pin with a 1 then OR it we will set it. And if we AND the NOT of it we will AND a 0 and thus unset it!



Expression	Symbol	Venn diagram	Boolean algebra	Values		
				A	B	Output
AND			$A \cdot B$	0 0 1 1	0 1 0 1	0 0 0 1
				A	B	Output
OR			$A + B$	0 0 1 1	0 1 0 1	0 1 1 1
				A	B	Output
XOR			$A \oplus B$	0 0 1 1	0 1 0 1	0 1 1 0
				A		Output
NOT			$\overline{A}$	0 1		1 0

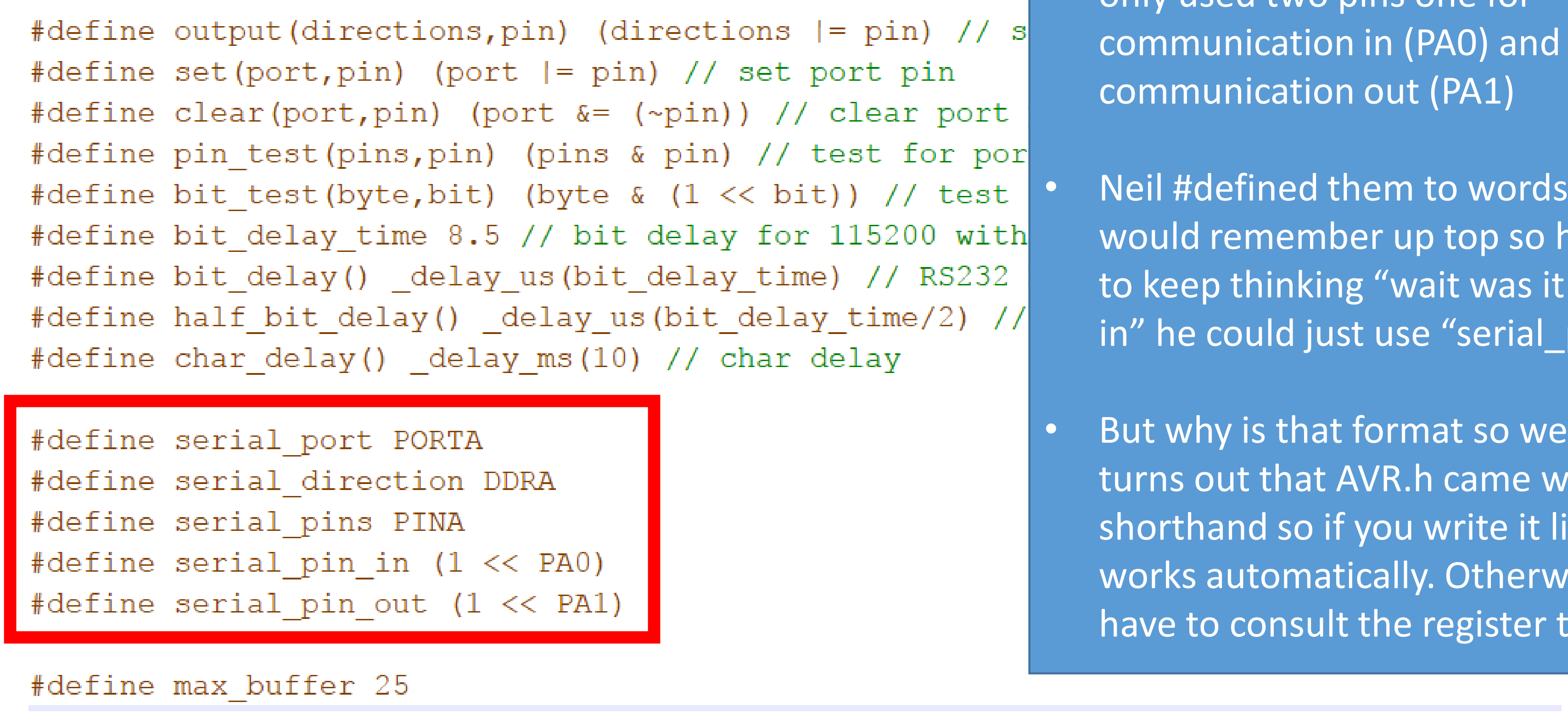


```
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
```

| is logical OR
& is logical AND
~ is logical NOT

So if we pick a pin with a 1 then OR it we will set it. And if we AND the NOT of it we will AND a 0 and thus unset it!

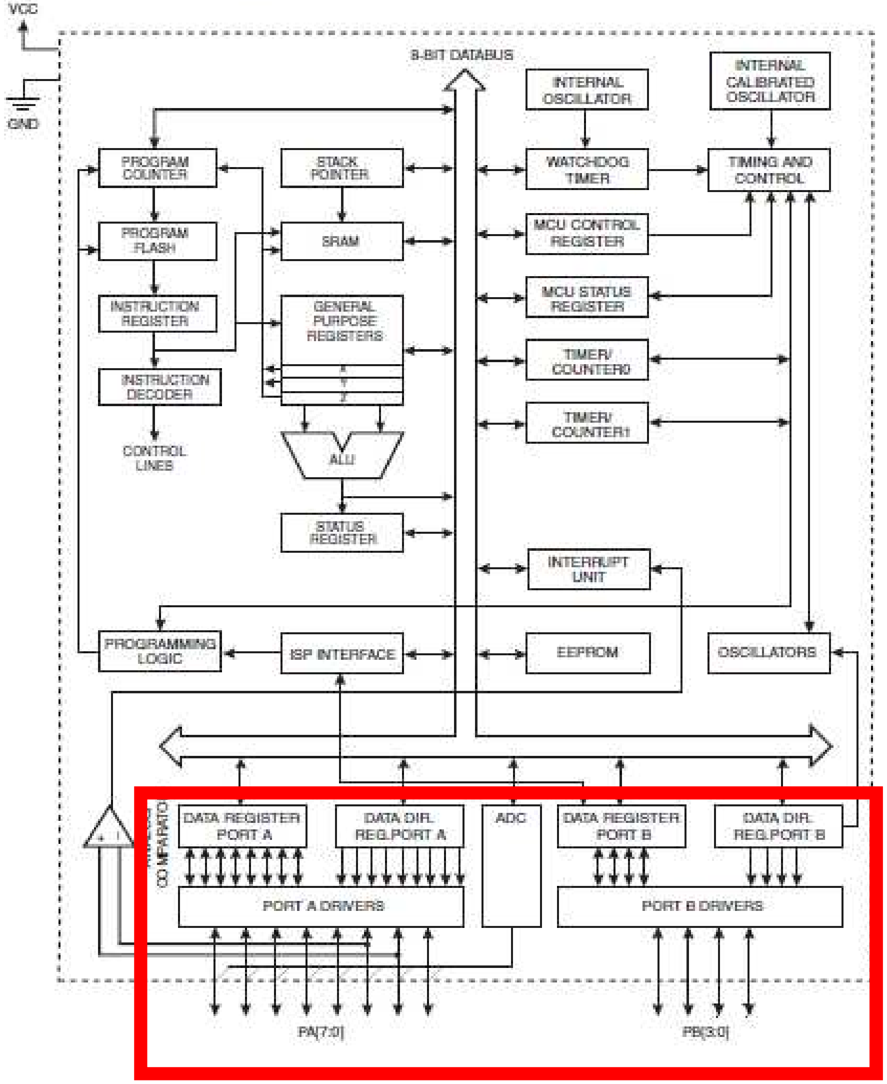
But again Neil gives us this stuff so just remember to use it and you won’t have to worry about it as much! :-)

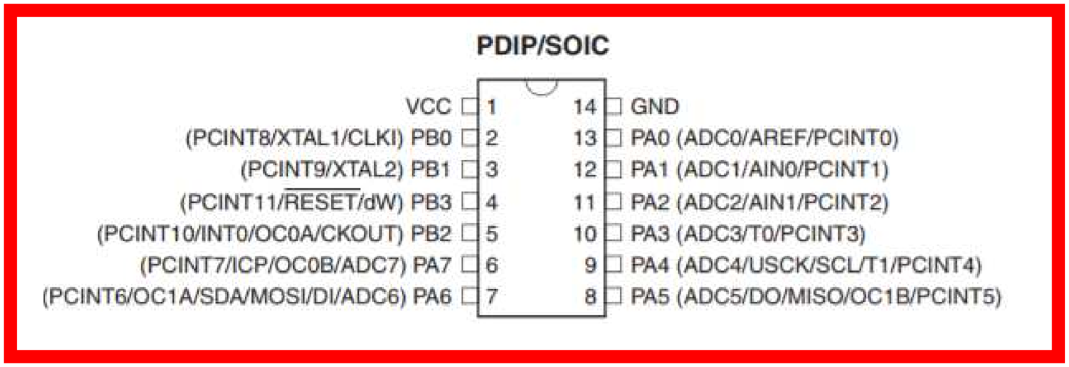


```
#define output(directions,pin) (directions |= pin) // s
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port
#define pin_test(pins,pin) (pins & pin) // test for por
#define bit_test(byte,bit) (byte & (1 << bit)) // test
#define bit_delay_time 8.5 // bit delay for 115200 with
#define bit_delay() _delay_us(bit_delay_time) // RS232
#define half_bit_delay() _delay_us(bit_delay_time/2) //
#define char_delay() _delay_ms(10) // char delay

#define serial_port PORTA
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)

#define max_buffer 25
```

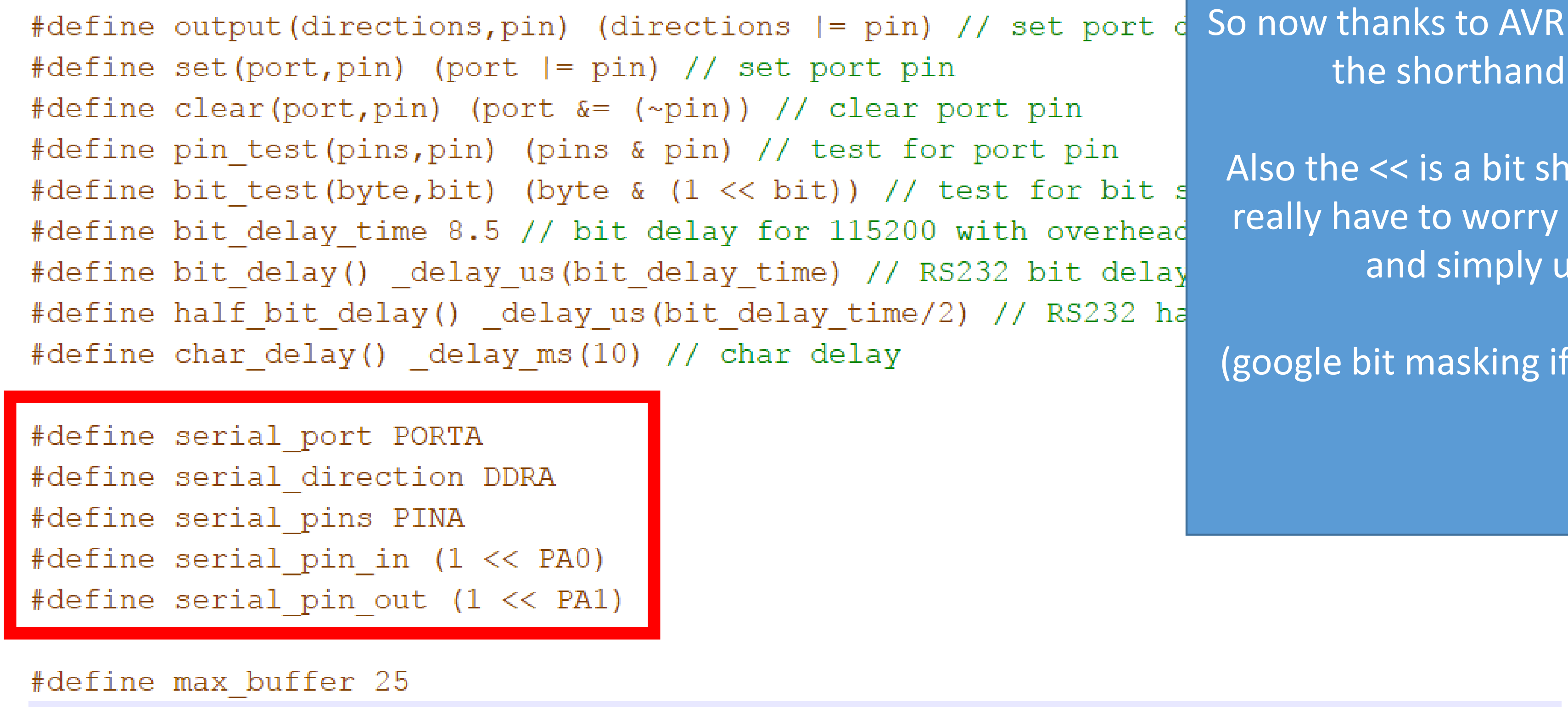
- Oh right this code was talking over serial with the computer and that was it so it only used two pins one for communication in (PA0) and one for communication out (PA1)
- Neil #defined them to words that he would remember up top so he didn’t have to keep thinking “wait was it PA0 or 1 for in” he could just use “serial_pin_in”
- But why is that format so weird? Well it turns out that AVR.h came with a bunch of shorthand so if you write it like that it works automatically. Otherwise you would have to consult the register table!

Figure 2-1. Block Diagram




PDIP/SOIC
VCC 1
(PCINT8/XTAL1/CLKI) PB0 2
(PCINT9/XTAL2) PB1 3
(PCINT11/RESET/dW) PB3 4
(PCINT10/INT0/OC0A/CKOUT) PB2 5
(PCINT7/ICP/OC0B/ADC7) PA7 6
(PCINT6/OC1A/SDA/MOSI/DI/ADC6) PA6 7
14 GND
13 PA0 (ADC0/AREF/PCINT0)
12 PA1 (ADC1/AIN0/PCINT1)
11 PA2 (ADC2/AIN1/PCINT2)
10 PA3 (ADC3/T0/PCINT3)
9 PA4 (ADC4/USCK/SCL/T1/PCINT4)
8 PA5 (ADC5/DO/MISO/OC1B/PCINT5)

Remember from last time (electronics design) that the data sheet describes all of the ports and their names and what pins they are etc.

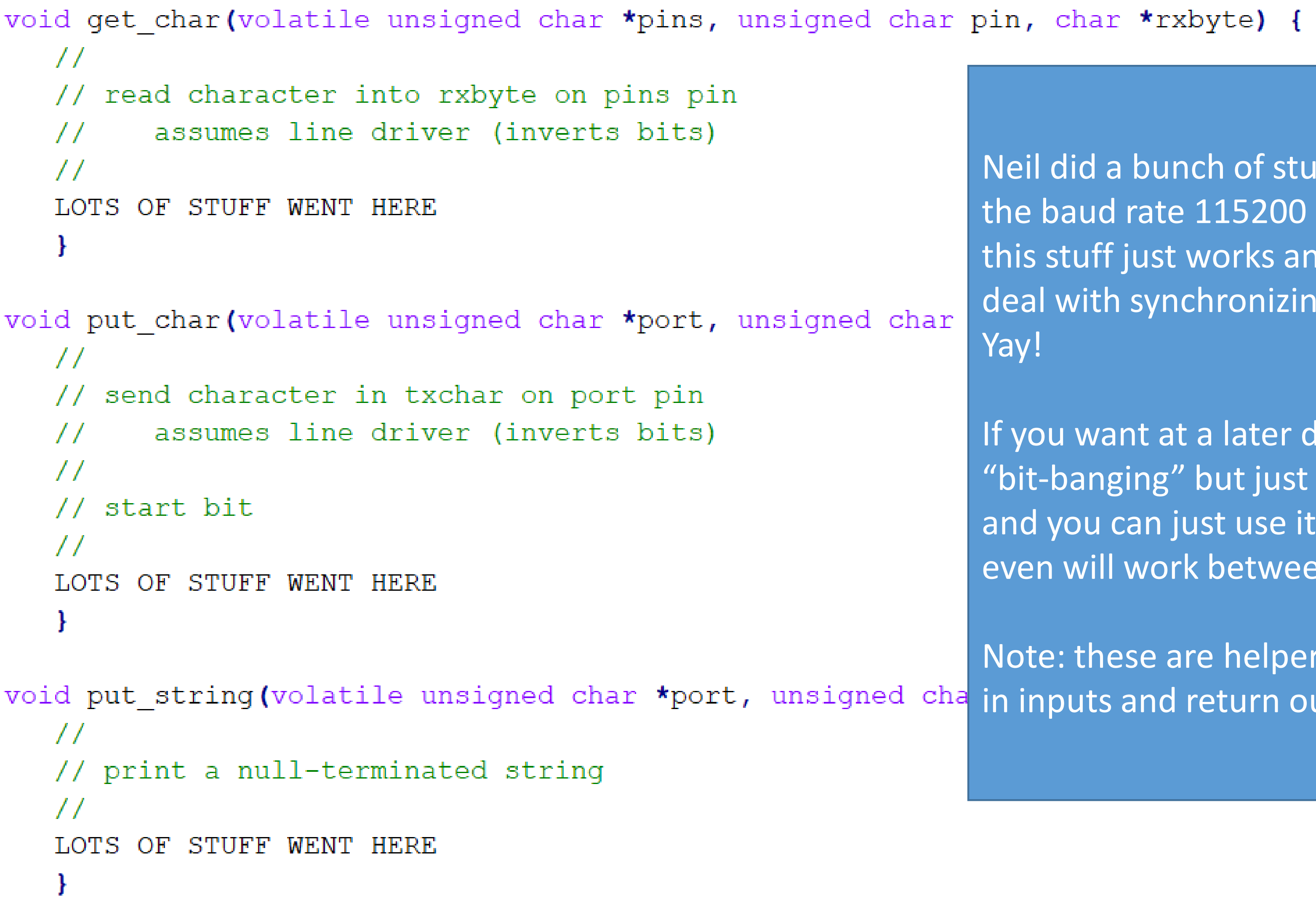


```
#define output(directions,pin) (directions |= pin) // set port d
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << bit)) // test for bit s
#define bit_delay_time 8.5 // bit delay for 115200 with overhea
#define bit_delay() _delay_us(bit_delay_time) // RS232 bit delay
#define half_bit_delay() _delay_us(bit_delay_time/2) // RS232 ha
#define char_delay() _delay_ms(10) // char delay

#define serial_port PORTA
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)

#define max_buffer 25
```

So now thanks to AVR.h we can just use the shorthand mapping!

