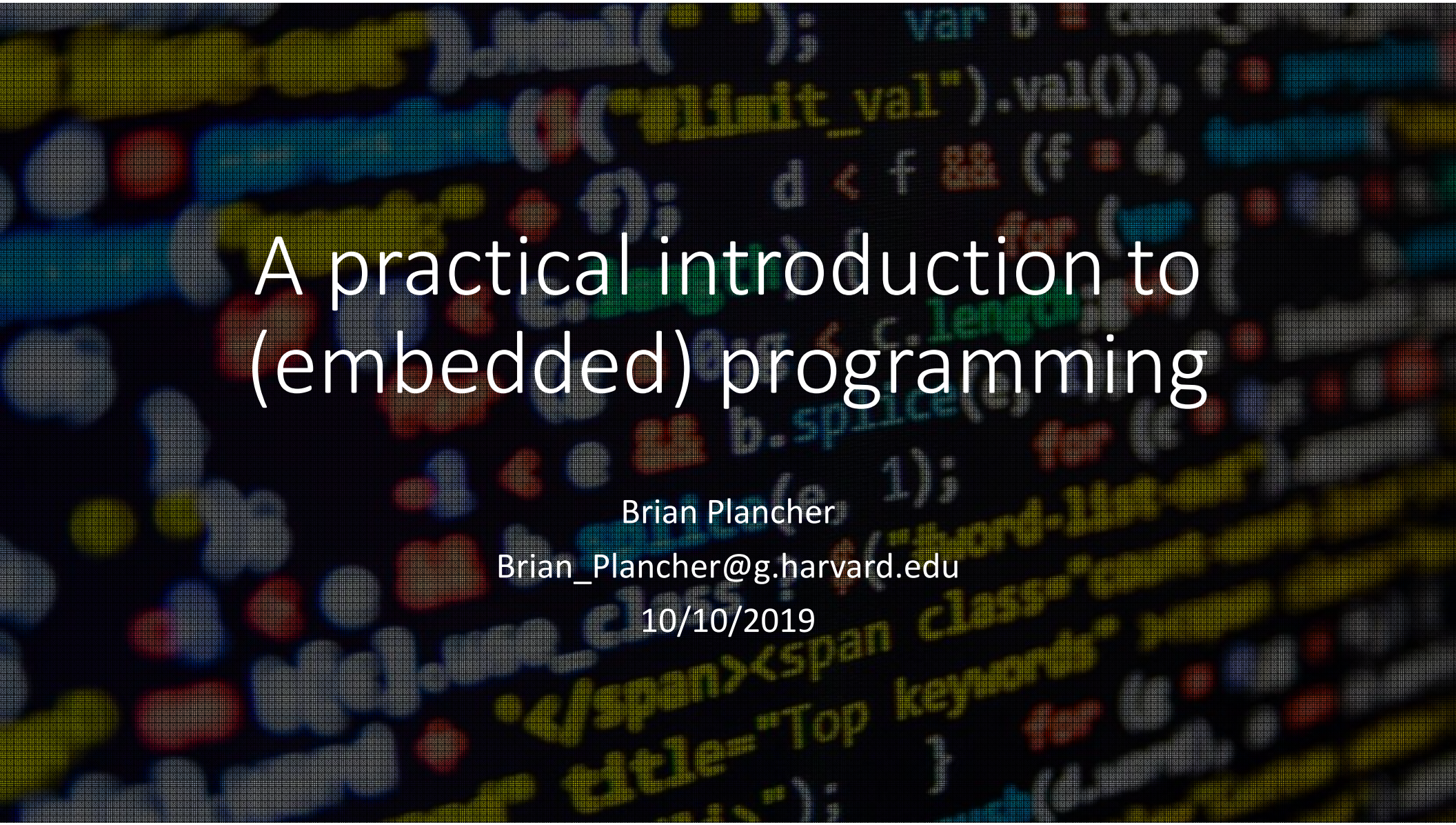


A practical introduction to (embedded) programming

Brian Plancher

Brian_Plancher@g.harvard.edu

10/10/2019

Next week's task is simple:

1. Since the boards you made last week are perfect and are still in perfect shape and are totally programmable…

Next week's task is simple:

1. Since the boards you made last week are perfect and are still in perfect shape and are totally programmable…
2. And since you already know how to code in C…

Next week's task is simple:

1. Since the boards you made last week 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)?

Next week's task is simple:

1. Since the boards you made last week 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)?

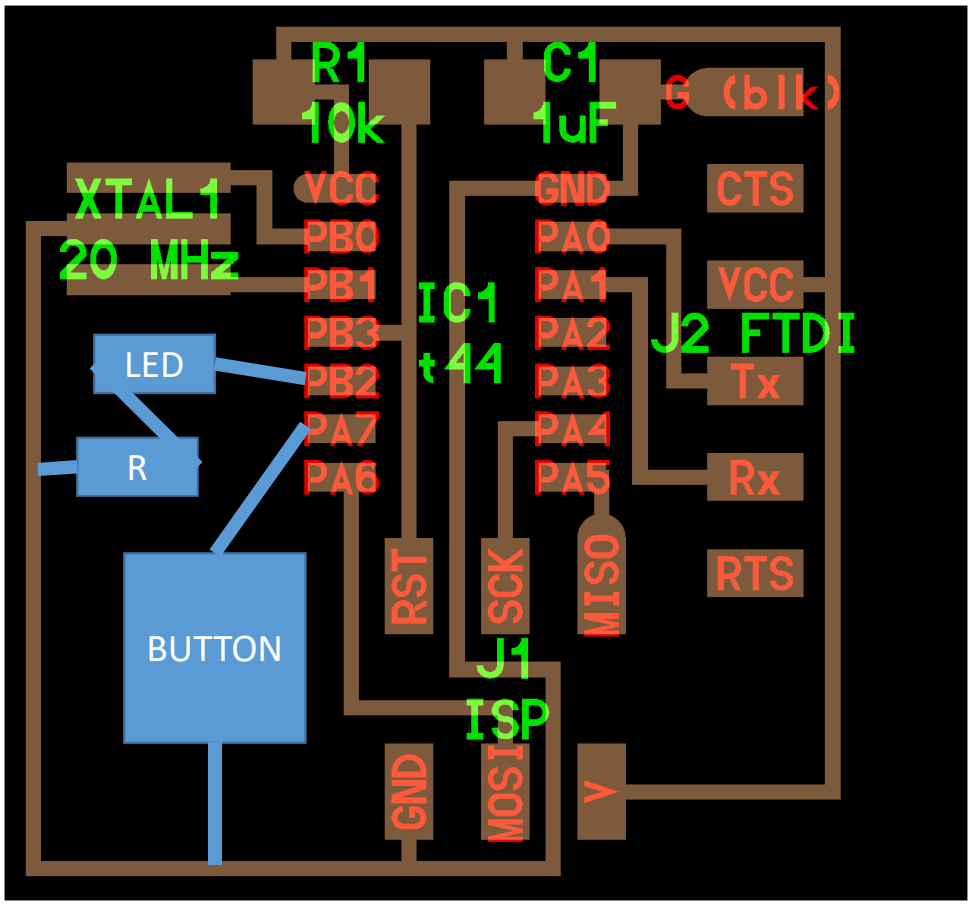
So if you are feeling like…


I HAVE NO IDEA
WHAT I'M DOING


RELAX
WE GOT THIS

One quick aside on boards before we talk about coding…

If you are going 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”