Also the << is a bit shift but you don’t really have to worry about it for now and simply use it! :-)

(google bit masking if you are curious)

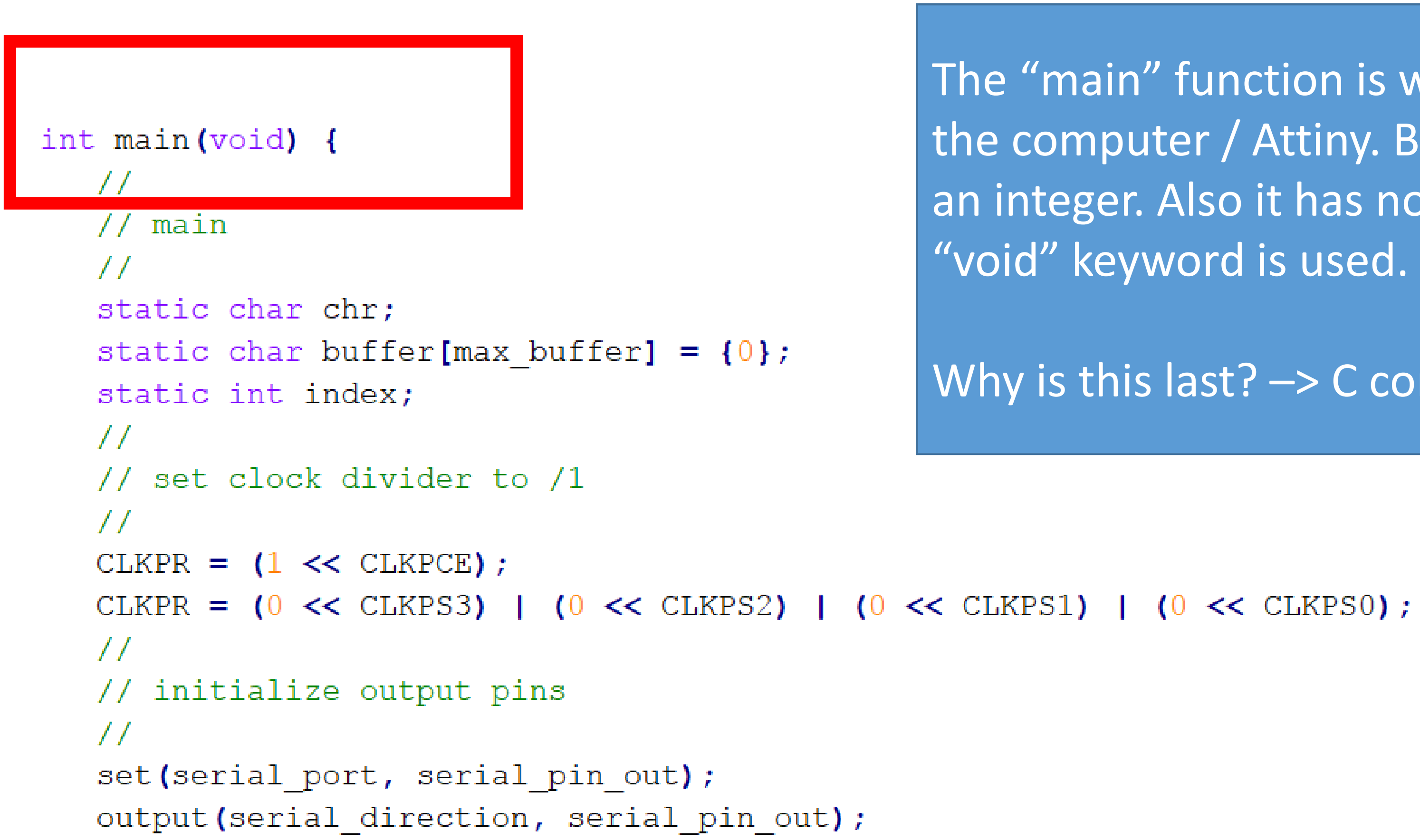


```
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte) {
   //
   // read character into rxbyte on pins pin
   //    assumes line driver (inverts bits)
   //
   LOTS OF STUFF WENT HERE
   }

void put_char(volatile unsigned char *port, unsigned char
   //
   // send character in txchar on port pin
   //    assumes line driver (inverts bits)
   //
   // start bit
   //
   LOTS OF STUFF WENT HERE
   }

void put_string(volatile unsigned char *port, unsigned cha
   //
   // print a null-terminated string
   //
   LOTS OF STUFF WENT HERE
   }
```

Neil did a bunch of stuff for you so if you use the baud rate 115200 (like from last week) this stuff just works and you don’t have to deal with synchronizing with the computer! Yay!

If you want at a later date we can talk about “bit-banging” but just know that this works and you can just use it to send characters. It even will work between two different Attinys.

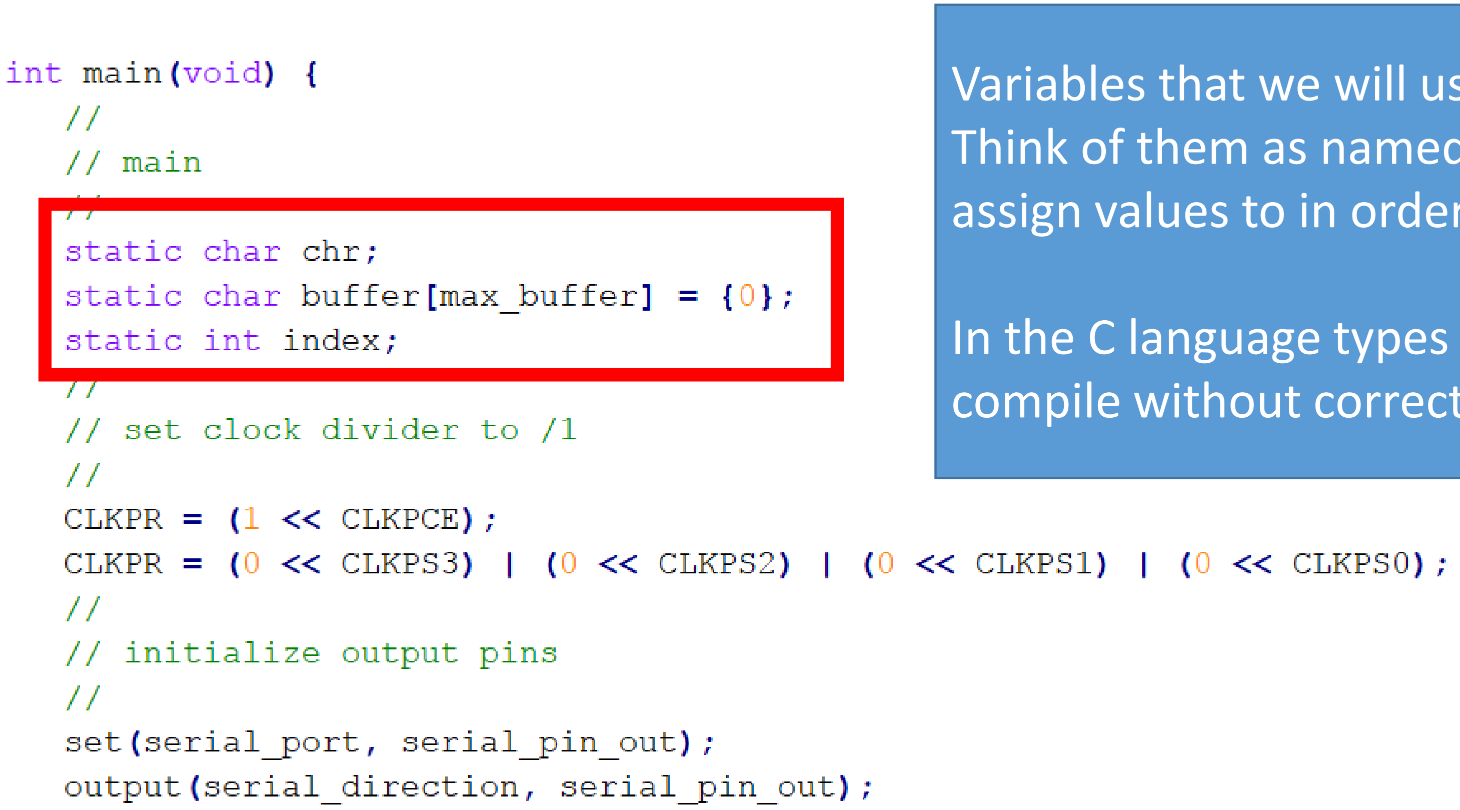
Note: these are helper functions as they take in inputs and return outputs



```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;
   //
   // set clock divider to /1
   //
   CLKPR = (1 << CLKPCE);
   CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
   //
   // initialize output pins
   //
   set(serial_port, serial_pin_out);
   output(serial_direction, serial_pin_out);
```

The “main” function is what is actually run by the computer / Attiny. By standard it returns an integer. Also it has no inputs thus the “void” keyword is used.

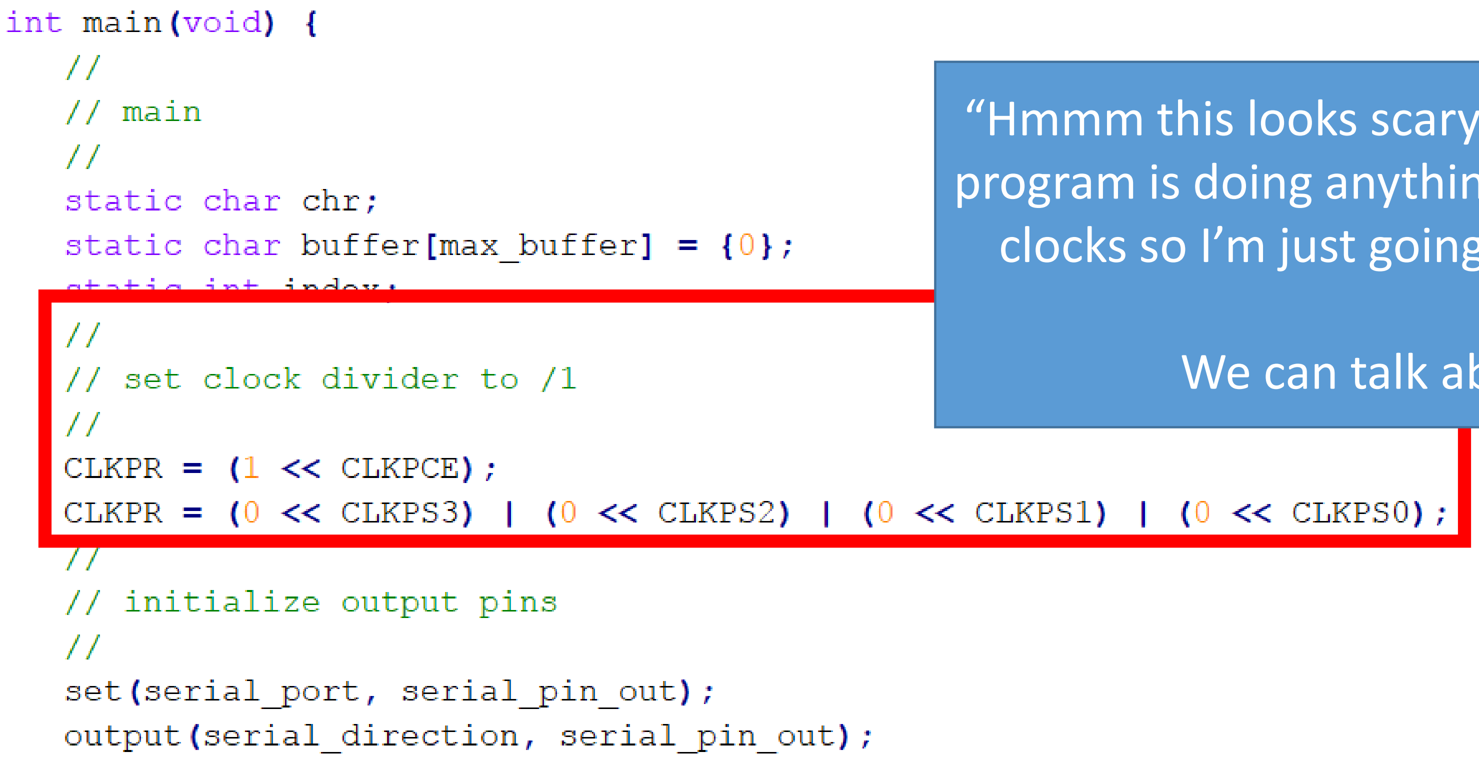
Why is this last? –> C compiles top down



```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;
   //
   // set clock divider to /1
   //
   CLKPR = (1 << CLKPCE);
   CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
   //
   // initialize output pins
   //
   set(serial_port, serial_pin_out);
   output(serial_direction, serial_pin_out);
```

Variables that we will use in our function. Think of them as named things which we can assign values to in order to do things.

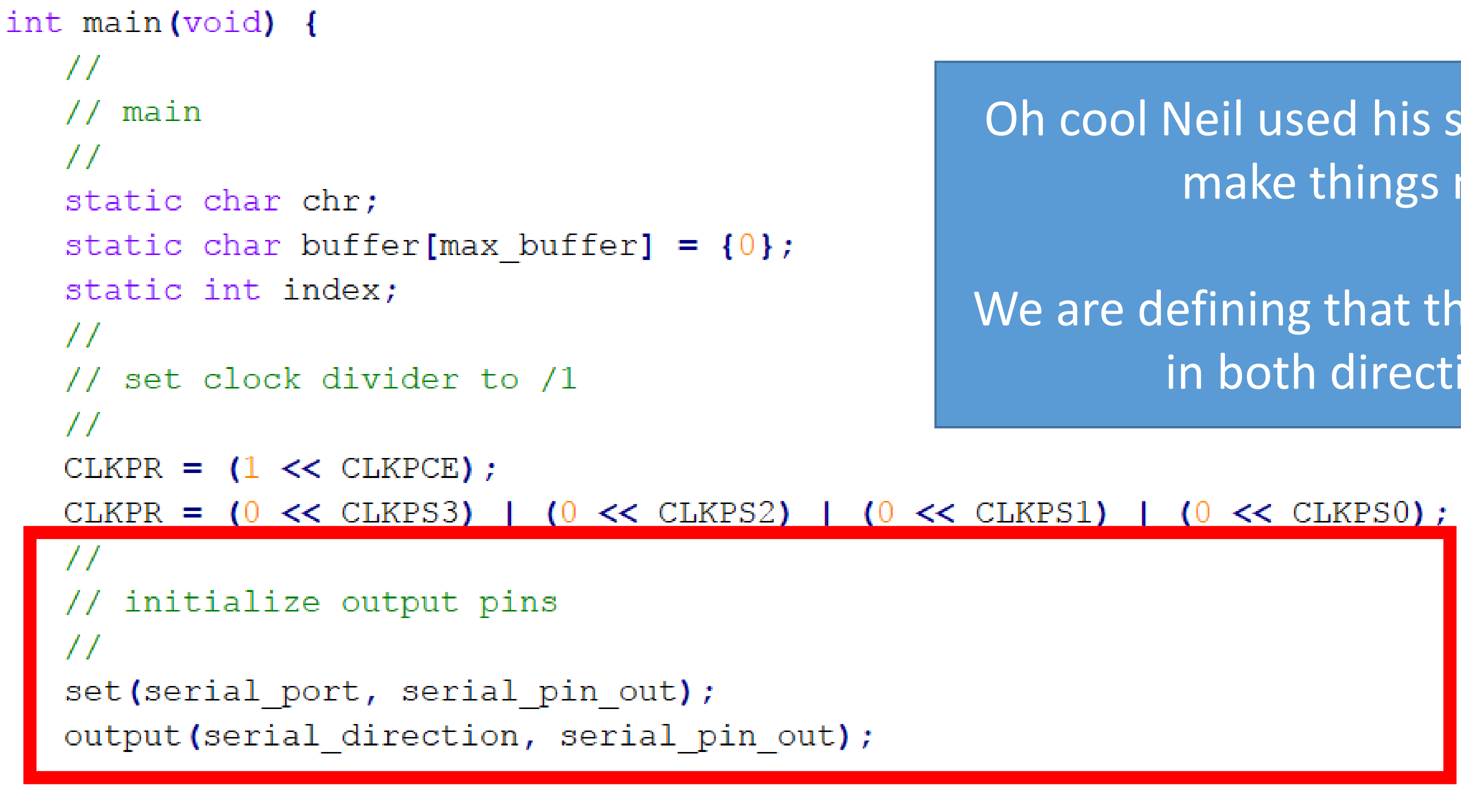
In the C language types MATTER. It will not compile without correct types.



```c
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;
   //
   // set clock divider to /1
   //
   CLKPR = (1 << CLKPCE);
   CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
   //
   // initialize output pins
   //
   set(serial_port, serial_pin_out);
   output(serial_direction, serial_pin_out);
```

“Hmmm this looks scary and I don’t think this program is doing anything crazy with timing or clocks so I’m just going to leave that as is.”

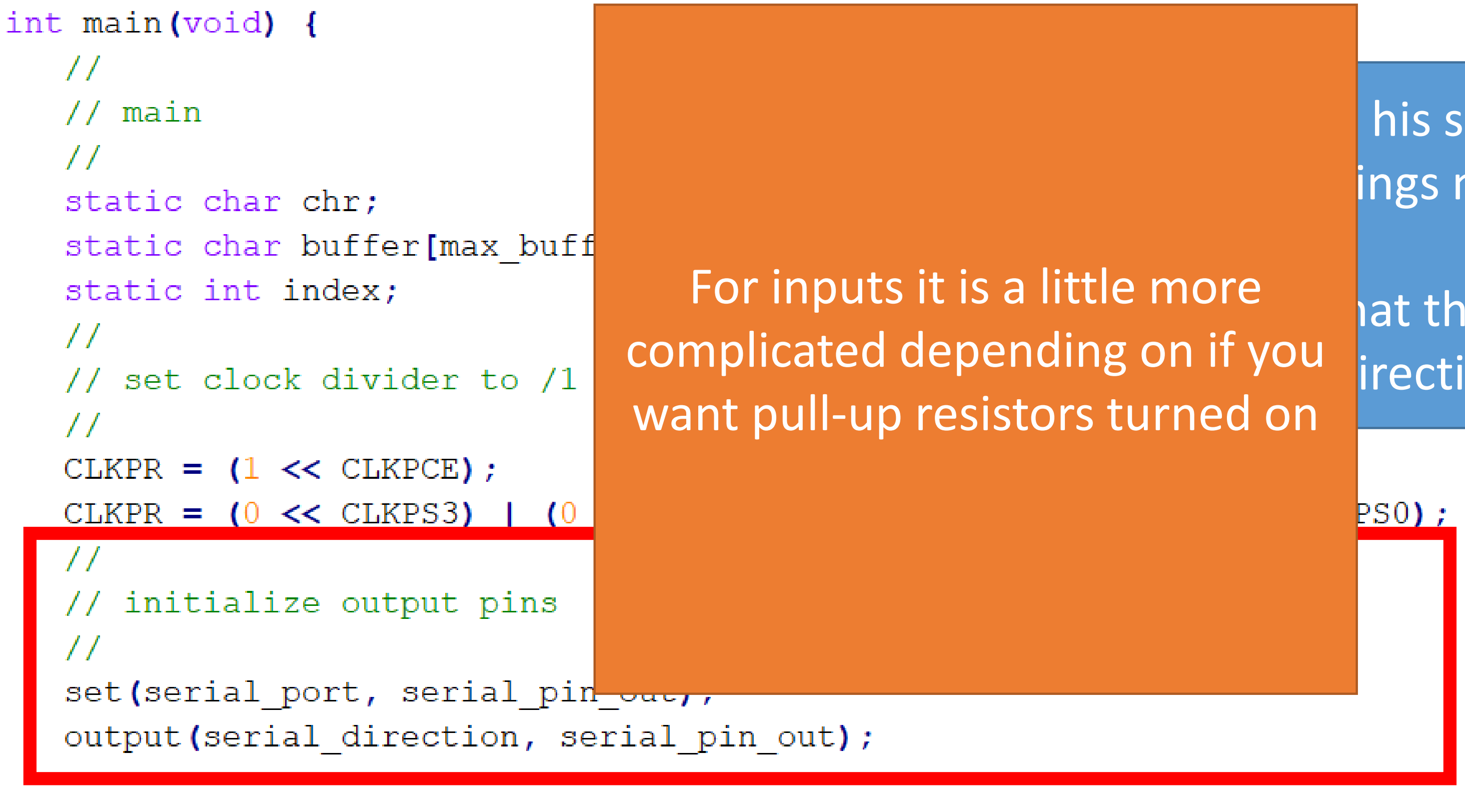
We can talk about it later



```c
int main(void) {
    //
    // main
    //
    static char chr;
    static char buffer[max_buffer] = {0};
    static int index;
    //
    // set clock divider to /1
    //
    CLKPR = (1 << CLKPCE);
    CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
    //
    // initialize output pins
    //
    set(serial_port, serial_pin_out);
    output(serial_direction, serial_pin_out);
```

Oh cool Neil used his shorthand #defines to make things make sense!

We are defining that the out pin is an output in both direction and port!

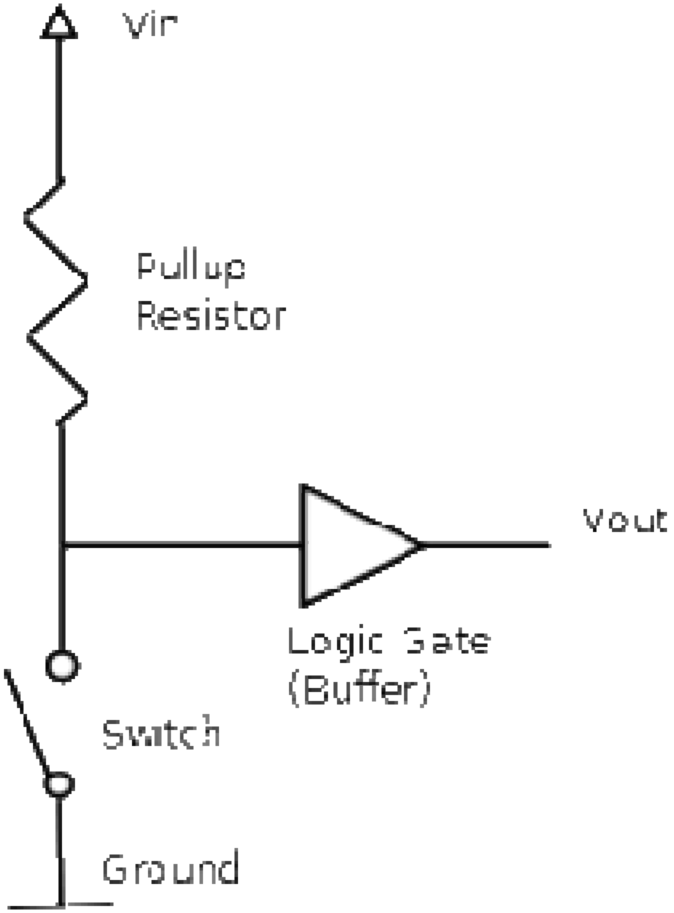


```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buff
   static int index;
   //
   // set clock divider to /1
   //
   CLKPR = (1 << CLKPCE);
   CLKPR = (0 << CLKPS3) | (0                                  PS0);
   //
   // initialize output pins
   //
   set(serial_port, serial_pin_out);
   output(serial_direction, serial_pin_out);
```

For inputs it is a little more complicated depending on if you want pull-up resistors turned on

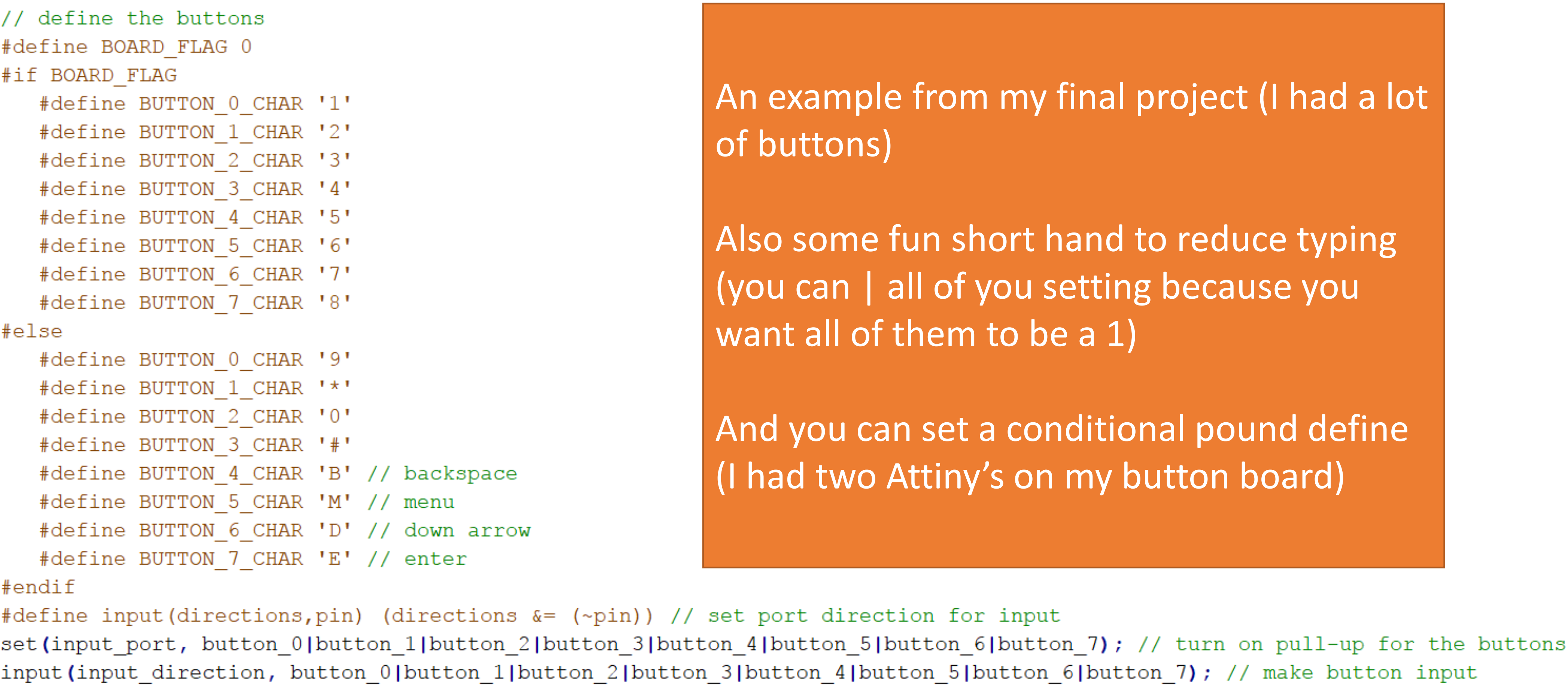
his shorthand #defines to
ings make sense!

hat the out pin is an output
irection and port!



Remember from last time if your input is a GND for a signal you need the pullup resistor!

cough button *cough*

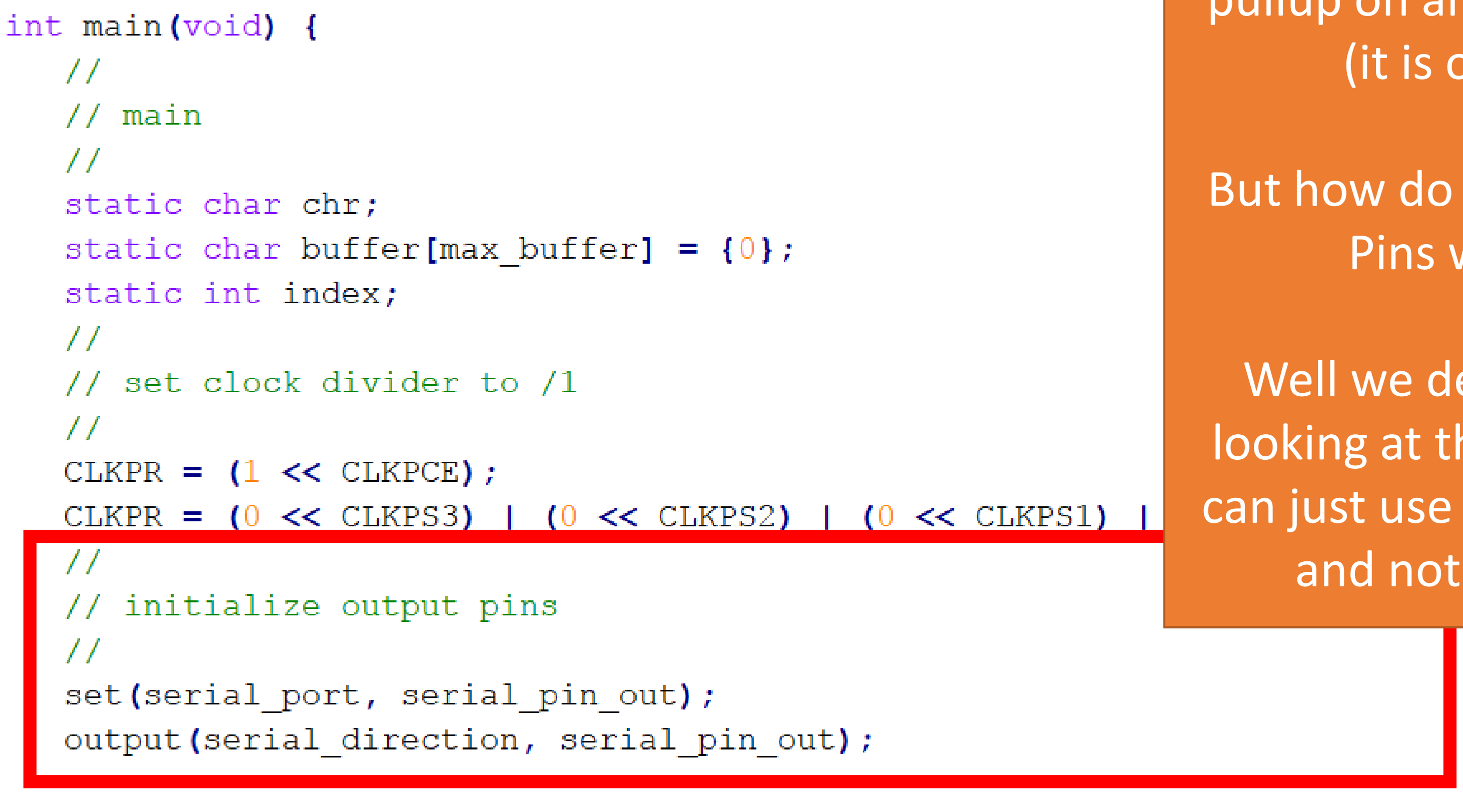


```
// define the buttons
#define BOARD_FLAG 0
#if BOARD_FLAG
   #define BUTTON_0_CHAR '1'
   #define BUTTON_1_CHAR '2'
   #define BUTTON_2_CHAR '3'
   #define BUTTON_3_CHAR '4'
   #define BUTTON_4_CHAR '5'
   #define BUTTON_5_CHAR '6'
   #define BUTTON_6_CHAR '7'
   #define BUTTON_7_CHAR '8'
#else
   #define BUTTON_0_CHAR '9'
   #define BUTTON_1_CHAR '*'
   #define BUTTON_2_CHAR '0'
   #define BUTTON_3_CHAR '#'
   #define BUTTON_4_CHAR 'B' // backspace
   #define BUTTON_5_CHAR 'M' // menu
   #define BUTTON_6_CHAR 'D' // down arrow
   #define BUTTON_7_CHAR 'E' // enter
#endif
#define input(directions,pin) (directions &= (~pin)) // set port direction for input
set(input_port, button_0|button_1|button_2|button_3|button_4|button_5|button_6|button_7); // turn on pull-up for the buttons
input(input_direction, button_0|button_1|button_2|button_3|button_4|button_5|button_6|button_7); // make button input
```

An example from my final project (I had a lot of buttons)

Also some fun short hand to reduce typing (you can | all of you setting because you want all of them to be a 1)

And you can set a conditional pound define (I had two Attiny's on my button board)

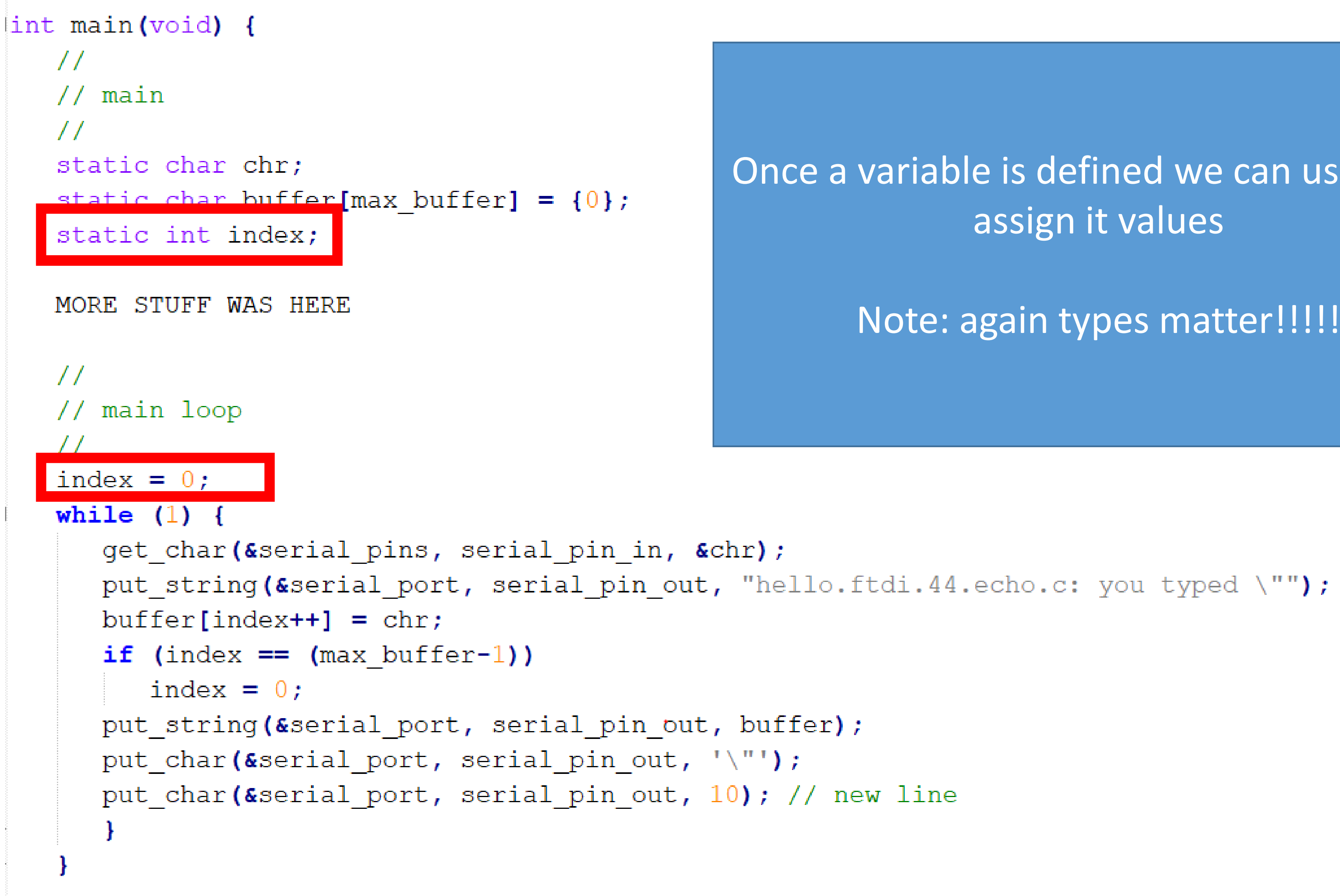


```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;
   //
   // set clock divider to /1
   //
   CLKPR = (1 << CLKPCE);
   CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) |
   //
   // initialize output pins
   //
   set(serial_port, serial_pin_out);
   output(serial_direction, serial_pin_out);
```

In this case the computer sends us values so we don't want the pullup on and so we do nothing (it is off by default)

But how do we tell what Ports / Pins we are using?