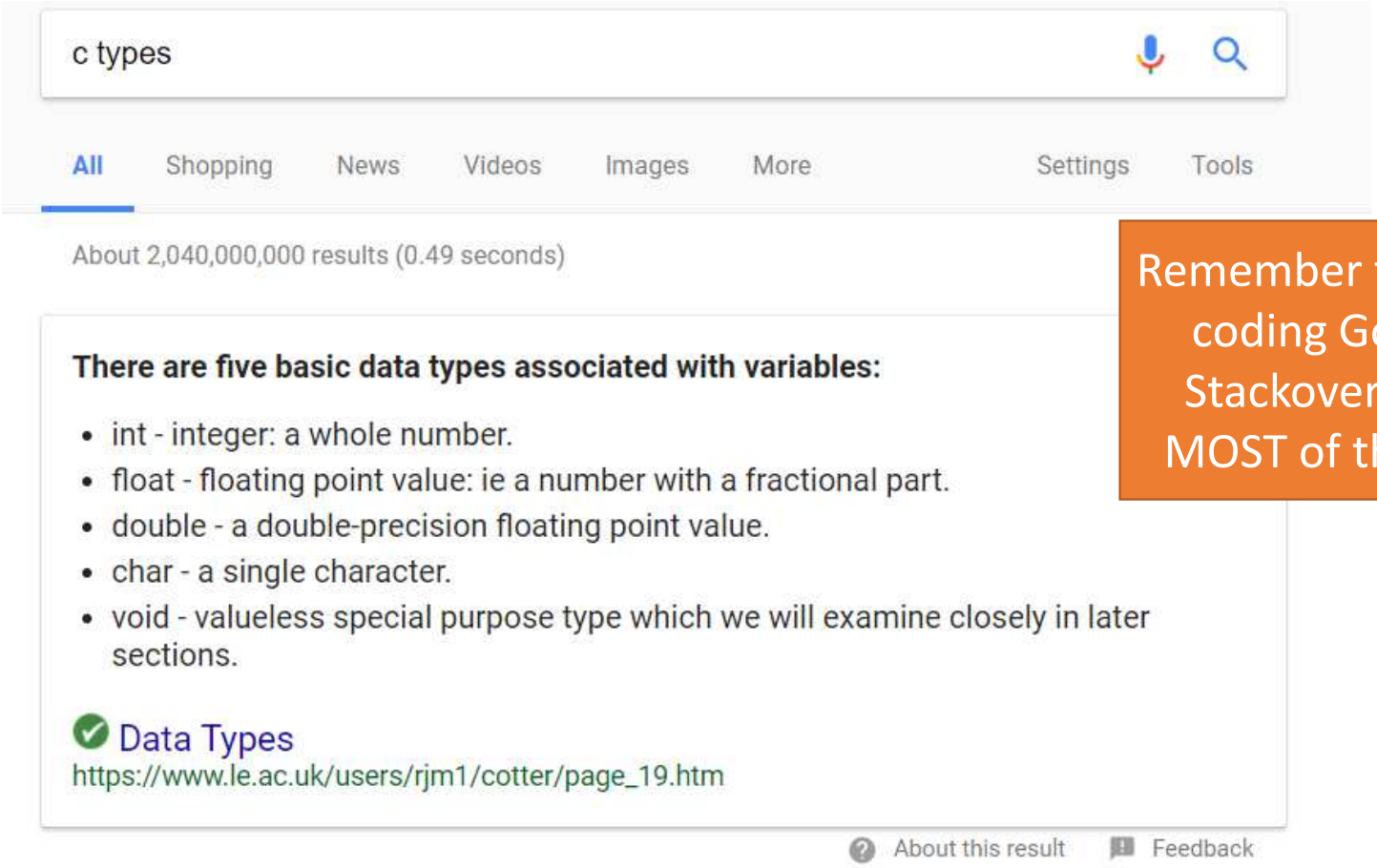
In short, computer code is a human-readable language which tells the computer what to do

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



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 = 28;
char first_initial = 'B';
char last_initial = 'P';
```

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

```
int my_age = 28;
char first_initial = 'B';
char last_initial = 'P';
```

Almost everything ends in semicolons in C!

Don’t forget them!

And everything needs a type!

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 = 28;
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 = 28;
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 operation which you use a lot

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

```
int my_age = 28;
char first_initial = 'B';
char last_initial = 'P';
int above_drinking_age = checkID(my_age);
```

You can create functions to encapsulate some operation 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 operation 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];
```

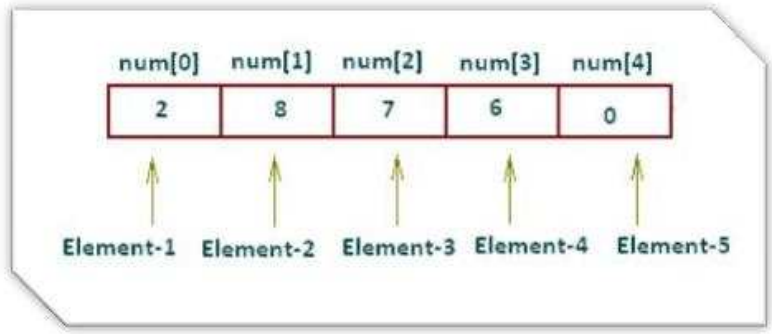
This is an **ARRAY** which is a list of some type. In this case it is 3 ints.

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

Wait so what did we learn?

```
int my age = 28;
```

1. We use variables to store information
2. Each variable has a type (int, char, float, double)
3. We can create arrays of variables to group multiple things of the same type together
4. We use conditional statements (if, else) to branch our code depending on the data
5. We create functions to encapsulate common operations
6. We use loops (while, for) to repetitively call the same set of actions

Wait so what did we learn?

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

1. We use variables to store information
2. Each variable has a type (int, char, float, double)
3. We can create arrays of variables to group multiple things of the same type together
4. We use conditional statements (if, else) to branch our code depending on the data
5. We create functions to encapsulate common operations
6. We use loops (while, for) to repetitively call the same set of actions

Wait so what did we learn?

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

1. We use variables to store information
2. Each variable has a type (int, char, float, dou
3. We can create arrays of variables to group multiple things of the same type together
4. We use conditional statements (if, else) to branch our code depending on the data
5. We create functions to encapsulate common operations
6. We use loops (while, for) to repetitively call the same set of actions

Wait so what did we learn?

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

1. We use variables to store information
2. Each variable has a type (int, char, float, dou
3. We can create arrays of variables to group multiple things of the same type together
4. We use conditional statements (if, else) to branch our code depending on the data
5. We create functions to encapsulate common operations
6. We use loops (while, for) to repetitively call the same set of actions

Wait so what did we learn?

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

1. We use variables to store informatio
2. Each variable has a type (int, char, fl
3. We can create arrays of variables to group multiple things of the same type together
4. We use conditional statements (if, else) to branch our code depending on the data
5. We create functions to encapsulate common operations
6. We use loops (while, for) to repetitively call the same set of actions


NOT BAD

Ok but how does a program know what function to run?

All C programs when run will automagically invoke a special function called **main**

All C programs when run will automagically invoke a special function called **main**

By convention it returns an int as an error code

main.c

```c
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

All C programs when run will automagically invoke a special function called **main**

By convention it returns an int as an error code

main.c

```c
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

main can call all of your other functions (and included external functions)

All C programs when run will automagically invoke a special function called **main**

By convention it returns an int as an error code

main.c

```c
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
```

Here we are calling **printf** which sends text to the console (a really easy way to debug!)

main can call all of your other functions (and included external functions)

All C programs when run will automagically invoke a special function called **main**

By convention it returns an int as an error code

Printf can not only print hard coded strings but also the values of variables

https://alvinalexander.com/programming/printf-format-cheat-sheet

Here we are calling **printf** which sends text to the console (a really easy way to debug!)

ain can call
of your
her
nctions
nd included
xternal
functions)

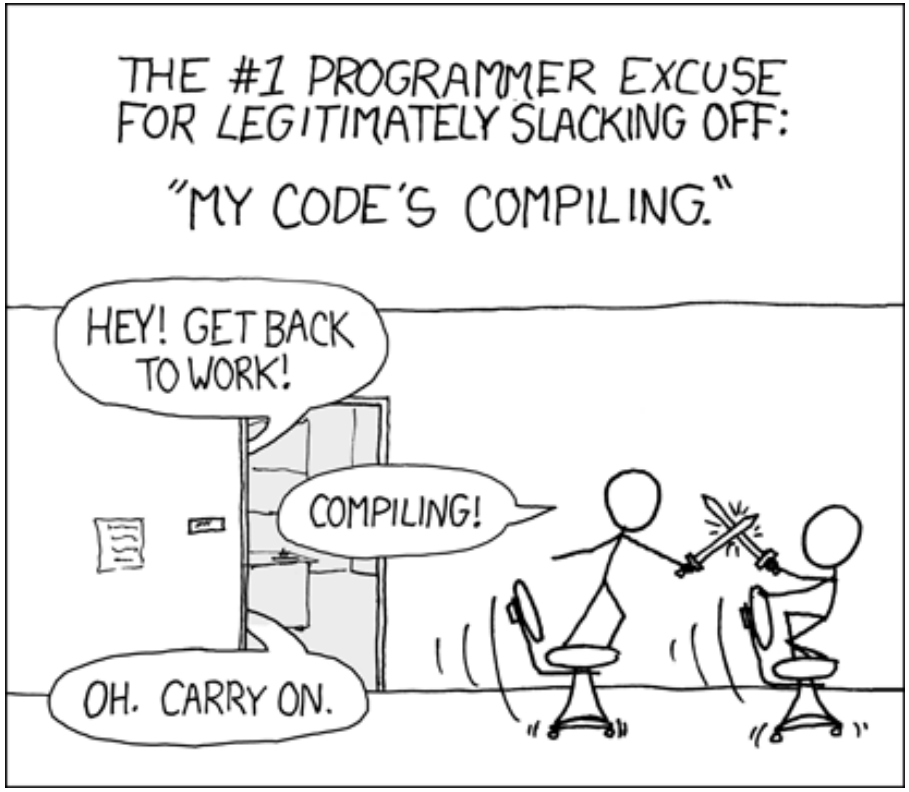
Ok great so I type code in, call it from main, and then the computer just runs it right?