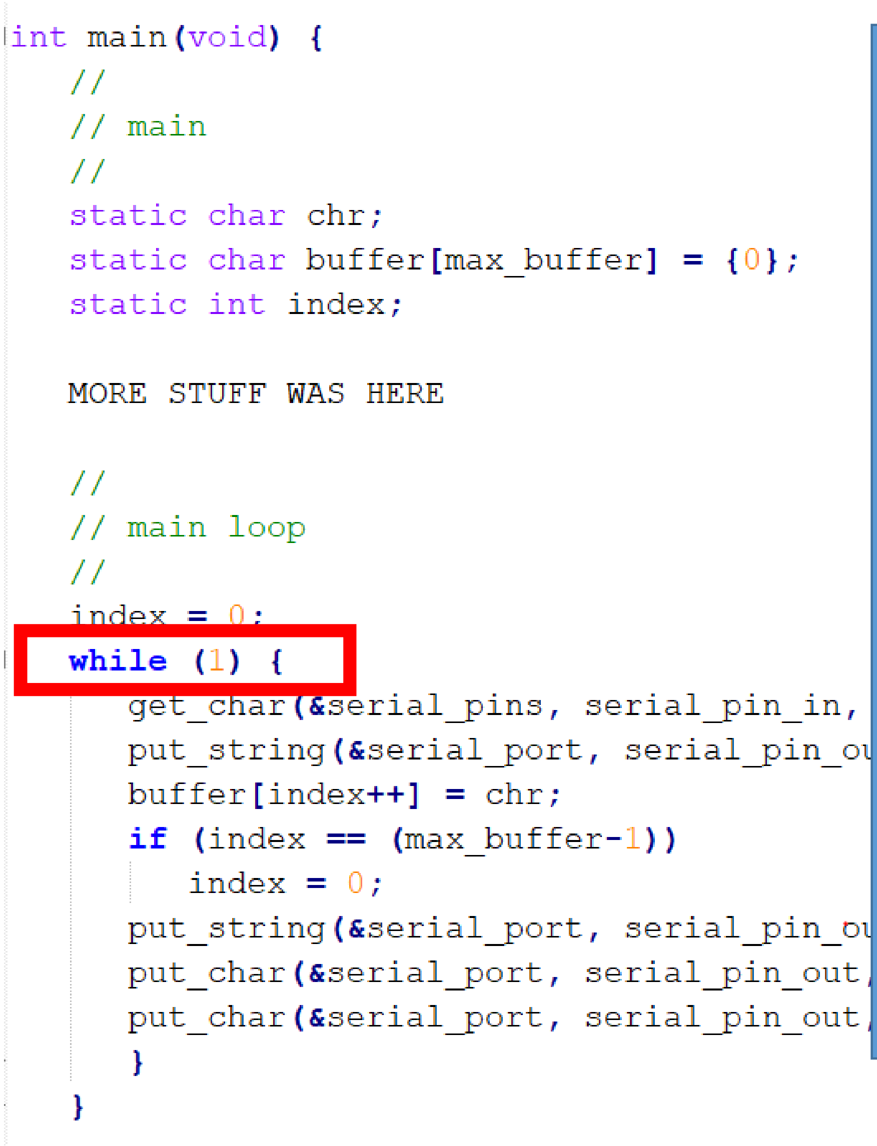
Well we defined it before by looking at the data sheet so we can just use our #defined values and not worry about it!



```c
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed \"");
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

Once a variable is defined we can use it and assign it values

Note: again types matter!!!!!



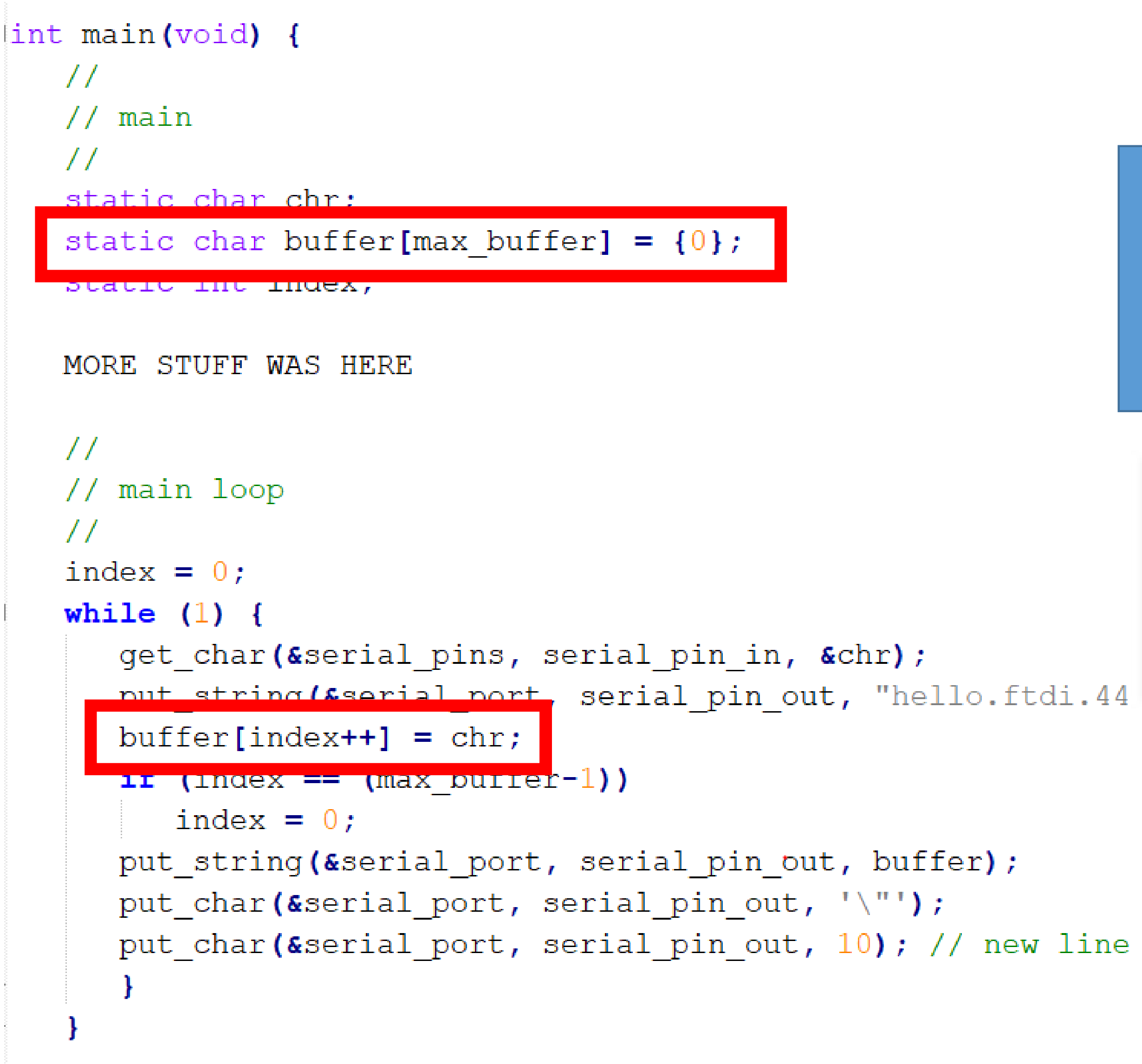
```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in,
      put_string(&serial_port, serial_pin_ou
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_ou
      put_char(&serial_port, serial_pin_out,
      put_char(&serial_port, serial_pin_out,
      }
   }
```

"While" defines a LOOP (can also use "for")

This is a core programming concept in C – we do things repetitively in loops and branch on conditional statements "if" and "else"

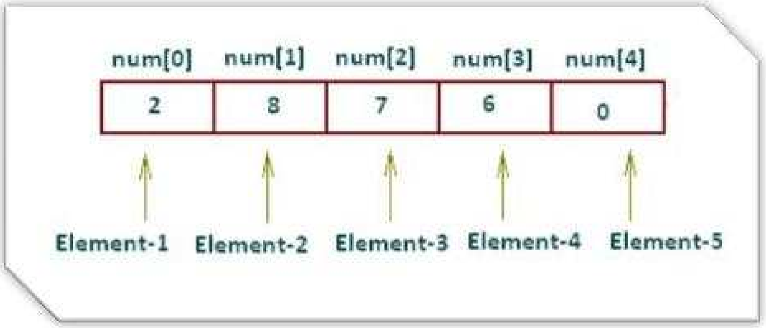
"While" will run until the condition in the "()" is FALSE so in this case it runs forever → thus our Attiny will repeat this action forever (one loop this small can run thousands of times a second so it better run for a long time or it will be too fast for us humans).

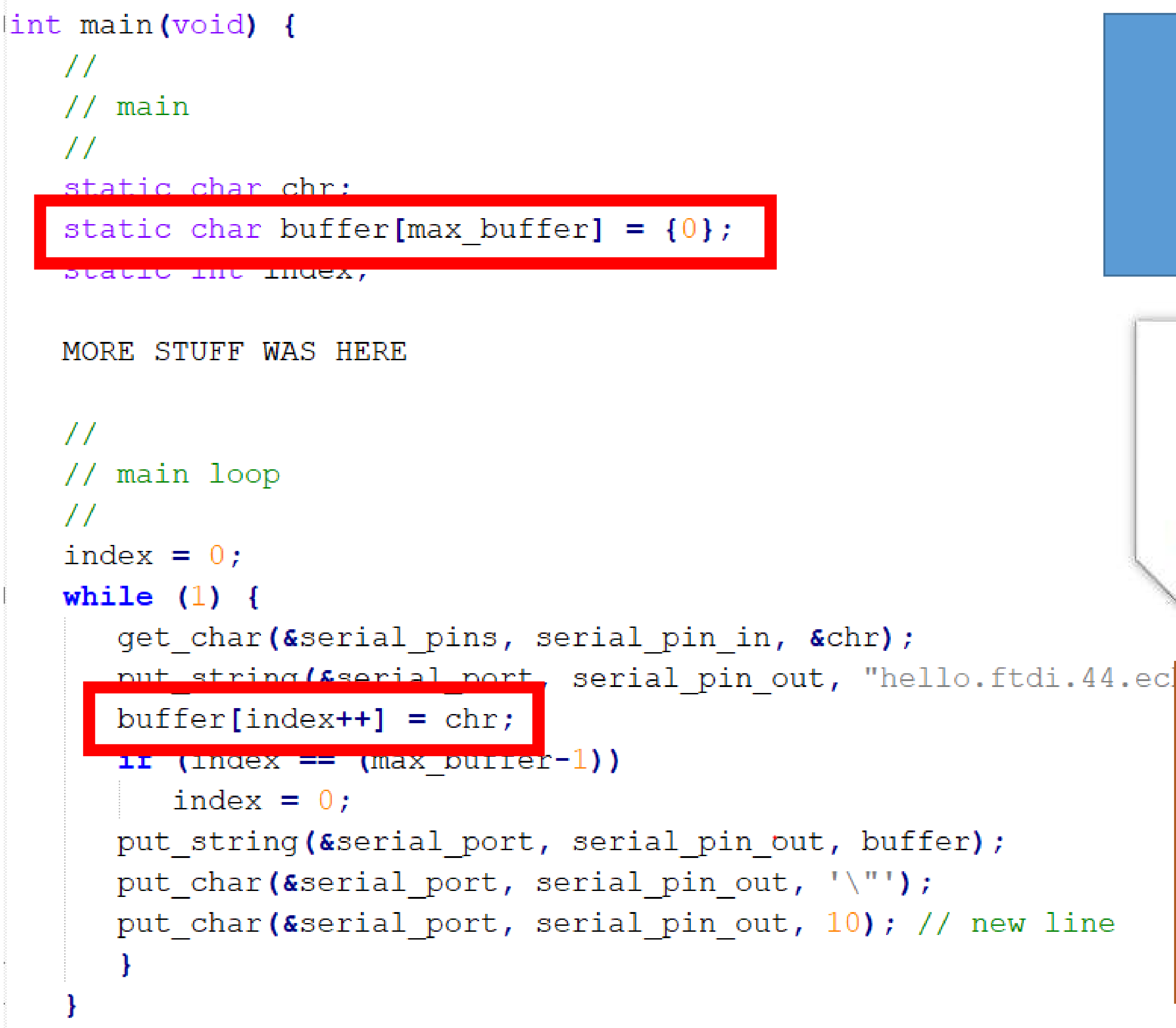
In general for AVR purposes we write all of the code that we want the AVR to do inside the while(1) loop



```
int main(void) {
    //
    // main
    //
    static char chr;
    static char buffer[max_buffer] = {0};
    static int index;

    MORE STUFF WAS HERE

    //
    // main loop
    //
    index = 0;
    while (1) {
        get_char(&serial_pins, serial_pin_in, &chr);
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44
        buffer[index++] = chr;
        if (index == (max_buffer-1))
            index = 0;
        put_string(&serial_port, serial_pin_out, buffer);
        put_char(&serial_port, serial_pin_out, '\"');
        put_char(&serial_port, serial_pin_out, 10); // new line
        }
    }
```

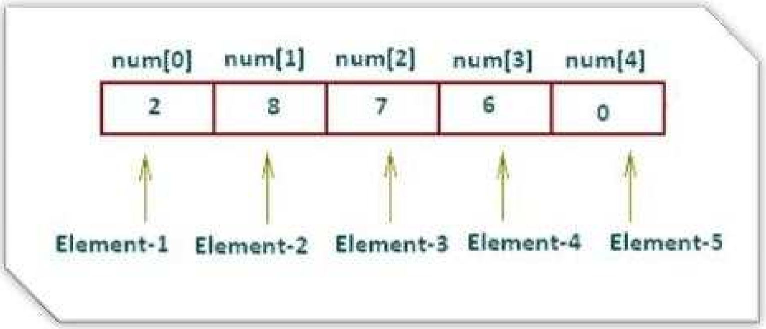
Buffer is an ARRAY (list) of char



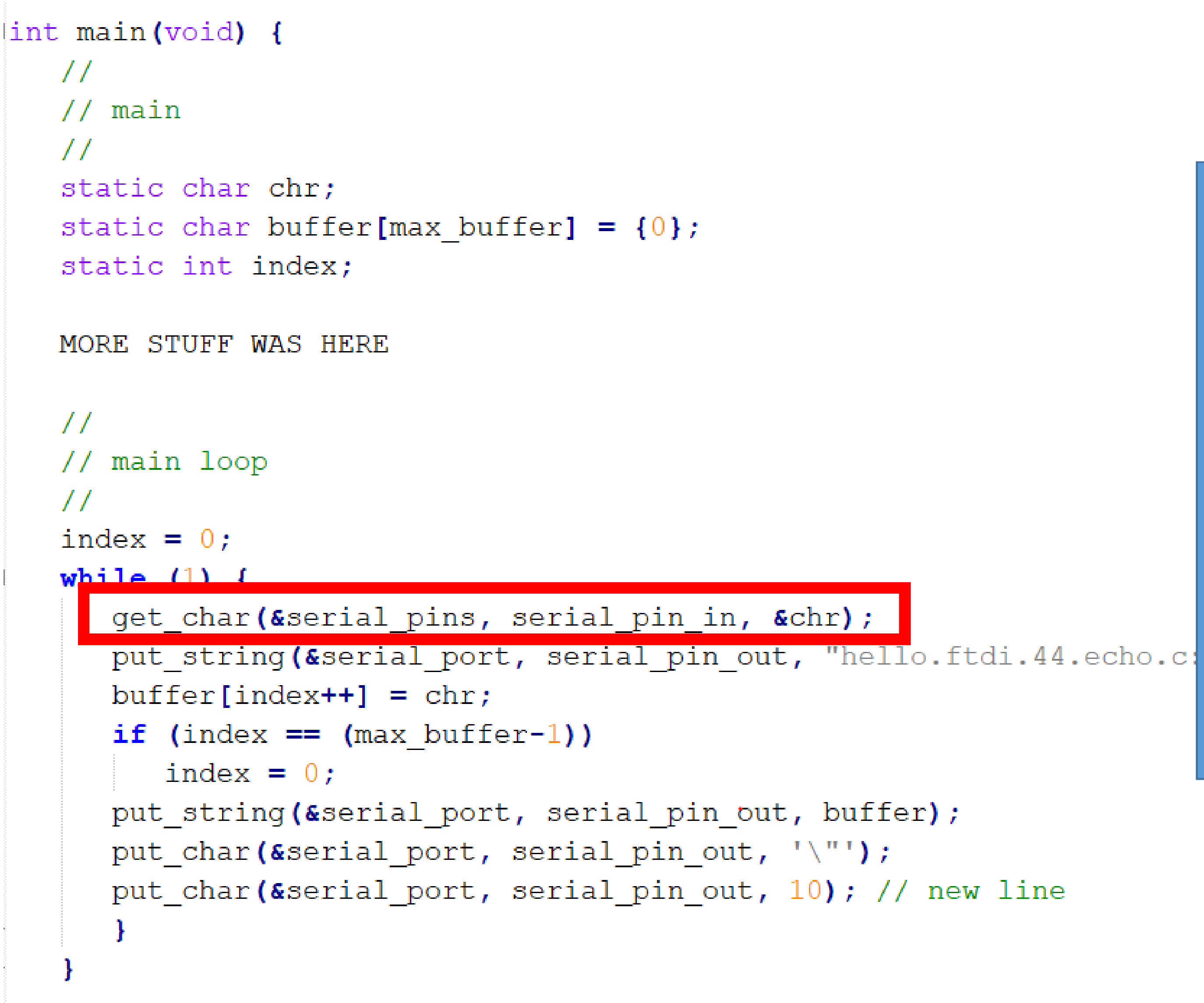


```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.ec
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

Buffer is an ARRAY (list) of char



++ is shorthand for:
buffer[index] = chr;
index = index + 1;

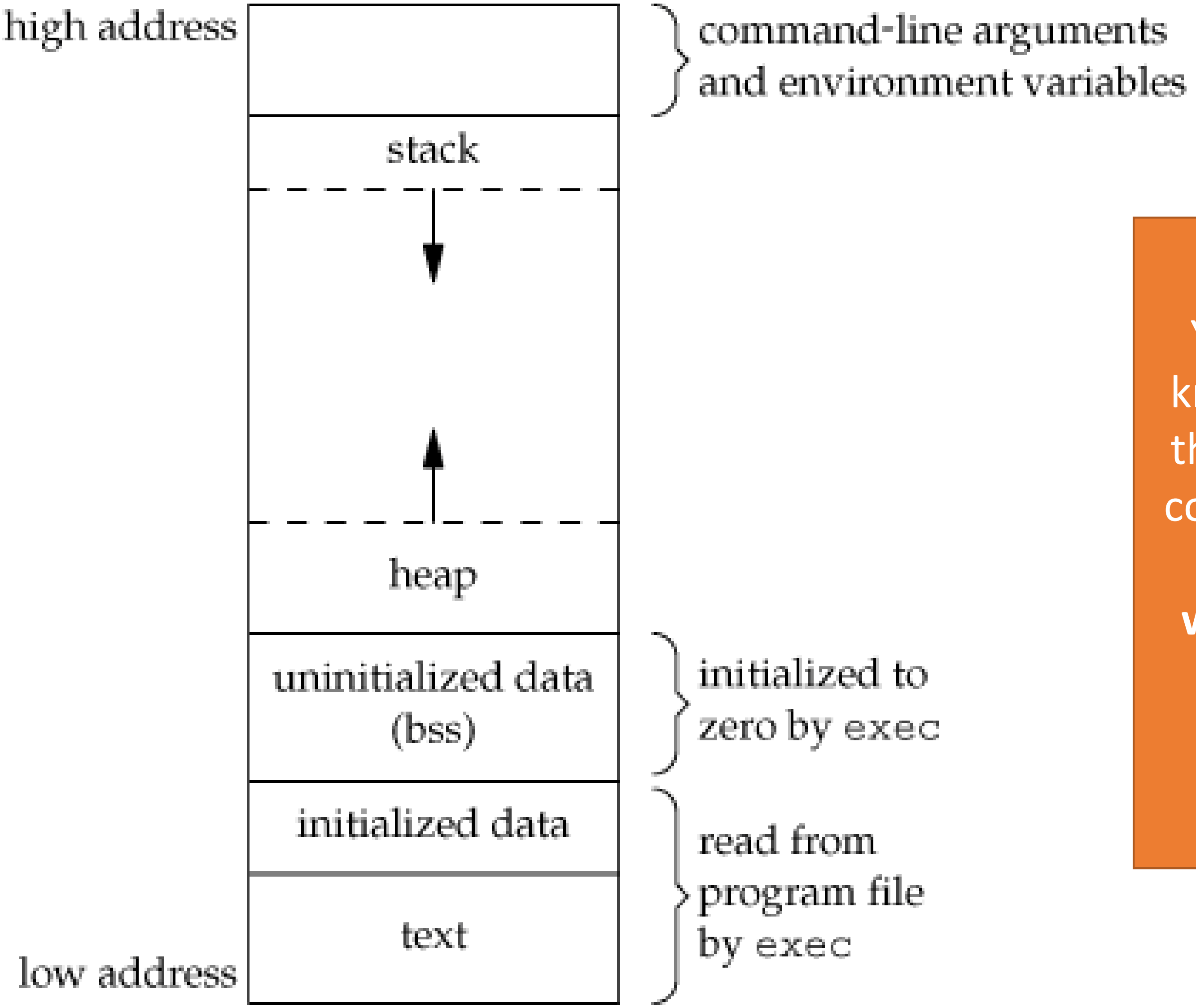


```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c:
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

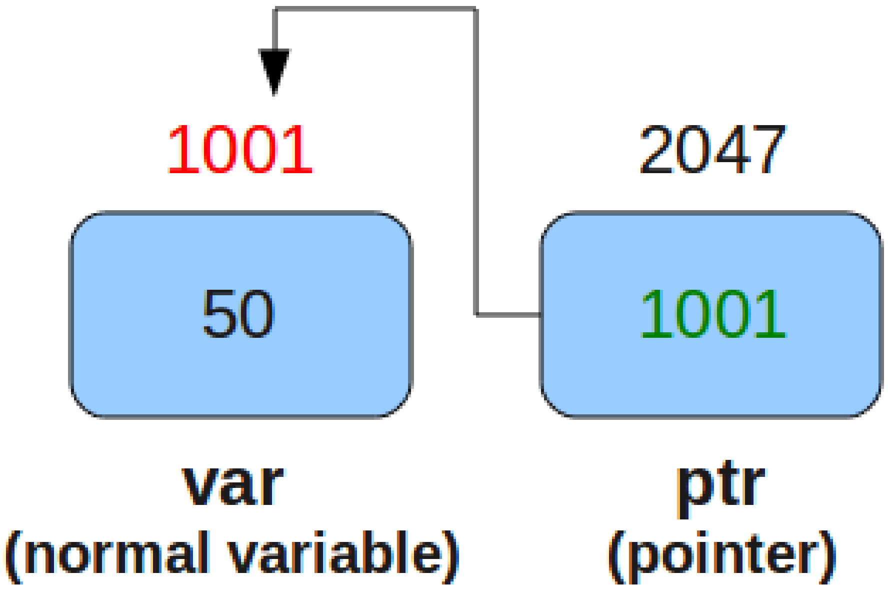
Let’s use Neil’s helper function to get a value from the computer and save it in our chr variable

What about the &s

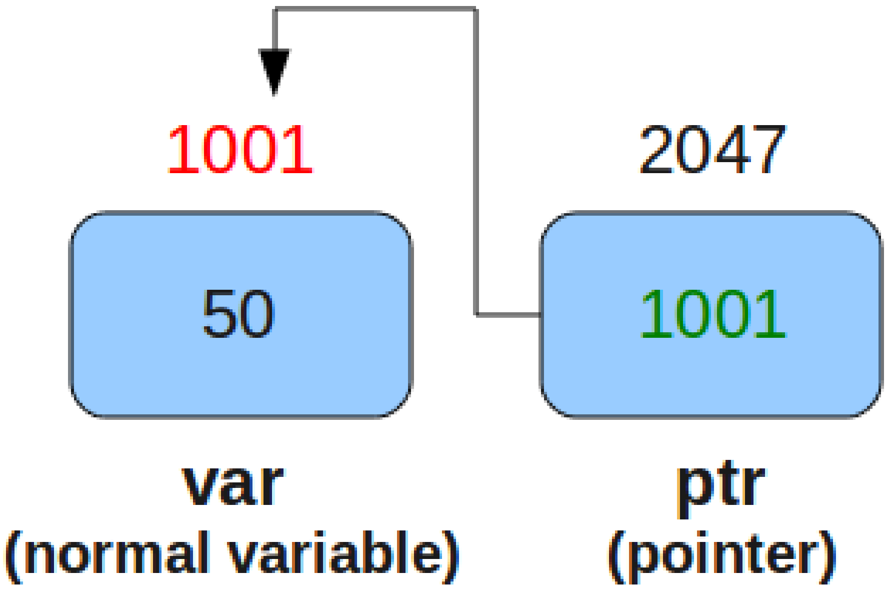
Pointer FUN?!


high address
low address
stack
heap
uninitialized data (bss)
initialized data
text
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

You don't really need to know this just understand that the memory layout is complex and **sometimes it is helpful to remember where you stored things and reference them indirectly**



```
var -> 50     // the variable itself has the value 50
ptr -> 1001   // the value of the ptr is the address of what it points
              // to and therefore since it points to var it is 1001
&var -> 1001  // & operator gets us the adress of that variable
*ptr -> 50    // * operator evaluates a pointer to get the value
              // at this address
*(&var) -> 50 // The value at the address of var is just its value
```



Hmm this is a little complicated do I need to remember all of this right now?

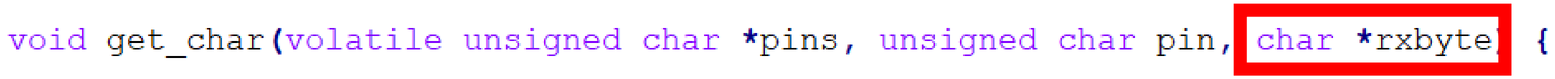
```
var -> 50     // the variable itself has the value 50
ptr -> 1001   // the value of the ptr is the address of what it points
              // to and therefore since it points to var it is 1001
&var -> 1001  // & operator gets us the adress of that variable
*ptr -> 50    // * operator evaluates a pointer to get the value
              // at this address
*(&var) -> 50 // The value at the address of var is just its value
```

Hmm this is a

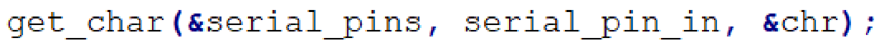
able)

Not really just work off of the example code and copy the patterns but if you get confused later when you are doing some advanced code creation this slide is helpful!

```
va          itse
pt          the                                              ts
            fore
&v          ets
*p          valu
                    // at this address
*(&var) -> 50 // The value at the address of var is just its value
```



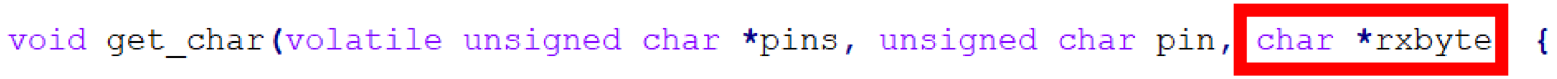
```
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte) {
```



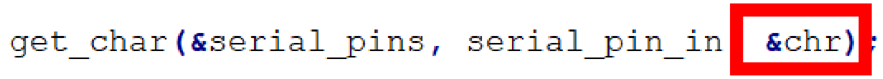
```
get_char(&serial_pins, serial_pin_in, &chr);
```

Looks like get_char wants a pointer variable type for the char it recieves

char *pins means pointer to a char (as a type)



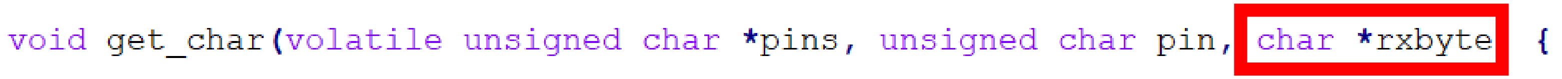
```
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte  {
```



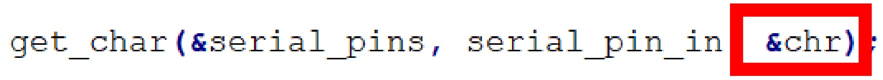
```
get_char(&serial_pins, serial_pin_in  &chr);
```

So lets pass it the address of our local chr variable so it can save it there

Remember a pointer is really just an address!



```
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte  {
```



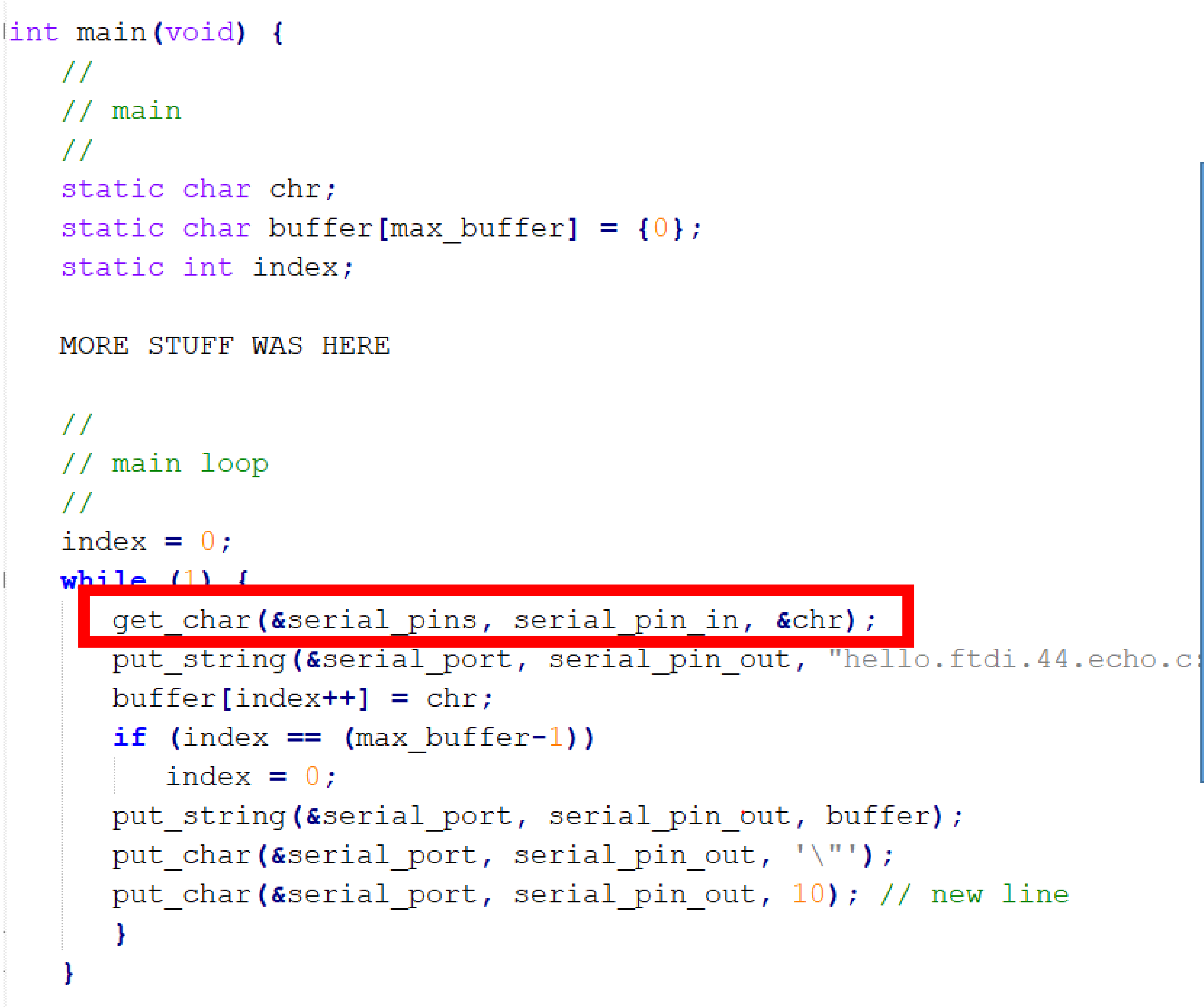
```
get_char(&serial_pins, serial_pin_in  &chr);
```

So lets pass it the address of our local chr variable so it can save it there

Remember a pointer is really just an address!

Ok but this still seems scary –oh wait we have Neil's example code and WE CAN JUST BASE OUR CODE ON HIS FOR NOW UNTIL WE FULLY UNDERSTAND IT!!!!

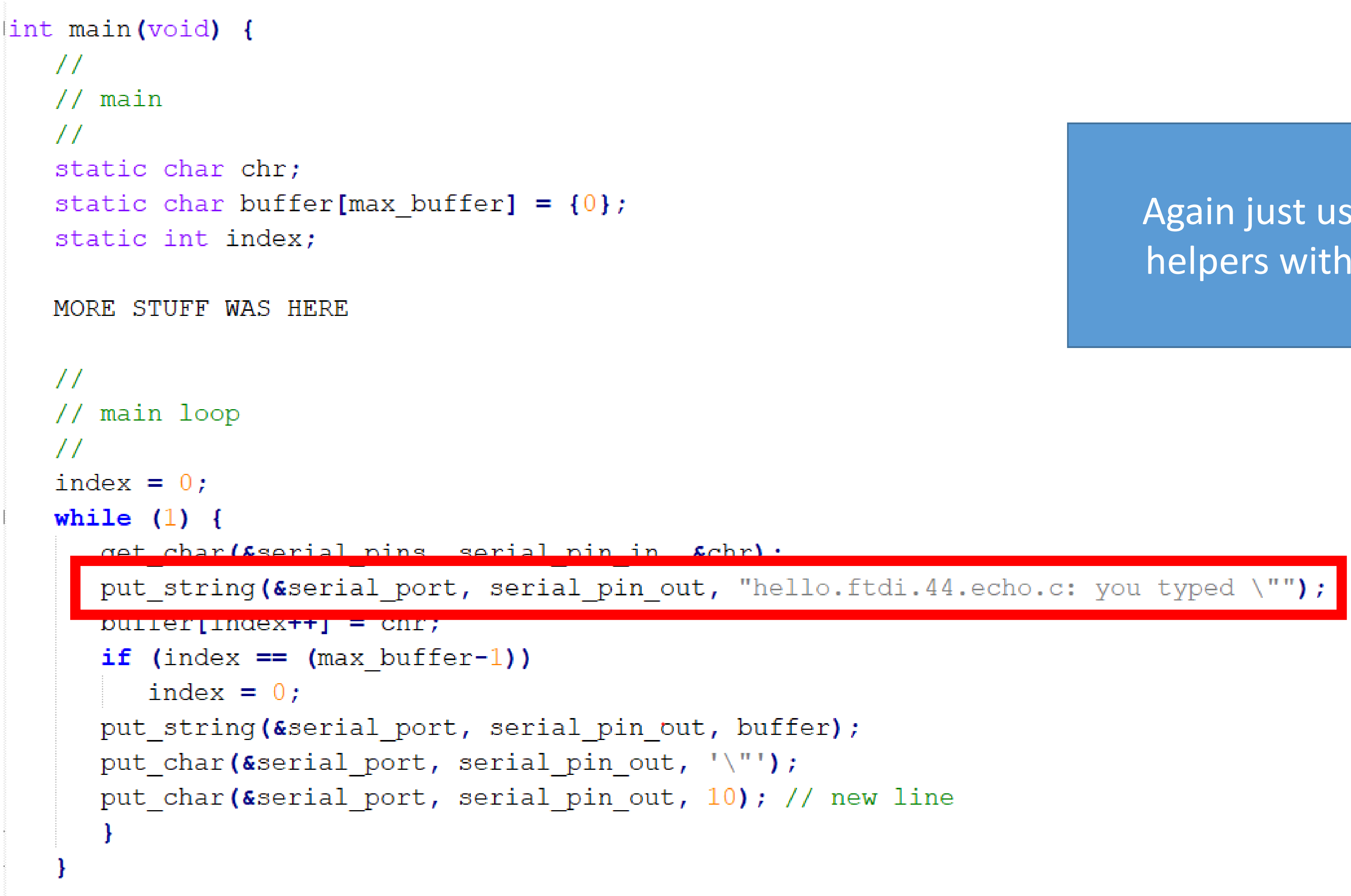
:-)