Ok great so I type code in, call it from main, and then the computer just runs it right?

Well not exactly…

We first need to compile the code from words into 0s and 1s

We first need to **compile** the code from words into 0s and 1s



We first need to **compile** the code from words into 0s and 1s

The beauty of this is that compilers are written for you and you can just use them!

We first need to **compile** the code from words into 0s and 1s

The beauty of this is that compilers are written for you and you can just use them!

In this class you've already compiled code with **make**

We

into

ords

Type:

```
make hex
```

If you are successful - you will see this response from the system:

```
akaziuna@Titan:~/Desktop/firmware$ make hex
avr-gcc -Wall -Os -DF_CPU=20000000       -Iusbdrv -I. -DDEBUG_LEVEL=0
-mmcu=attiny44 -c usbdrv/usbdrv.c -o usbdrv/usbdrv.o
avr-gcc -Wall -Os -DF_CPU=20000000       -Iusbdrv -I. -DDEBUG_LEVEL=0
-mmcu=attiny44 -x assembler-with-cpp -c usbdrv/usbdrvasm.S -o usbdrv/usbdrvasm.o
avr-gcc -Wall -Os -DF_CPU=20000000       -Iusbdrv -I. -DDEBUG_LEVEL=0
-mmcu=attiny44 -c usbdrv/oddebug.c -o usbdrv/oddebug.o
avr-gcc -Wall -Os -DF_CPU=20000000       -Iusbdrv -I. -DDEBUG_LEVEL=0
-mmcu=attiny44 -c main.c -o main.o
avr-gcc -Wall -Os -DF_CPU=20000000       -Iusbdrv -I. -DDEBUG_LEVEL=0
-mmcu=attiny44 -o main.elf usbdrv/usbdrv.o usbdrv/usbdrvasm.o usbdrv/oddebug.o
main.o
rm -f main.hex main.eep.hex
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
avr-size main.hex
   text    data     bss     dec     hex filename
      0    2020       0    2020     7e4 main.hex
```

Next, you need to set the fuses so your board will use the external clock (crystal)

Type:

```
make fuse
```

If you are successful - you will see the following response from the system:

```
akaziuna@Titan:~/Desktop/firmware$ sudo make fuse
avrdude -c usbtiny -p attiny44  -U hfuse:w:0xDF:m -U lfuse:w:0xFF:m

avrdude: AVR device initialized and ready to accept instructions
```

We first need to **compile** the code from words into 0s and 1s

One thing to keep in mind is code is compiled **TOP DOWN** – so any helper functions, variables, etc. need to be written **ABOVE** wherever they are used! This is why we need to #include all external code first!