```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c:
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

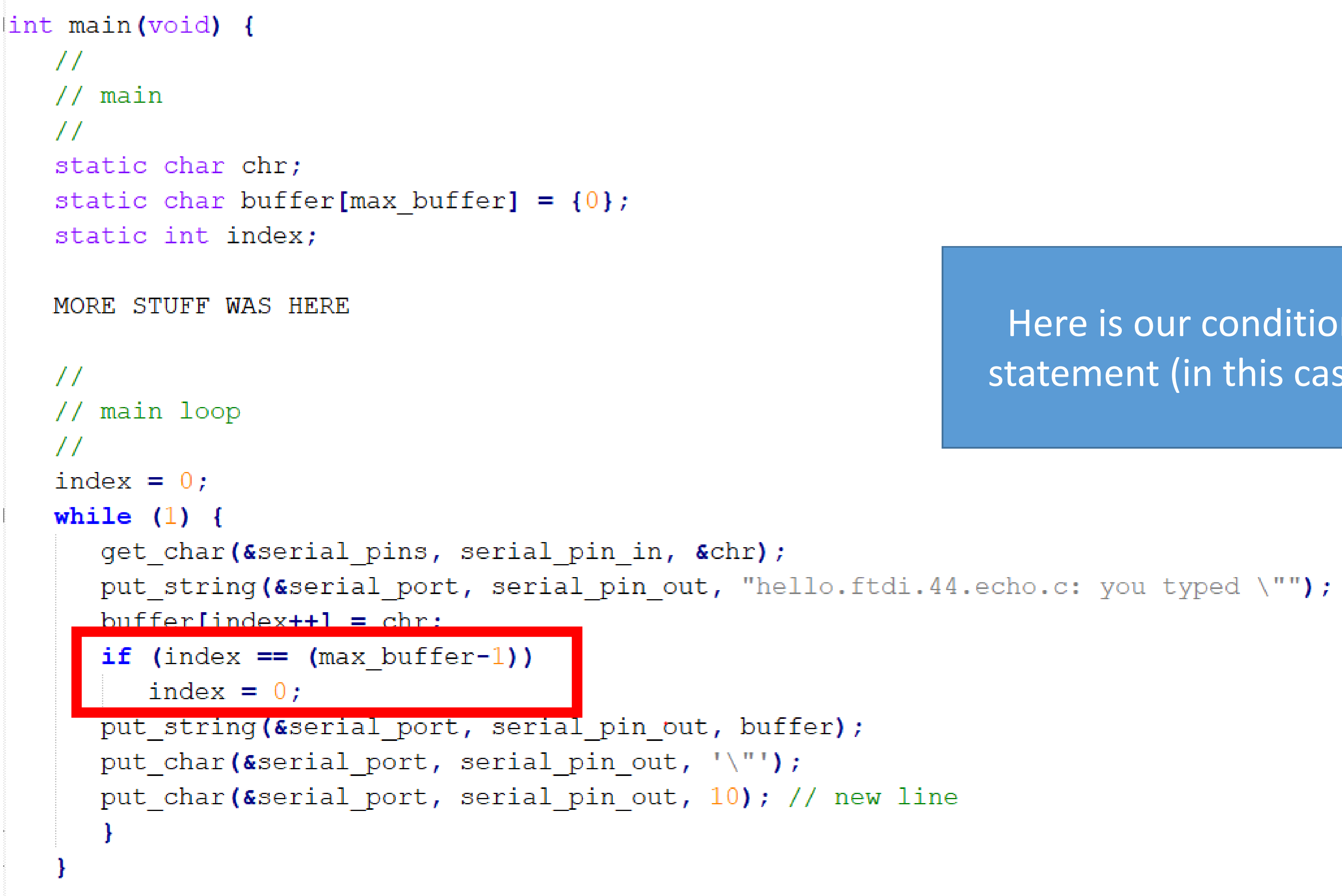
Ok so the & thing isn't that scary and the function definitions tell us what to pass things

We can use his examples for now and think about it over the next couple of weeks to understand it better



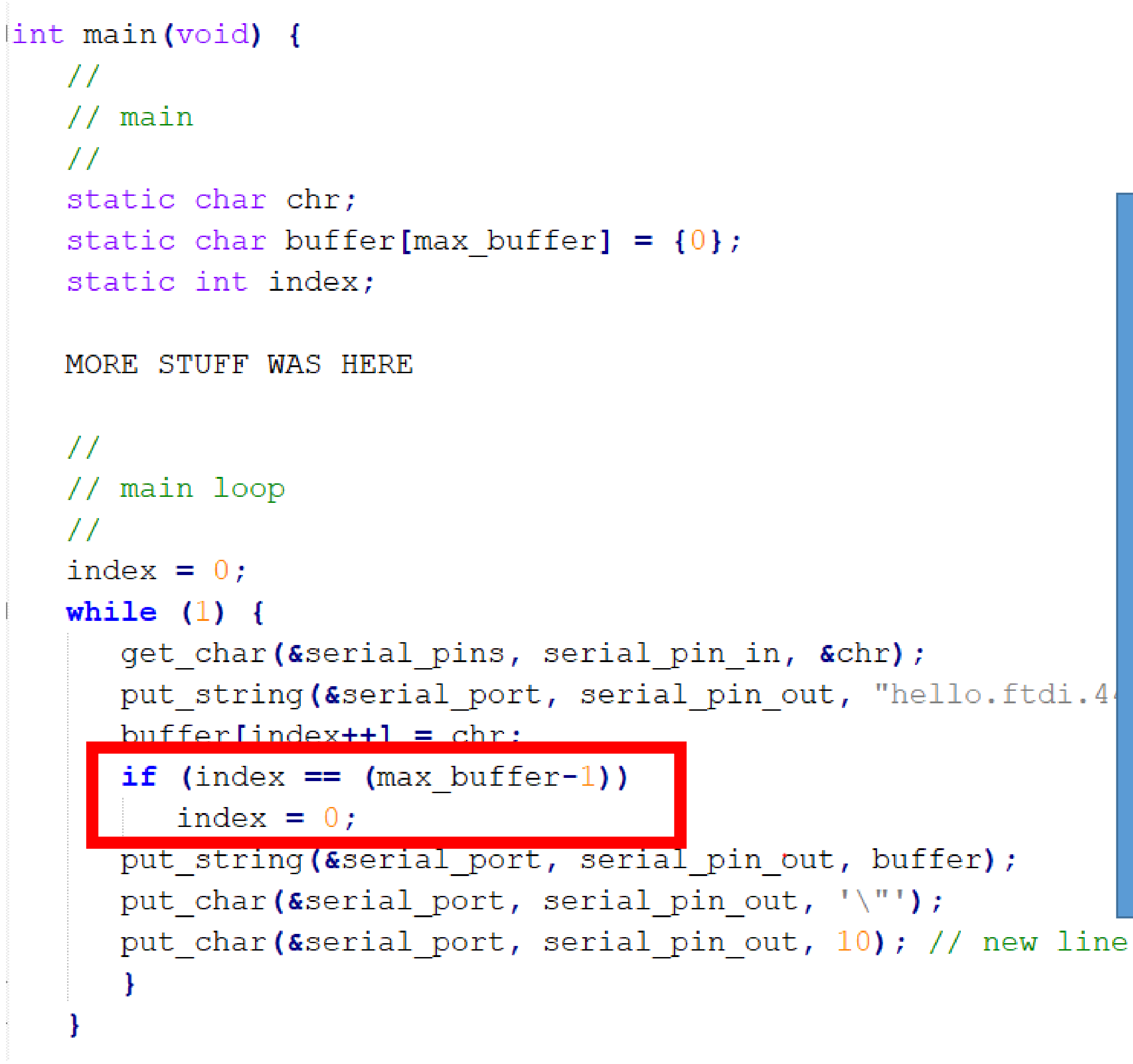
```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed \"");
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

Again just using Neil’s helpers with pointers



```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed \"");
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

Here is our conditional IF ELSE statement (in this case just an if)

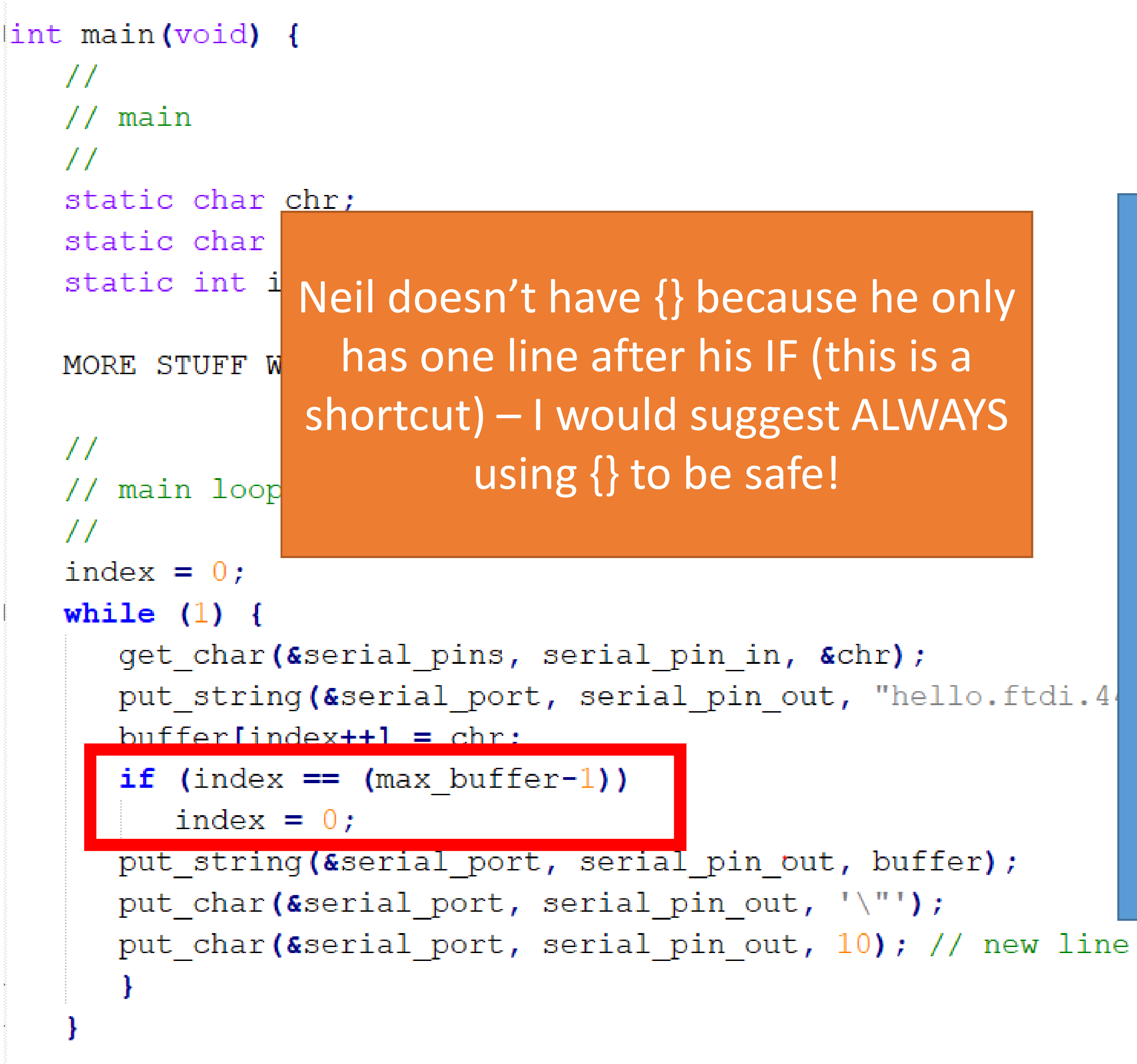


```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.4
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

Neil is using this to say if you reach the end of the buffer go back to the beginning and loop around!

This means if the buffer was length 4 and we added the alphabet in we would get:

[a,0,0,0] -> [a,b,0,0] -> [a,b,c,0] -> [a,b,c,d] -> [e,b,c,d] -> [e,f,c,d]



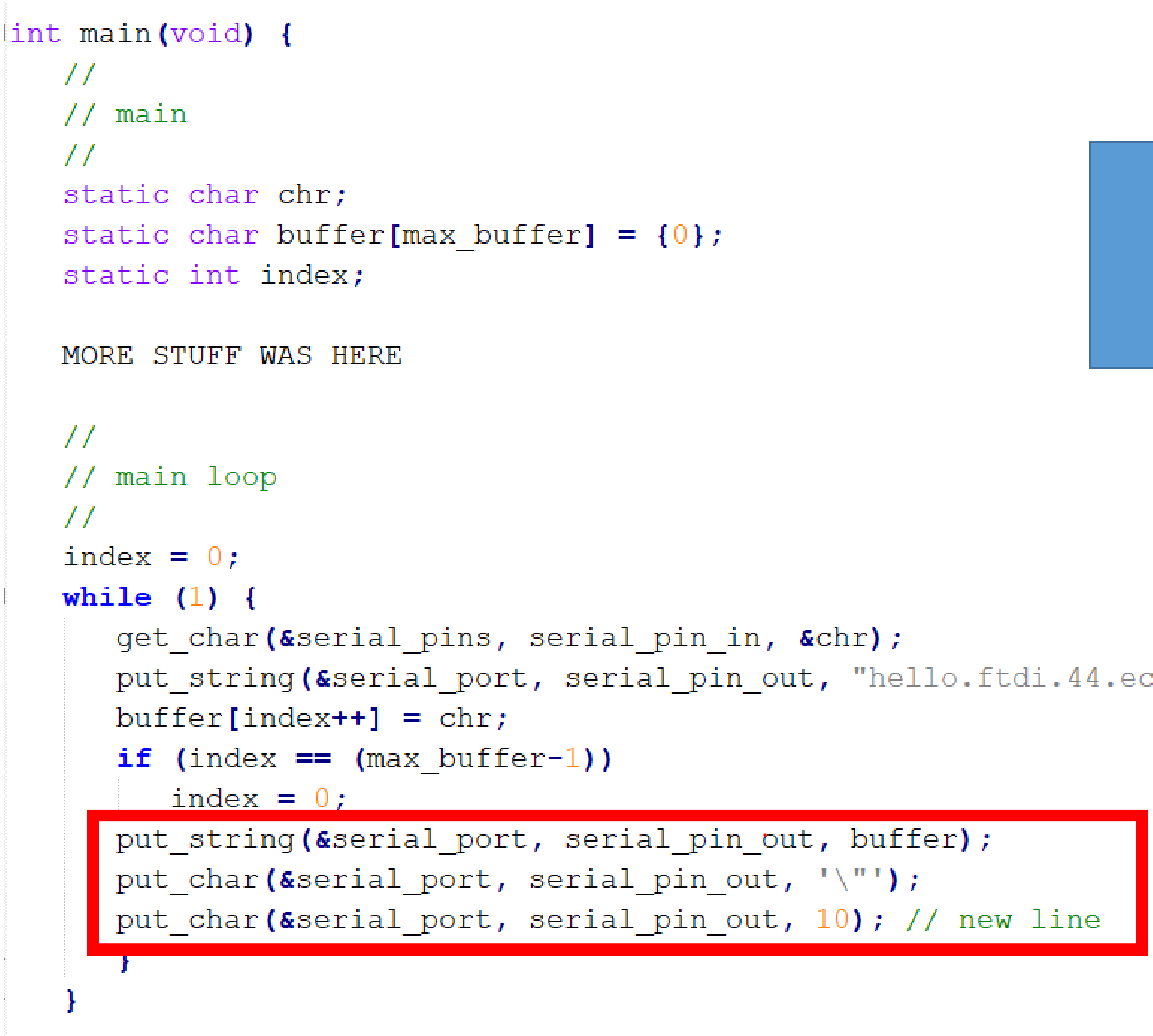
```
int main(void) {
   //
   // main
   //
   static char chr;
   static char
   static int i

   MORE STUFF W

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.4
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

Neil doesn't have {} because he only has one line after his IF (this is a shortcut) – I would suggest ALWAYS using {} to be safe!

Neil is using this to say if you reach the end of the buffer go back to the beginning and loop around!

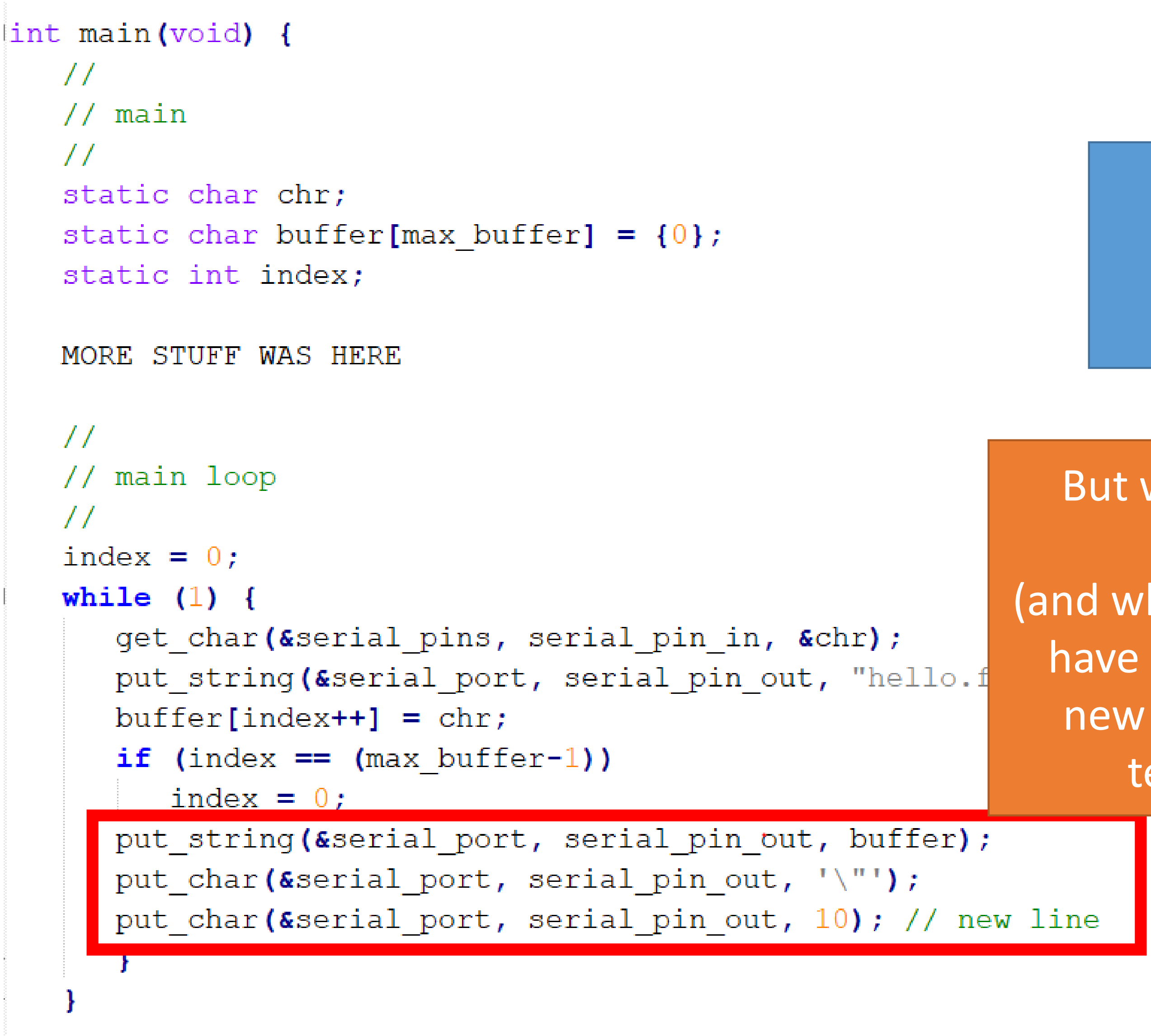
This means if the buffer was length 4 and we added the alphabet in we would get:

[a,0,0,0] -> [a,b,0,0] -> [a,b,c,0] -> [a,b,c,d] -> [e,b,c,d] -> [e,f,c,d]



```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed \"");
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

More Neil functions and we are done!

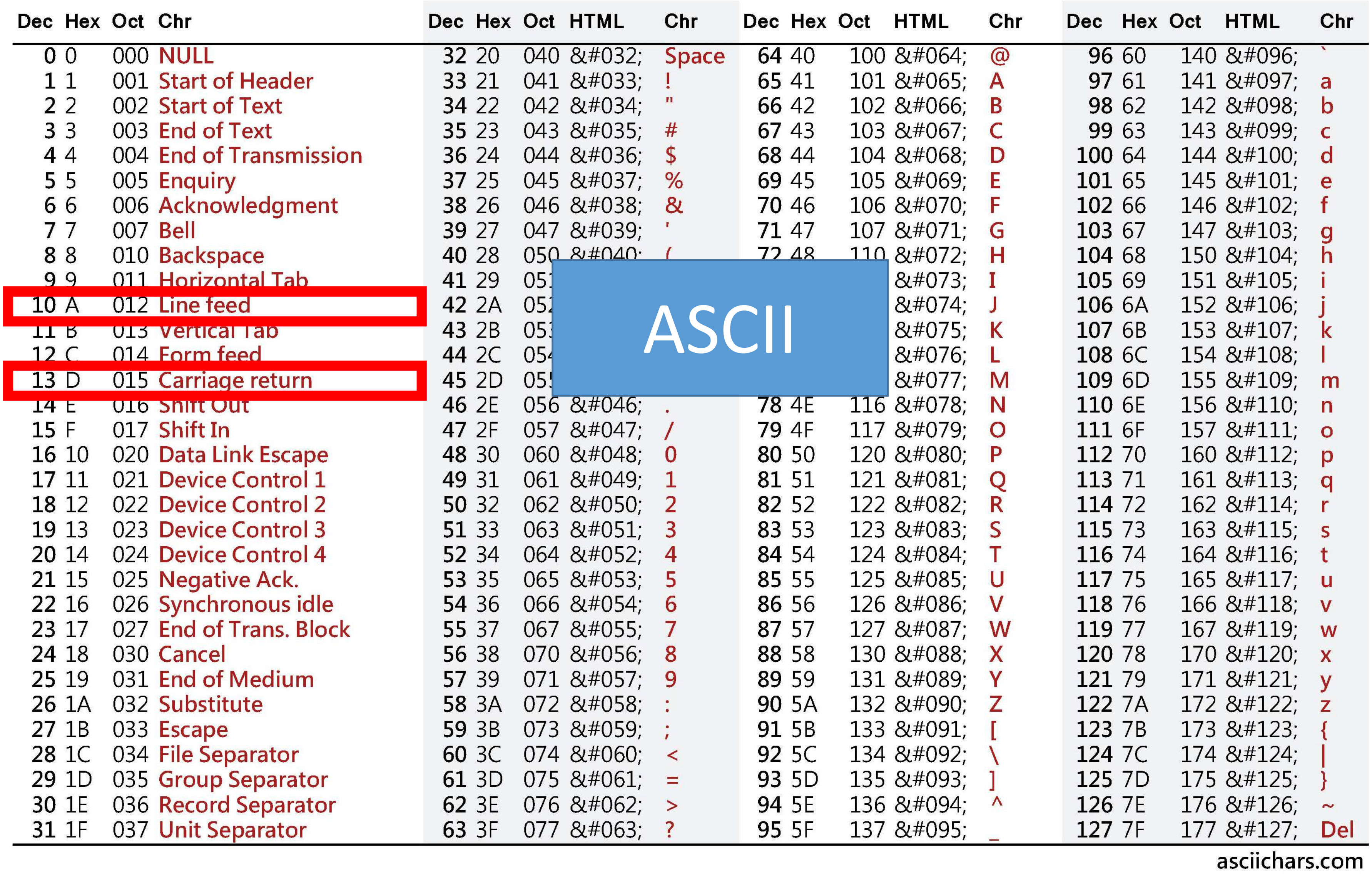


```
int main(void) {
   //
   // main
   //
   static char chr;
   static char buffer[max_buffer] = {0};
   static int index;