We first need to compile the code from words into 0s and 1s

One th e is
compi lper
functi be
writte e used!
This is ll
extern

main.c

```c
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

Lets work together on a programming example!

1. In one tab open:
   https://www.onlinegdb.com/online_c_compiler
   which is an online console and compiler!

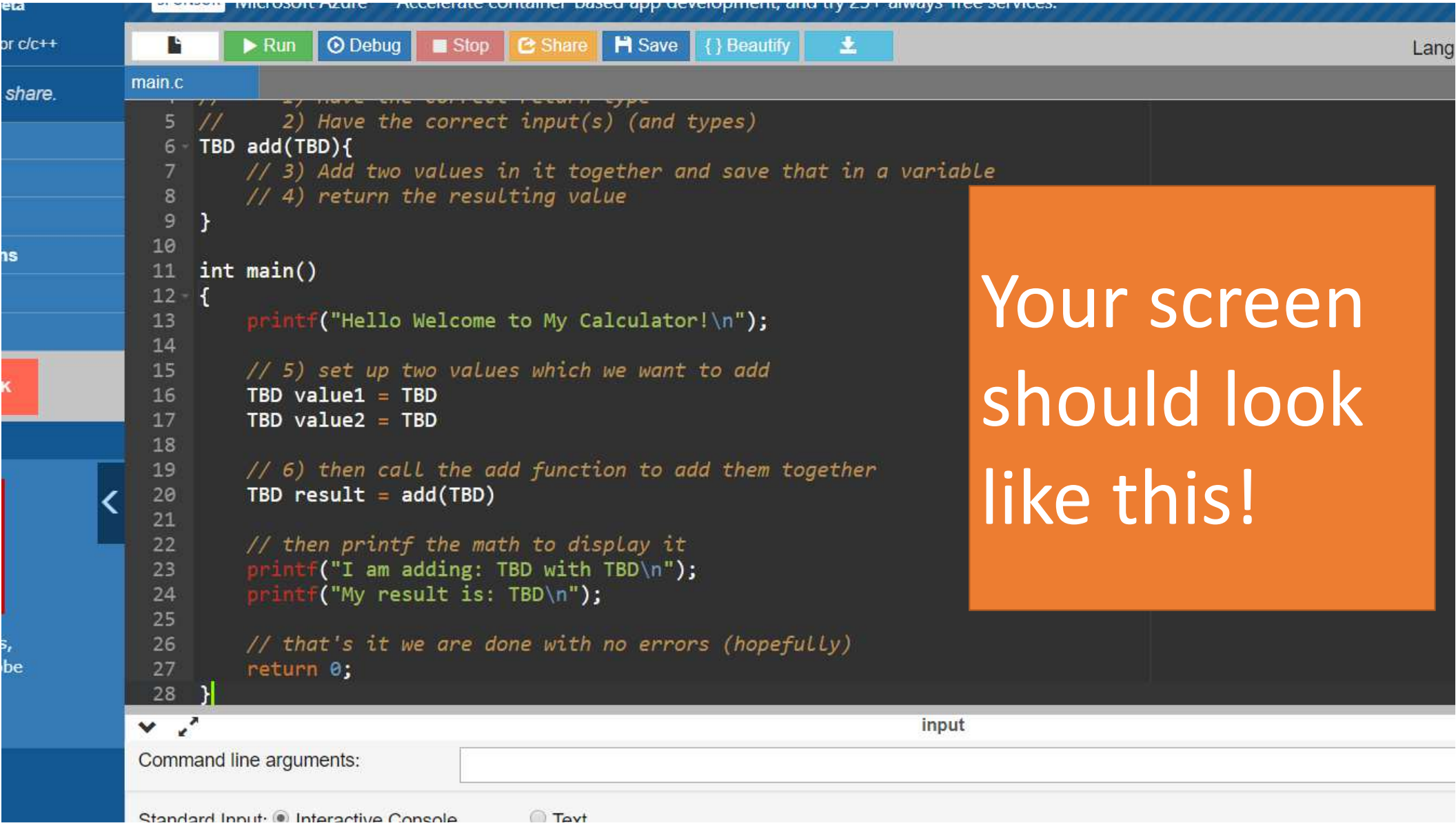

OnlineGDB beta
online compiler and debugger for c/c++
code. compile. run. debug. share.
IDE
My Projects
Learn Programming
Programming Questions
Sign Up
Login
28.5K
Students and Teachers, save up to 60% on Adobe Creative Cloud.
ADS VIA CARBON
FAQ • Blog • Terms of Use • Contact Us • GDB
SPONSOR Microsoft Azur
Run
C
main.c
Click run to compile and run it!
/***
Online C Compiler.
Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.
***/
#include <stdio.h>
int main()
{
printf("Hello World");
return 0;
}
Code can be typed in here!
input
Hello World
...Program finished with exit code 0
Press ENTER to exit console.
Output is here

Lets work together on a programming example!

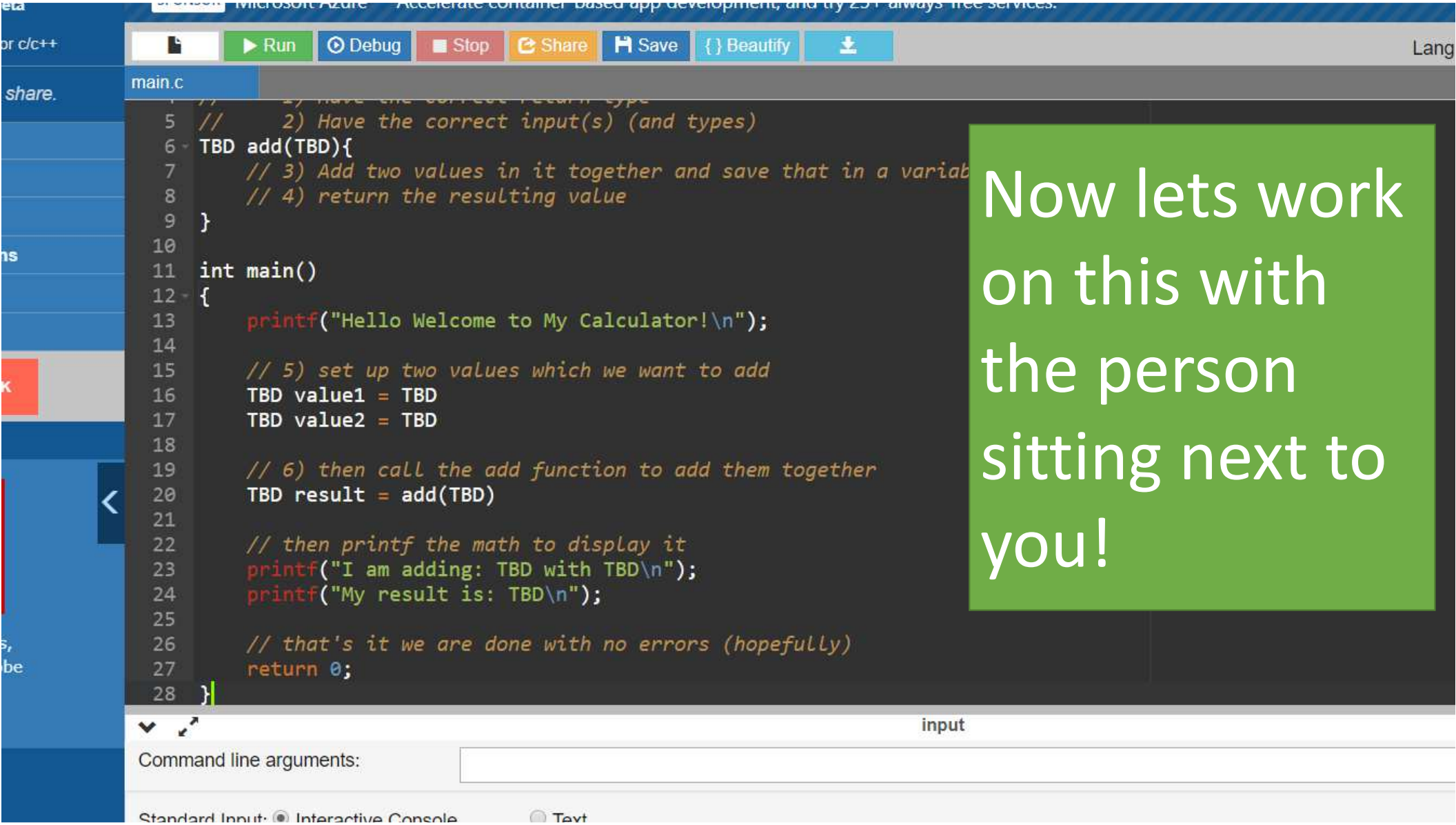
1. In one tab open:
   https://www.onlinegdb.com/online_c_compiler
   which is an online console and compiler!
2. In the other open:
   http://bit.ly/HTM_sample_code
   which some starter code I wrote.

Lets work together on a programming example!

1. In one tab open:
   https://www.onlinegdb.com/online_c_compiler
   which is an online console and compiler!
2. In the other open:
   http://bit.ly/HTM_sample_code
   which some starter code I wrote.
3. Copy and paste the starter code into the online compiler!



Your screen should look like this!


Run
Debug
Stop
Share
Save
{} Beautify
Lang
main.c
// 2) Have the correct input(s) (and types)
TBD add(TBD){
// 3) Add two values in it together and save that in a variab
// 4) return the resulting value
}
int main()
{
printf("Hello Welcome to My Calculator!\n");
// 5) set up two values which we want to add
TBD value1 = TBD
TBD value2 = TBD
// 6) then call the add function to add them together
TBD result = add(TBD)
// then printf the math to display it
printf("I am adding: TBD with TBD\n");
printf("My result is: TBD\n");
// that's it we are done with no errors (hopefully)
return 0;
}
input
Command line arguments:
Standard Input: Interactive Console
Text
Now lets work on this with the person sitting next to you!

Lets work together on a programming example!

One example solution can be found at:

http://bit.ly/HTM_sample_code_sol

Ok so now that we have a little comfort with C lets explore AVR-C by building up / walk through Neil's hello.ftdi.44.echo.c to explore AVR C code

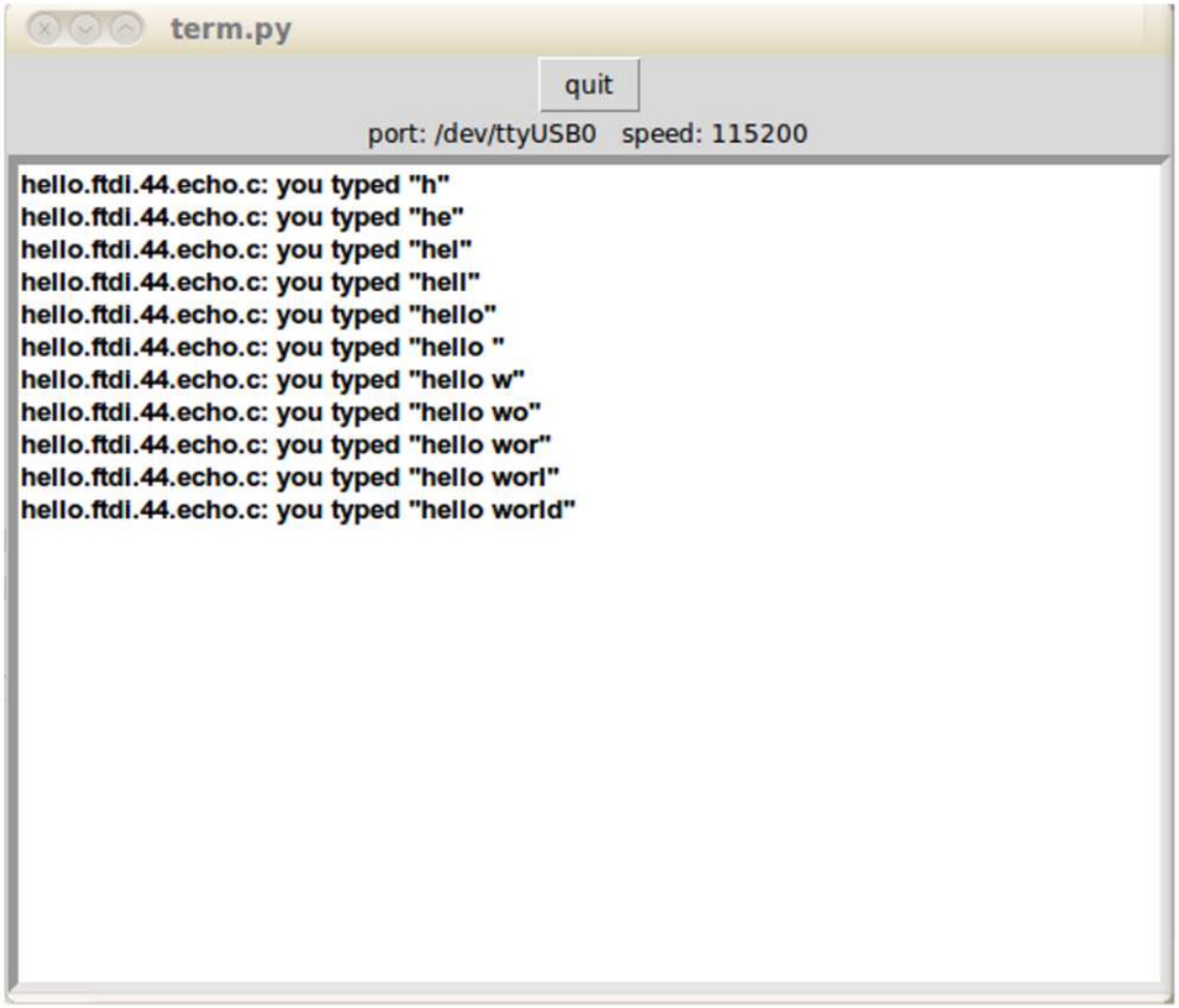
1) what is the program trying to do?

1) what is the program trying to do?	Listen to whatever you type and then echo it back to you.

1) wh

the p

tryin



ever

hen

you.

1) what is the program trying to do?

Listen to whatever you type and then echo it back to you.

Lets try to code this up in pseudo-code!

1) what is the program trying to do?	Listen to whatever you type and then echo it back to you.

Lets try to code this up in pseudo-code!

```
REPEAT FOREVER {
   Read in the next character the user types
   Save it to the end of an array (lets call it BUFFER)
   Then Display "hello.ftdi.44.echo.c: you typed" + BUFFER
}
```

1) what is the program trying to do?	Listen to whatever you type and then echo it back to you.

Lets try to code this up in pseudo-code!

```
REPEAT FOREVER {
   Read in the next character the user types
   Save it to the end of an array (lets call it BUFFER)
   Then Display "hello.ftdi.44.echo.c: you typed" + BUFFER
}
```

We want our Attiny to repeat forever as a simple loop can occur thousands of times a second!

Lets slowly replace all of these words with the code we need to get it to work on the ATTiny

```
REPEAT FOREVER {
   Read in the next character the user types
   Save it to the end of an array (lets call it BUFFER)
   Then Display "hello.ftdi.44.echo.c: you typed" + BUFFER
}
```

Remember “while” defines a **LOOP** (can also use “for”)

“while” will run until the condition in the “()” is FALSE
so in this case it runs forever as 1 is always TRUE

```
While (1) {
    Read in the next character the user types
    Save it to the end of an array (lets call it BUFFER)
    Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
}
```

Remember “while” defines a LOOP (can also use “for”)

“while” will run until the condition in the “()” is FALSE
so in this case it runs forever as 1 is always TRUE

```
while (1) {
    Read in the next character the user types
    Save it to the end of an array (lets call it BUFFER)
    Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
}
```

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

It turns out that a C program always starts by running a special function called “main”

Remember a function is an encapsulated block of code

```
int main (void) {
    While (1) {
        Read in the next character the user types
        Save it to the end of an array (lets call it BUFFER)
        Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
    }
}
```

So we need to wrap our while loop in a “main” function if we want it to actually run forever!

It turns out that a C program always starts by running a special function called “main”

Remember a function is an encapsulated block of code

```
int main (void) {
    While (1) {
        Read in the next character the user types
        Save it to the end of an array (lets call it BUFFER)
        Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
    }
    return 0;
}
```

“main” always returns an integer (that’s just a C standard) and for our purposes it doesn’t take any inputs and thus the use of “void”

In theory we therefore need the “return 0” but since we never exit the while loop main will never return so Neil omits it for brevity

```
//
//
// 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 dir
#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
*/

If we look at Neil's final code we will see that he starts with a big long comment – because comments are helpful! Trust me you want to comment A LOT. It makes it much easier to debug. You will be happy later! I promise!

Note: comments are for humans they are invisible to the computer!

So lets add some comments to our code!

```
// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        Read in the next character the user types
        Save it to the end of an array (lets call it BUFFER)
        Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
    }
}
```

Ok so then now how do we actually start to replace the words with code?

```
// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        Read in the next character the user types
        Save it to the end of an array (lets call it BUFFER)
        Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
    }
}
```

Lets use some HELPER FUNCTIONS (that do the work for us)

```
RETURN_TBD get_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_string(INPUTS_TBD){CODE_TBD;}
```

Neil defines these 3 for this program and their names say what they do (note: this is good coding practice!)

```
// the function that actually gets run
int main (void) {
   // repeat forever
   While (1) {
      Read in the next character the user types
      Save it to the end of an array (lets call it BUFFER)
      Then Display "hello.ftdi.44.echo.c: you typed" + BUFFER
   }
}
```

Since C code gets “compiled” (turned from code to 0s and 1s for the computer to use) top down if we want to define any “helper functions” they need to appear before the main (as the main will call them to use them)

```
RETURN_TBD get_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_string(INPUTS_TBD){CODE_TBD;}
```

Neil defines these 3 for this program and their names say what they do (note: this is good coding practice!)

```
// the function that actually gets run
int main (void) {
   // repeat forever
   While (1) {
      Read in the next character the user types
      Save it to the end of an array (lets call it BUFFER)
      Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
   }
}
```

Since C code gets “compiled” (turned from code to 0s and 1s for the computer to use) top down if we want to define any “helper functions” they need to appear before the main (as the main will call them to use them)

```
RETURN_TBD get_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_string(INPUTS_TBD){CODE_TBD;}
```

Neil defines these 3 for this program and their names say what they do (note: this is good coding practice!)

```
// th
int m
    //
    Wh
        types
s call it BUFFER)
 you typed” + BUFFER
    }
}
```

Side note: put_string is the closest thing to printf for our AVRs as we can see the printed value on the console!

Since C code gets “compiled” (turned from code to 0s and 1s for the computer to use) top down if we want to define any “helper functions” they need to appear before the main (as the main will call them to use them)

```
RETURN_TBD get_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_string(INPUTS_TBD){CODE_TBD;}

// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        Read in the next character the user types
        Save it to the end of an array (lets call it BUFFER)
        Then Display “hello.ftdi.44.echo.c: you typed” + BUFFER
    }
}
```

Neil defines these 3 for this program and their names say what they do (note: this is good coding practice!)

Lets use them to help with these lines!

Since C code gets "compiled" (turned from code to 0s and 1s for the computer to use) top down if we want to define any "helper functions" they need to appear before the main (as the main will call them to use them)

```
RETURN_TBD get_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_string(INPUTS_TBD){CODE_TBD;}

// the function that actually gets run
int main (void) {
    // repeat forever
    while (1) {
        TBD = get_char(TBD);
        Save it to the end of an array (lets call it BUFFER)
        TBD = put_string("hello.ftdi.44.echo.c: you typed", TBD);
        TBD = put_string(BUFFER, TBD)
    }
}
```

We want to get the character from the user and then save it (still TBD) and then put the default string and the buffer out to the user

Since C code gets “compiled” (turned from code to 0s and 1s for the computer to use) top down if we want to define any “helper functions” they need to appear before the main (as the main will call them to use them)

```
RETURN_TBD get_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_char(INPUTS_TBD){CODE_TBD;}
RETURN_TBD put_string(INPUTS_TBD){CODE_TBD;}

// the function that actually gets run
int main (void) {
    // repeat forever
    while (1) {
        TBD = get_char(TBD);
        Save it to the end of an array (lets call it BUFFER)
        TBD = put_string(“hello.ftdi.44.echo.c: you typed”, TBD);
        TBD = put_string(BUFFER, TBD)
    }
}
```

We want to get the character from the user and then save it (still TBD) and then put the default string and the buffer out to the user

Ok but what should all the types and input/outputs be?

```
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 pin, char txchar) {
   //
   // 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 char pin, char *str) {
   //
   // print a null-terminated string
   //
   LOTS OF STUFF WENT HERE
   }
```

Here are Neil's functions. He did a ton of work for you so that this just magically if you use the baud rate 115200 (like from last week).

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.

Also don't worry about "static" or "volatile" or "unsigned" for now – they are complex type things we can get into at another date

```
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 pin, char txchar) {
   //
   // 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 char pin, char *str) {
   //
   // print a null-terminated string
   //
   LOTS OF STUFF WENT HERE
   }
```

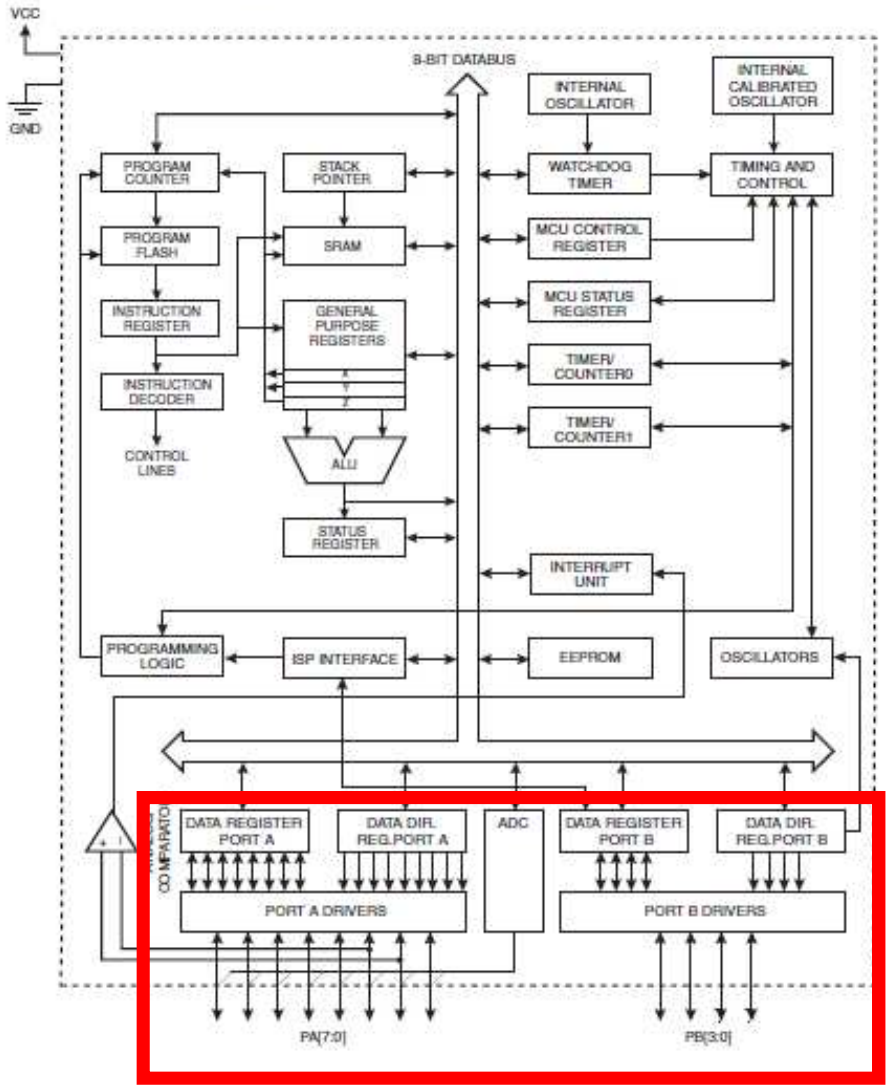
But what are these ports and pins he is talking about?!?