   MORE STUFF WAS HERE

   //
   // main loop
   //
   index = 0;
   while (1) {
      get_char(&serial_pins, serial_pin_in, &chr);
      put_string(&serial_port, serial_pin_out, "hello.f
      buffer[index++] = chr;
      if (index == (max_buffer-1))
         index = 0;
      put_string(&serial_port, serial_pin_out, buffer);
      put_char(&serial_port, serial_pin_out, '\"');
      put_char(&serial_port, serial_pin_out, 10); // new line
      }
   }
```

More Neil functions and we are done!

But wait why is new line a 10?!?

(and why do windows computers not have the terminal actually go to a new line when you were testing term.py two weeks ago?)



ASCII

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	05	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

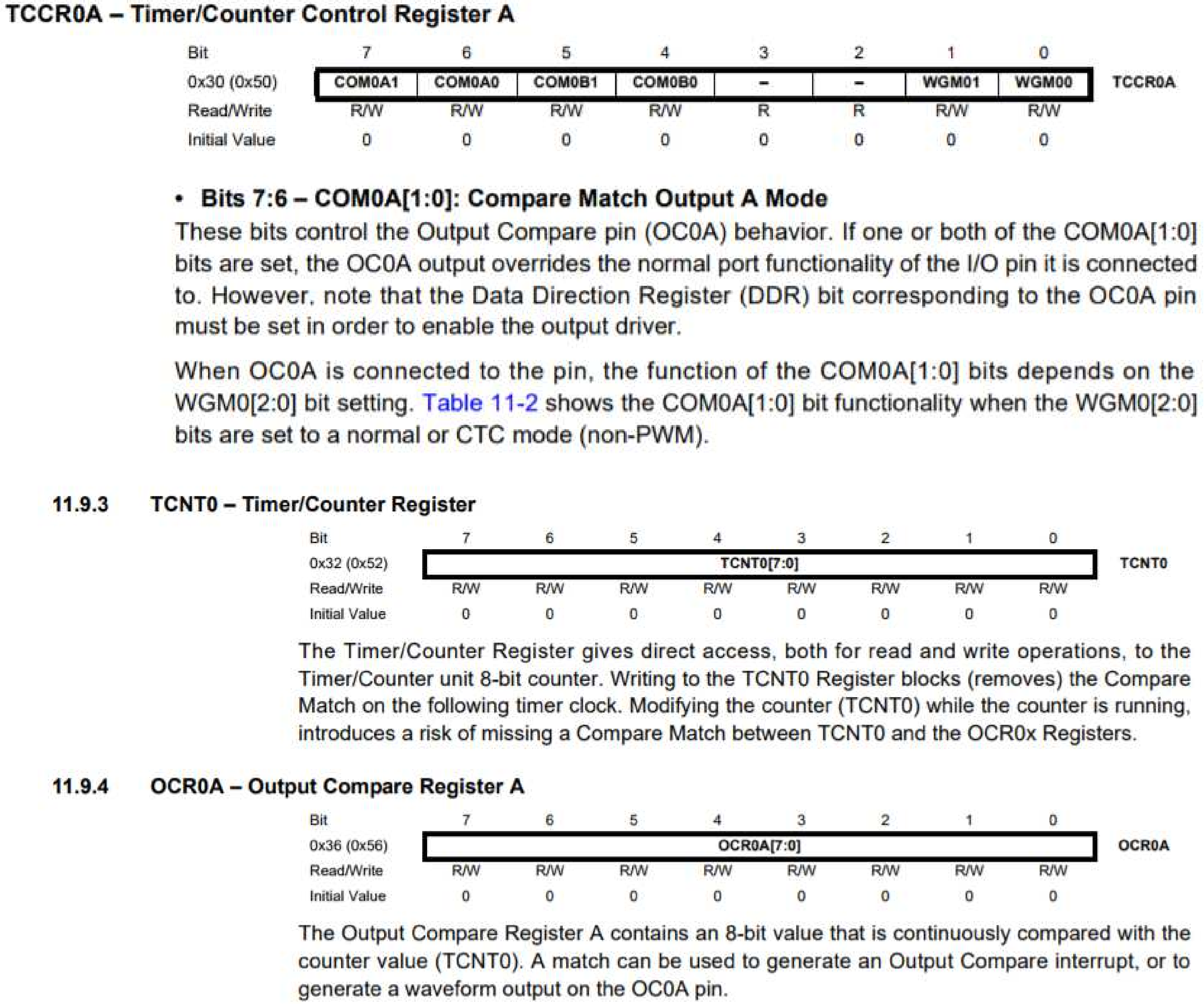
asciichars.com

Key things to make sure you are doing in your code!!

- USE BRACKETS {}
- USE SEMICOLONS ;
- All helper things come before Main
- GOOGLE IS YOUR FRIEND!

So what else is in that data sheet?

Timers and Clock Registers



TCCR0A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
0x30 (0x50)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7:6 – COM0A[1:0]: Compare Match Output A Mode**

These bits control the Output Compare pin (OC0A) behavior. If one or both of the COM0A[1:0] bits are set, the OC0A output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0A pin must be set in order to enable the output driver.

When OC0A is connected to the pin, the function of the COM0A[1:0] bits depends on the WGM0[2:0] bit setting. Table 11-2 shows the COM0A[1:0] bit functionality when the WGM0[2:0] bits are set to a normal or CTC mode (non-PWM).

11.9.3 TCNT0 – Timer/Counter Register

Bit	7	6	5	4	3	2	1	0	
0x32 (0x52)	TCNT0[7:0]								TCNT0
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	

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the Compare Match on the following timer clock. Modifying the counter (TCNT0) while the counter is running, introduces a risk of missing a Compare Match between TCNT0 and the OCR0x Registers.

11.9.4 OCR0A – Output Compare Register A

Bit	7	6	5	4	3	2	1	0	
0x36 (0x56)	OCR0A[7:0]								OCR0A
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	

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC0A pin.

Interrupts

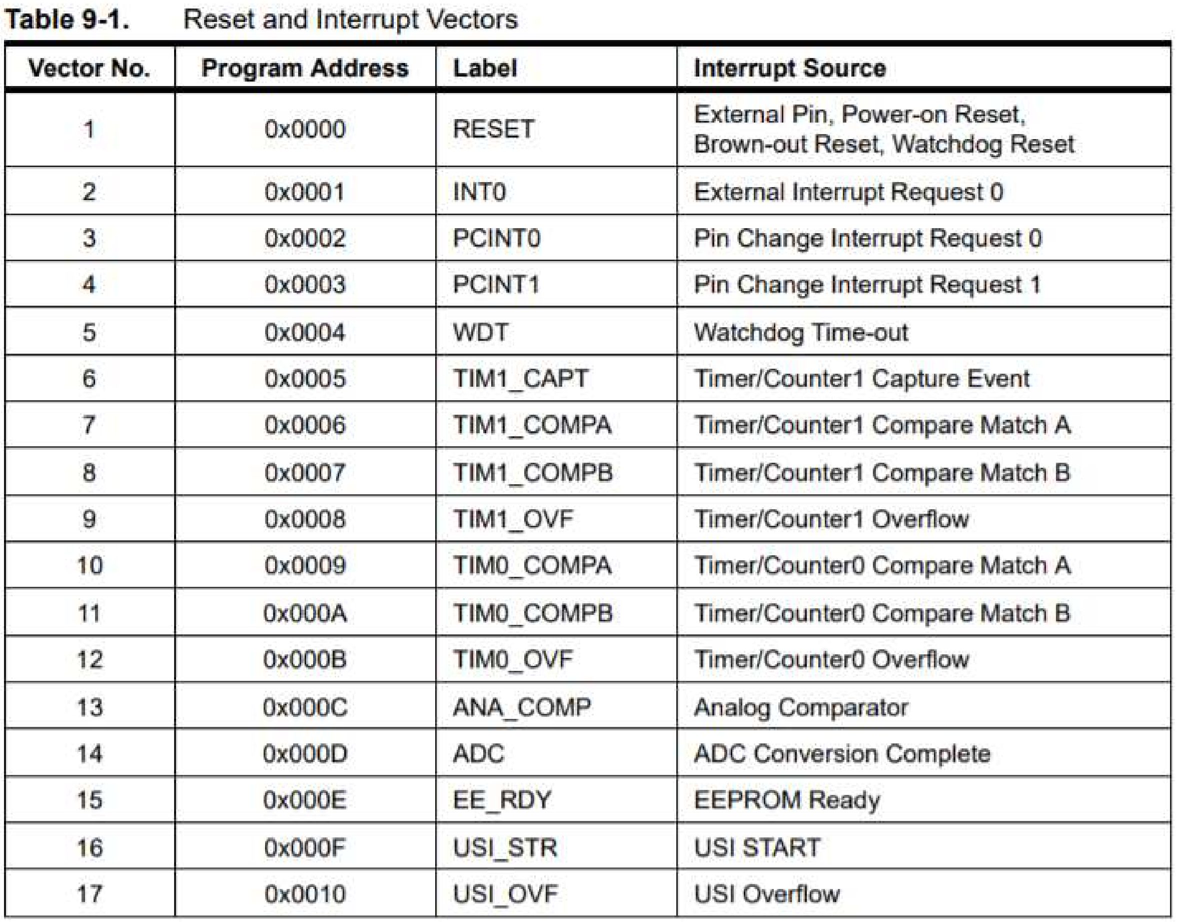
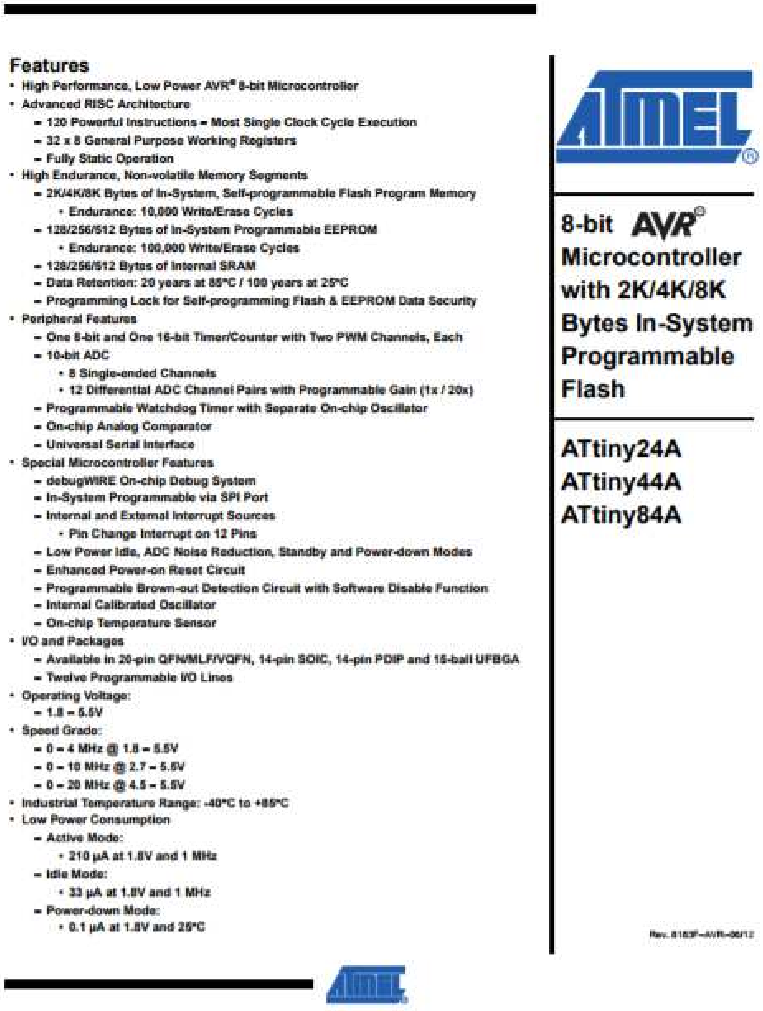


Table 9-1. Reset and Interrupt Vectors

Vector No.	Program Address	Label	Interrupt Source
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	PCINT1	Pin Change Interrupt Request 1
5	0x0004	WDT	Watchdog Time-out
6	0x0005	TIM1_CAPT	Timer/Counter1 Capture Event
7	0x0006	TIM1_COMPA	Timer/Counter1 Compare Match A
8	0x0007	TIM1_COMPB	Timer/Counter1 Compare Match B
9	0x0008	TIM1_OVF	Timer/Counter1 Overflow
10	0x0009	TIM0_COMPA	Timer/Counter0 Compare Match A
11	0x000A	TIM0_COMPB	Timer/Counter0 Compare Match B
12	0x000B	TIM0_OVF	Timer/Counter0 Overflow
13	0x000C	ANA_COMP	Analog Comparator
14	0x000D	ADC	ADC Conversion Complete
15	0x000E	EE_RDY	EEPROM Ready
16	0x000F	USI_STR	USI START
17	0x0010	USI_OVF	USI Overflow

http://academy.cba.mit.edu/classes/embedded_programming/doc8183.pdf



Features

- High Performance, Low Power AVR® 8-bit Microcontroller
- Advanced RISC Architecture
  - 120 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
- High Endurance, Non-volatile Memory Segments
  - 2K/4K/8K Bytes of In-System, Self-programmable Flash Program Memory
    - Endurance: 10,000 Write/Erase Cycles
  - 128/256/512 Bytes of In-System Programmable EEPROM
    - Endurance: 100,000 Write/Erase Cycles
  - 128/256/512 Bytes of Internal SRAM
  - Data Retention: 20 years at 85°C / 100 years at 25°C
  - Programming Lock for Self-programming Flash & EEPROM Data Security
- Peripheral Features
  - One 8-bit and One 16-bit Timer/Counter with Two PWM Channels, Each
  - 10-bit ADC
    - 8 Single-ended Channels
    - 12 Differential ADC Channel Pairs with Programmable Gain (1x / 20x)
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
  - Universal Serial Interface
- Special Microcontroller Features
  - debugWIRE On-chip Debug System
  - In-System Programmable via SPI Port
  - Internal and External Interrupt Sources
    - Pin Change Interrupt on 12 Pins
  - Low Power Idle, ADC Noise Reduction, Standby and Power-down Modes
  - Enhanced Power-on Reset Circuit
  - Programmable Brown-out Detection Circuit with Software Disable Function
  - Internal Calibrated Oscillator
  - On-chip Temperature Sensor
- I/O and Packages
  - Available in 20-pin QFN/MLF/VQFN, 14-pin SOIC, 14-pin PDIP and 15-ball UFBGA
  - Twelve Programmable I/O Lines
- Operating Voltage:
  - 1.8 – 5.5V
- Speed Grade:
  - 0 – 4 MHz @ 1.8 – 5.5V
  - 0 – 10 MHz @ 2.7 – 5.5V
  - 0 – 20 MHz @ 4.5 – 5.5V
- Industrial Temperature Range: -40°C to +85°C
- Low Power Consumption
  - Active Mode:
    - 210 µA at 1.8V and 1 MHz
  - Idle Mode:
    - 33 µA at 1.8V and 1 MHz
  - Power-down Mode:
    - 0.1 µA at 1.8V and 25°C

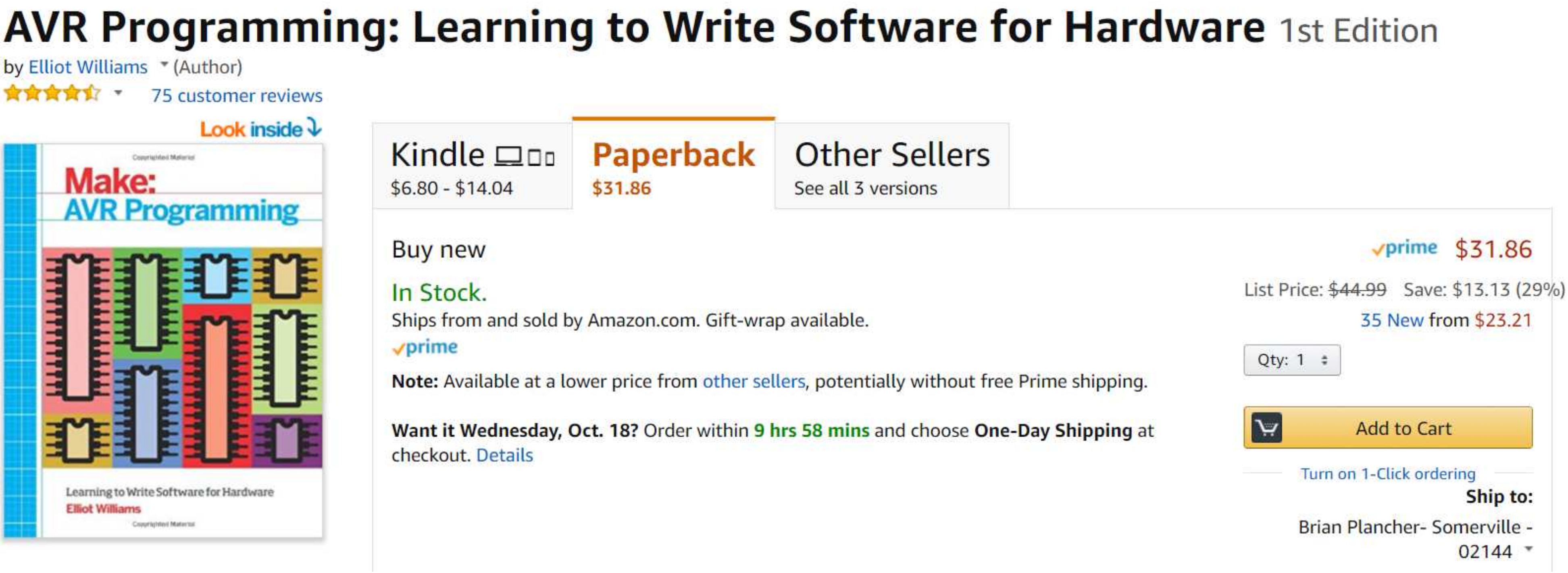
8-bit AVR® Microcontroller with 2K/4K/8K Bytes In-System Programmable Flash

ATtiny24A
ATtiny44A
ATtiny84A

Rev. 8183F–AVR–06/12

And so so so much more (e.g. ADC) so read up!
:-)

Embedded Programming



AVR Programming: Learning to Write Software for Hardware 1st Edition

by Elliot Williams (Author)

★★★★☆ 75 customer reviews

Kindle
$6.80 - $14.04

Paperback
$31.86

Other Sellers
See all 3 versions

Buy new

In Stock.

Ships from and sold by Amazon.com. Gift-wrap available.

✓prime

Note: Available at a lower price from other sellers, potentially without free Prime shipping.

Want it Wednesday, Oct. 18? Order within 9 hrs 58 mins and choose **One-Day Shipping** at checkout. Details

✓prime $31.86

List Price: ~~$44.99~~ Save: $13.13 (29%)

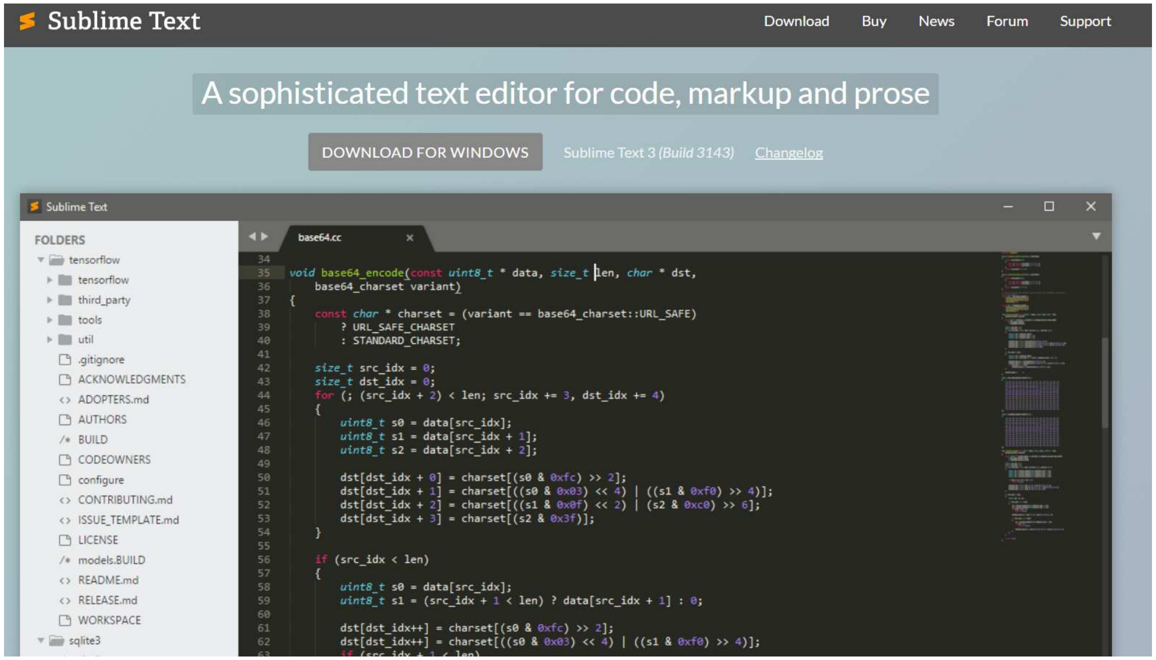
35 New from $23.21

Qty: 1

Ship to:

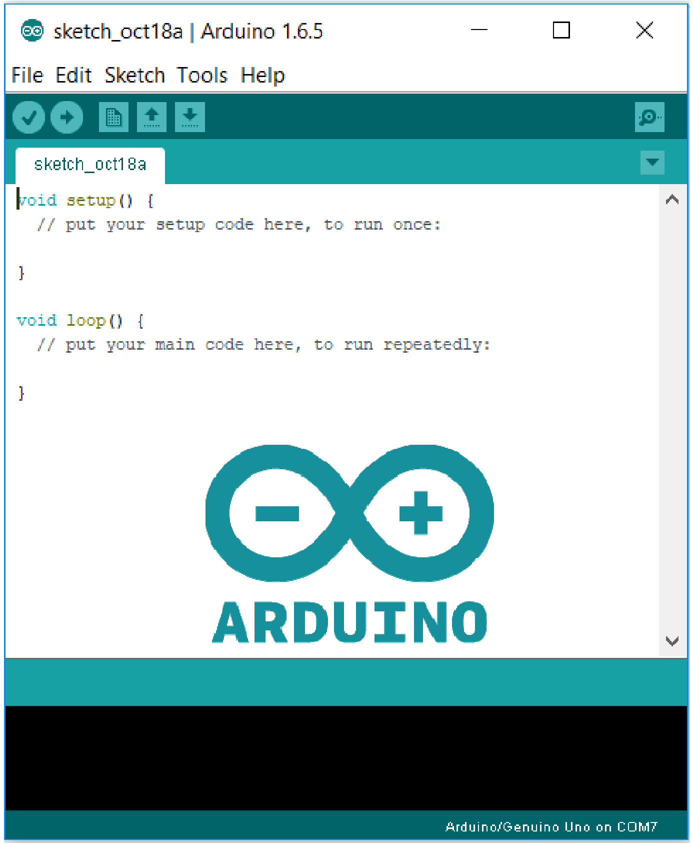
Brian Plancher- Somerville - 02144

Possible Lightweight Editors to Use (IDE)



A sophisticated text editor for code, markup and prose

```cpp

void base64_encode(const uint8_t * data, size_t len, char * dst,
    base64_charset variant)
{
    const char * charset = (variant == base64_charset::URL_SAFE)
        ? URL_SAFE_CHARSET
        : STANDARD_CHARSET;

    size_t src_idx = 0;
    size_t dst_idx = 0;
    for (; (src_idx + 2) < len; src_idx += 3, dst_idx += 4)
    {
        uint8_t s0 = data[src_idx];
        uint8_t s1 = data[src_idx + 1];
        uint8_t s2 = data[src_idx + 2];

        dst[dst_idx + 0] = charset[(s0 & 0xfc) >> 2];
        dst[dst_idx + 1] = charset[((s0 & 0x03) << 4) | ((s1 & 0xf0) >> 4)];
        dst[dst_idx + 2] = charset[((s1 & 0x0f) << 2) | (s2 & 0xc0) >> 6];
        dst[dst_idx + 3] = charset[(s2 & 0x3f)];
    }

    if (src_idx < len)
    {
        uint8_t s0 = data[src_idx];
        uint8_t s1 = (src_idx + 1 < len) ? data[src_idx + 1] : 0;

        dst[dst_idx++] = charset[(s0 & 0xfc) >> 2];
        dst[dst_idx++] = charset[((s0 & 0x03) << 4) | ((s1 & 0xf0) >> 4)];
        if (src_idx + 1 < len)
```


sketch_oct18a | Arduino 1.6.5
File Edit Sketch Tools Help
sketch_oct18a
ARDUINO
Arduino/Genuino Uno on COM7

```cpp
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Everything is harder on windows → Linux VM

And we’re done!

Questions?