Here are Neil's functions. He did a ton of work for you so that this just magically if you use the baud rate 115200 (like from last week).

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.

Also don't worry about "static" or "volatile" or "unsigned" for now – they are complex type things we can get into at another date

Figure 2-1. Block Diagram

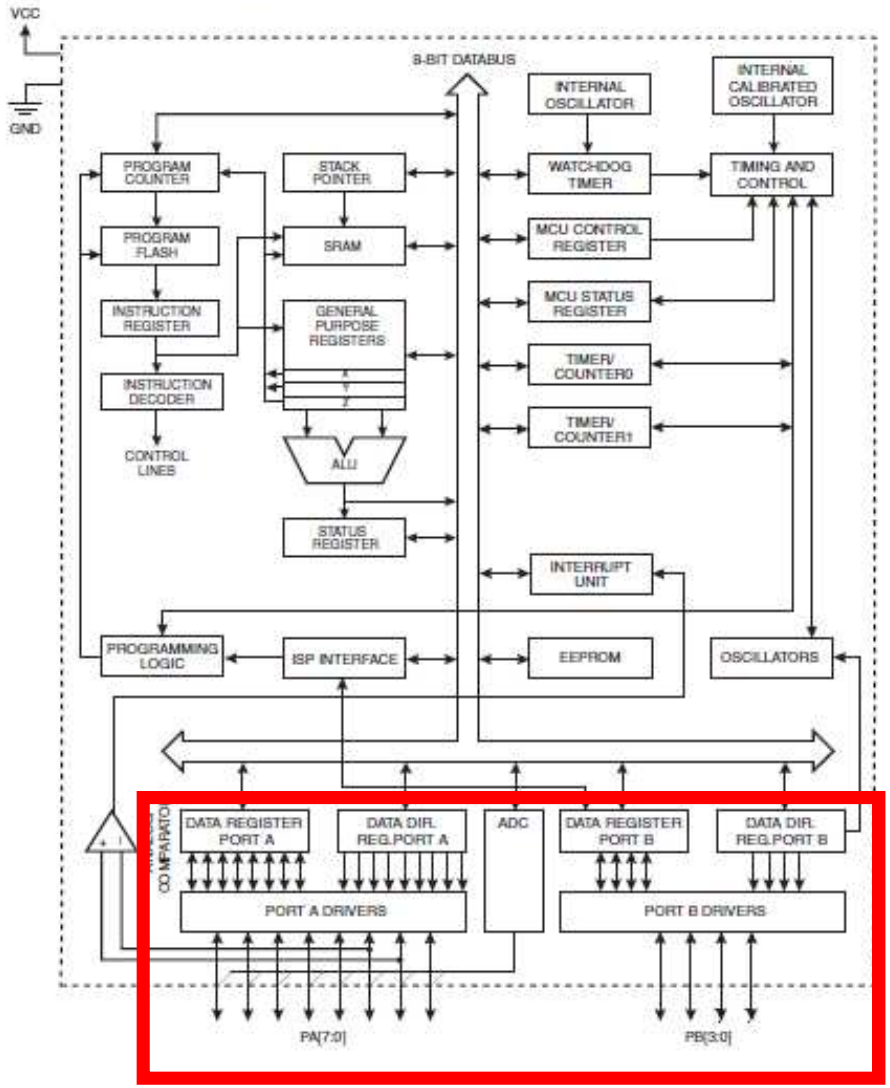


PDIP/SOIC

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

Figure 2-1. Block Diagram



But then do I have to memorize them for every function call that seems tedious!

PDIP/SOIC

VCC	1	14	GND
(PCINT8/XTAL1/CLKI) PB0	2	13	PA0 (ADC0/AREF/PCINT0)
(PCINT9/XTAL2) PB1	3	12	PA1 (ADC1/AIN0/PCINT1)
(PCINT11/$\overline{\text{RESET}}$/dW) PB3	4	11	PA2 (ADC2/AIN1/PCINT2)
(PCINT10/INT0/OC0A/CKOUT) PB2	5	10	PA3 (ADC3/T0/PCINT3)
(PCINT7/ICP/OC0B/ADC7) PA7	6	9	PA4 (ADC4/USCK/SCL/T1/PCINT4)
(PCINT6/OC1A/SDA/MOSI/DI/ADC6) PA6	7	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 serial_port PORTA
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
```

Of course not! Just like Neil, you can just “#define” then and then you can use the descriptive names later!

In this case we have two pins in use on PORTA direction DDRA:

- one for communication in (PA0)
- one for communication out (PA1)

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

Of course not! Just like Neil, you can just “#define” then and then you can use the descriptive names later!

In this case we have two pins in use on PORTA direction DDRA:

- one for communication in (PA0)
- one for communication out (PA1)

Side note 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)

So lets add the ports and pins into the code!

```
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

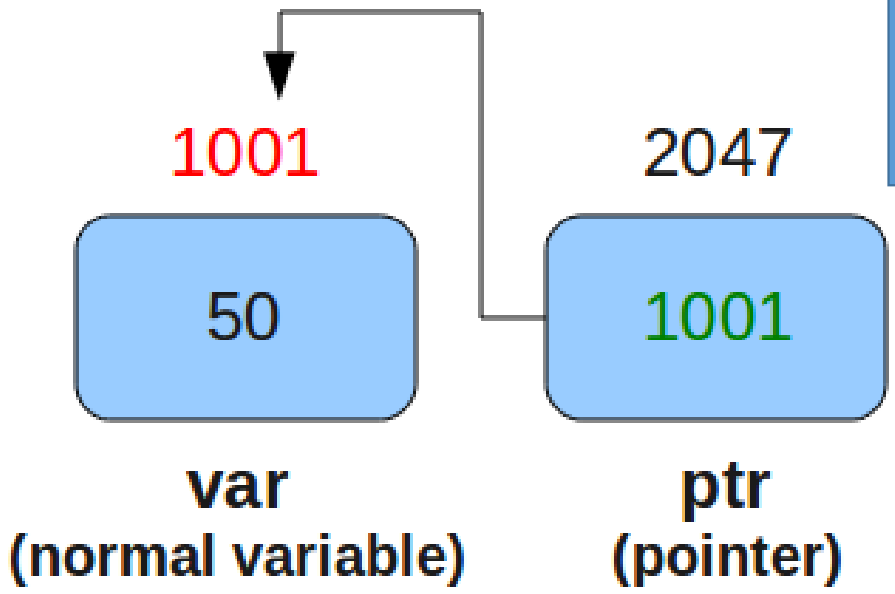
So lets add the ports and pins into the code!

But wait what are all of the "&"s doing?

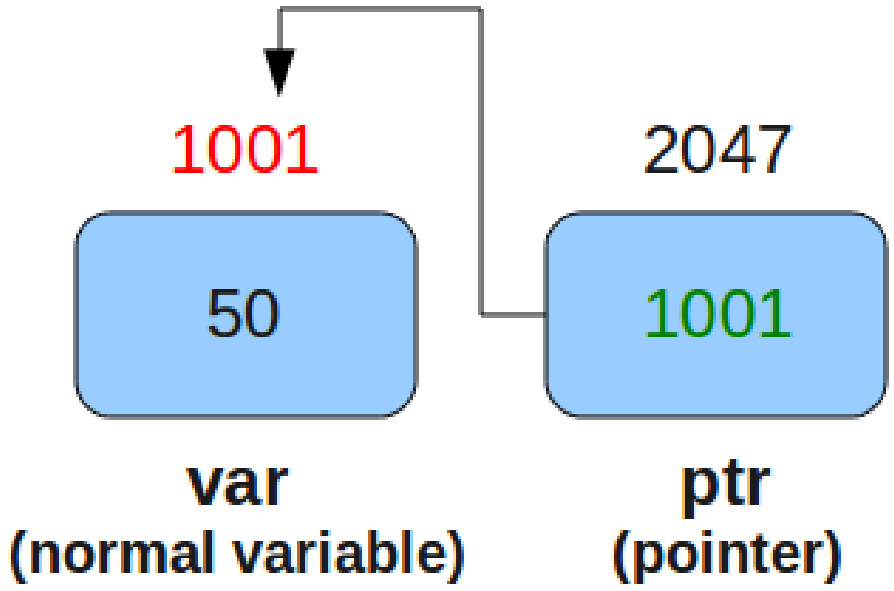
```
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

Pointer FUN?!

Pointer FUN?!




```
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
```

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!

Just remember *s and &s are for **referencing things indirectly**

```
// 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) {
```

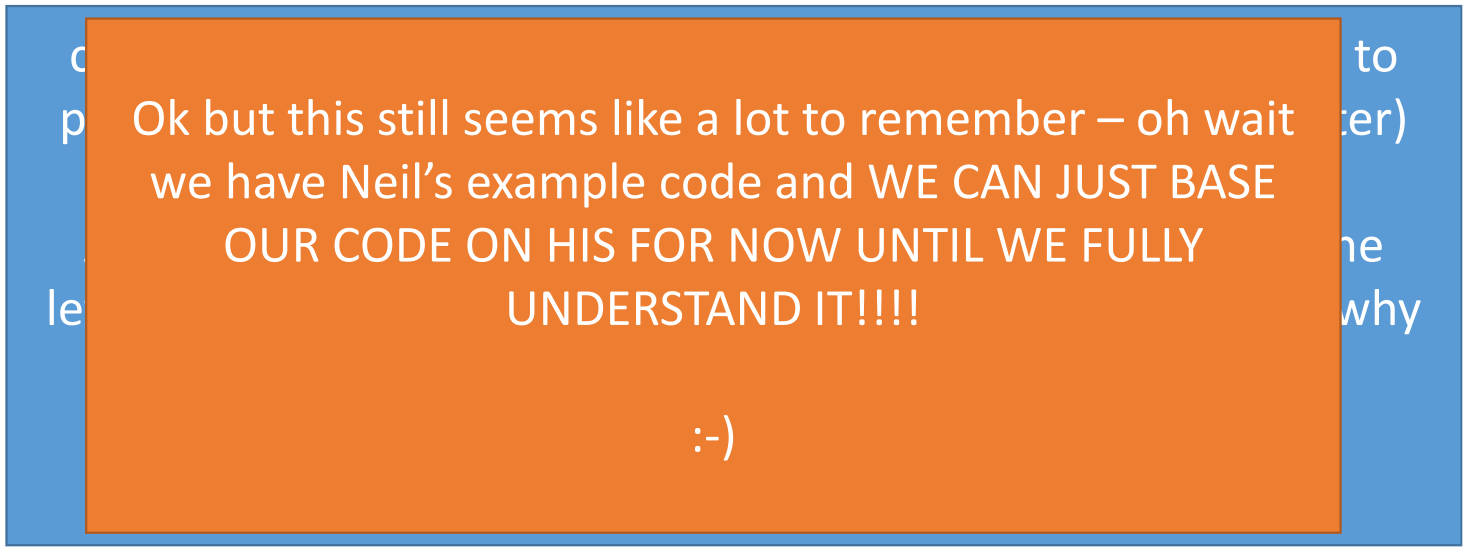
char *pins means pointer to a char (as a type). So we need to pass it the address of the pins (turning the value into a pointer)

And then we'll need to pass it a pointer to a char to store the letter the user types into. This is called a "side effect" and is why the function is "void" (returns nothing)

So we'll just do:

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

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




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

Ok so the * and & thing isn't that scary and the function definitions tell us what to do and we can use Neil's examples for now!

```
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, ADDRESS_OF_CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

Ok so the * and & thing isn't that scary and the function definitions tell us what to do and we can use Neil's examples for now!

Side note: turns out a string is a character array and an array is just a pointer to the start of the array

But again copy Neil's examples!

```
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
void get_char(volatile unsigned char *pins, unsigned char pin,
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, ADDRESS_OF_CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

Ok fine but one other thing – how does the computer know what “PA0” and “PA1” mean?

```
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

// the function that actually gets run
int main (void) {
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, ADDRESS_OF_CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, “hello.ftdi.44.echo.c: you typed”);
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

Ok fine but one other thing – how does the computer know what “PA0” and “PA1” mean?

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
void get_char(volatile unsigned char *pins, unsigned char pin
void put_char(volatile unsigned char *port, unsigned char pin
void put_string(volatile unsigned char *port, unsigned char p

// the function that actually gets run
int main (void) {
     // repeat forever
     while (1) {
          get_char(&serial_pins, serial_pin_in, ADDRESS_OF_CHAR);
          Save it to the end of an array (lets call it BUFFER)
          put_string(&serial_port, serial_pin_out, “hello.ftdi.44.echo.c: you typed”);
          put_string(&serial_port, serial_pin_out, BUFFER)
          }
}
```

Good point – its doesn’t but if we “include” the avr library then we reuse the avr defaults *that someone else wrote* in our code. In this case it happens to define DDRA and PINA and PA0 and PA1!

Side note: Neil uses delay.h in his helper functions which is why that is there too!

Lets hide all of the helper functions and #defines for a minute and finish building out our main function! First by specifying local variables.

We define our local variables outside the loop so that they exist forever. Variables defined inside the loop will get re-created and their values re-set each time the loop happens!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    while (1) {
        get_char(&serial_pins, serial_pin_in, ADDRESS_OF_CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

Lets hide all of the helper functions and #defines for a minute and finish building out our main function! First by specifying local variables.

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    while (1) {
        get_char(&serial_pins, serial_pin_in, ADDRESS_OF_CHAR);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, BUFFER)
    }
}
```

We define our local variables outside the loop so that they exist forever. Variables defined inside the loop will get re-created and their values re-set each time the loop happens!

We initialize both the current buffer size and buffer to 0 (aka nothing is there)

Lets hide all of the helper functions and #defines for a minute and finish building out our main function! First by specifying local variables.

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    while (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
    }
}
```

We define our local variables outside the loop so that they exist forever. Variables defined inside the loop will get re-created and their values re-set each time the loop happens!

We initialize both the current buffer size and buffer to 0 (aka nothing is there)

We can then use the local variables in the loop!

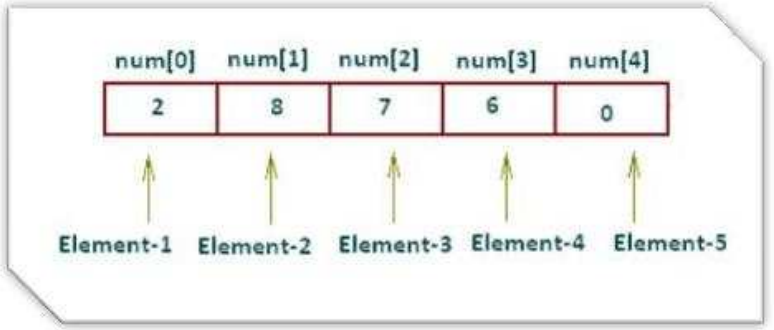
Now lets finish the main part of the loop by adding the char to the buffer!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        Save it to the end of an array (lets call it BUFFER)
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
    }
}
```

Now lets finish the main part of the loop by adding the char to the buffer!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello
        put_string(&serial_port, serial_pin_out, buffer
    }
}
```

Buffer is an ARRAY (list) of char



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

Now lets finish the main part of the loop by adding the char to the buffer!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    while (1) {
        get_char(&serial_pins, serial_pin_in, &cha
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "
        put_string(&serial_port, serial_pin_out, b
    }
}
```

We then have a conditional IF ELSE statement (in this case just an if)

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

For example if BUFFER_SIZE = 4 and we add the alphabet we 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]

Now lets finish the main part of the loop by adding the char to the buffer!

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!

```
    while (1) {
        get_char(&serial_pins, serial_pin_in, &cha
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "
        put_string(&serial_port, serial_pin_out, b
    }
}
```

We then have a conditional IF ELSE statement (in this case just an if)

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

For example if BUFFER_SIZE = 4 and we add the alphabet we 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]

So now we have a relatively complete main loop but there are a couple of things missing that are in Neil's code so lets take a look at them!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
    }
}
```

So now we have a relatively complete main loop but there are a couple of things missing that are in Neil's code so lets take a look at them!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
     // local variables to use in our loop
     static char chr;
     static char buffer[BUFFER_SIZE] = {0};
     static int size = 0;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1))
               size = 0;
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

Why is new line a 10?!?

(and why does the new line not work on all windows computers?)

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						&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052						&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053						&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054						&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055						&#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	3								Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	3								R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	3								S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	3								T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	3								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

ASCII

Cool I see it's a 10 but whats an A or a 012?

asciichars.com

Its just counting in different basses!

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

Now our main loop is complete but we are still missing two things from our program:
Setting the Clock and Initializing the Pins!

```
#define BUFFER_SIZE 24
// the function that actually gets run
int main (void) {
     // local variables to use in our loop
     static char chr;
     static char buffer[BUFFER_SIZE] = {0};
     static int size = 0;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1))
               size = 0;
          put_string(&serial_port, serial_pin_out, “hello.ftdi.44.echo.c: you typed”);
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

Now our main loop is complete but we are still missing two things from our program:
Setting the Clock and Initializing the Pins!

```
int main (void) {
    // set the clock divider to /1
    CLKPR = (1 << CLKPCE);
    CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
    // initialize the pins
    set(serial_port, serial_pin_out);
    output(serial_direction, serial_pin_out);
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
        put_char(&serial_port, serial_pin_out, 10); // new line
    }
}
```

What in the world is all of this?... Read the datasheet! (and don't change it for now – you don't have to)

Now our main loop is complete but we are still missing two things from our program:
Setting the Clock and Initializing the Pins!

```
int main (void) {
    // set the clock divider to /1
    CLKPR = (1 << CLKPCE);
    CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 <
    // initialize the pins
    set(serial_port, serial_pin_out);
    output(serial_direction, serial_pin_out);
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
        put_char(&serial_port, serial_pin_out, 10); // new line
    }
}
```

This is important – we need to tell the AVR which ports and pins will be used for output and on what direction

Now our main loop is complete but we are still missing two things from our program:
Setting the Clock and Initializing the Pins!

```
int main (void) {
    // set the clock divider to /1
    CLKPR = (1 << CLKPCE);
    CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 <
    // initialize the pins
    set(serial_port, serial_pin_out);
    output(serial_direction, serial_pin_out);
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
        put_char(&serial_port, serial_pin_out, 10); // new line
    }
}
```

This is important – we need to tell the AVR which ports and pins will be used for output and on what direction

But where do set and output come from?

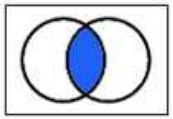
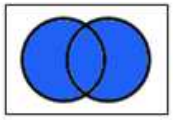
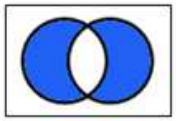
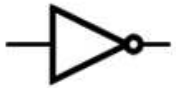
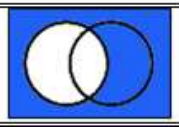
```
//
//
// hello.ftdi.44.echo.c
//
// 115200 baud FTDI character echo, with flash string
//
// set lfuse to 0x5E for 20 MHz xtal
//
// Neil Gershenfeld
// 12/8/10
//
// (c) Massachusetts Institute of Technology 2010
// This work may be reproduced, modified, distributed,
// performed, and displayed 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>
```

From more of Neil's handy #defines of course!

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 Boolean logic

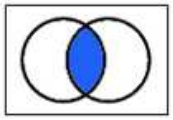
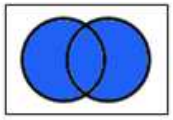
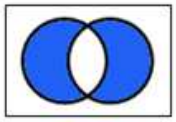
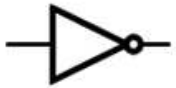
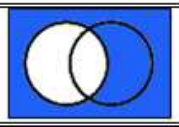
```
#define output(directions,pin) (directions |= pin) // set port direction for output
#define set(port,pin) (port |= pin) // set port pin
#define clear(port,pin) (port &= (~pin)) // clear port pin
#define pin_test(pins,pin) (pins & pin) // test for port pin
#define bit_test(byte,bit) (byte & (1 << 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
```

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

```
#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
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

```
#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

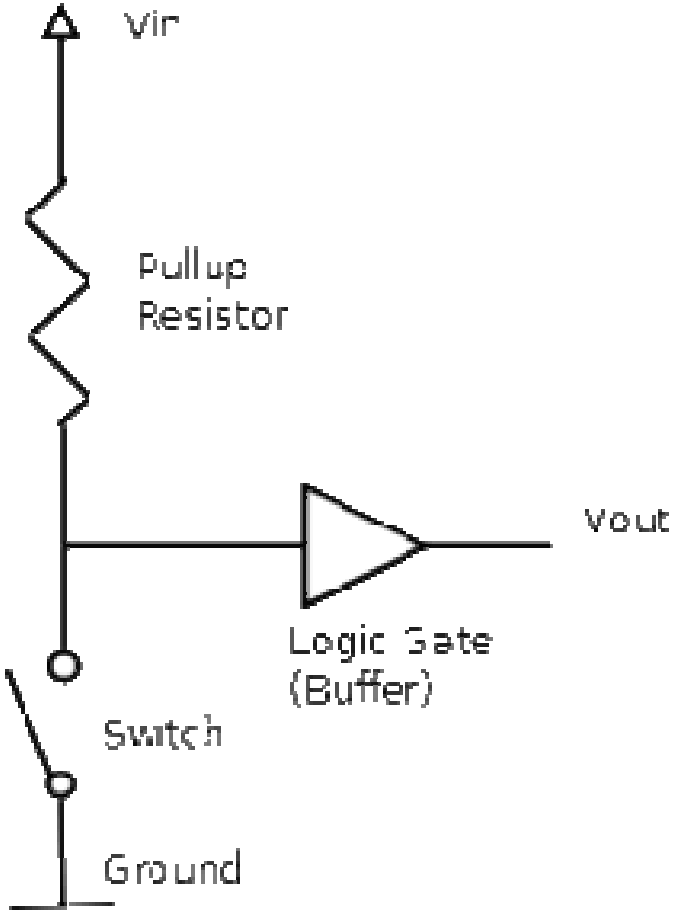
So if we pick a pin with a 1 then OR it we will set it. And if we AND the NOT of it we will AND a 0 and thus unset it!

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

Now our main loop is complete but we are still missing two things from our program:
Setting the Clock and Initializing the Pins!

```
int main (void) {
    // set the clock divider to /1
    CLKPR = (1 << CLKPCE);
    CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0
    // initialize the pins
    set(serial_port, serial_pin_out);
    output(serial_direction, serial_pin_out);
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &char);
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
        put_char(&serial_port, serial_pin_out, 10); // new line
    }
}
```

So **outputs we always set** and label as an output but for **inputs** it is a little more **complicated** depending on if you want the **pull-up resistor** turned on



Remember from electronics production 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)

Now our main loop is complete but we are still missing two things from our program:
Setting the Clock and Initializing the Pins!

```
int main (void) {
    // set the clock divider to /1
    CLKPR = (1 << CLKPCE);
    CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 <
    // initialize the pins
    set(serial_port, serial_pin_out);
    output(serial_direction, serial_pin_out);
    // local variables to use in our loop
    static char chr;
    static char buffer[BUFFER_SIZE] = {0};
    static int size = 0;
    // repeat forever
    While (1) {
        get_char(&serial_pins, serial_pin_in, &
        buffer[size++] = chr;
        if (size == (BUFFER_SIZE-1))
            size = 0;
        put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
        put_string(&serial_port, serial_pin_out, buffer)
        put_char(&serial_port, serial_pin_out, 10); // new line
    }
}
```

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

And since we defined nice names for the ports and pins earlier we can just use them again here!

So are we done?!?

Includes

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);  output(serial_direction, serial_pin_out);
     static int size = 0;               static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

So are we done?!?

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define output(directions,pin) (directions |= pin)
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);  output(serial_direction, serial_pin_out);
     static int size = 0;               static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

Defines

So are we done?!?

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define output(directions,pin) (directions |= pin)
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);   output(serial_direction, serial_pin_out);
     static int size = 0;                static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

Helpers

So are we done?!?

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define output(directions,pin) (directions |= pin)
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);   output(serial_direction, serial_pin_out);
     static int size = 0;                static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

One-time Setup

So are we done?!?

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define output(directions,pin) (directions |= pin)
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);   output(serial_direction, serial_pin_out);
     static int size = 0;                static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

One-time Setup

Note whitespace doesn't matter so I can cram all this code into three lines (but it's hard to read so BAD TO DO GENERALLY)

So are we done?!?

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define output(directions,pin) (directions |= pin)
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);   output(serial_direction, serial_pin_out);
     static int size = 0;                static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

Run Forever

So are we done?!?

Almost! We just need to talk about how we turn the code into 0s and 1s aka “compiling” (remember that from earlier?)

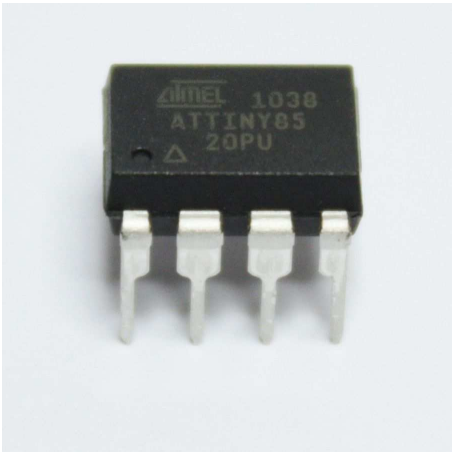
```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#define output(directions,pin) (directions |= pin) // set port dir
#define set(port,pin) (port |= pin) // set port pin
#define serial_direction DDRA
#define serial_pins PINA
#define serial_pin_in (1 << PA0)
#define serial_pin_out (1 << PA1)
#define BUFFER_SIZE 24
void get_char(volatile unsigned char *pins, unsigned char pin, char *rxbyte){CODE_HERE;}
void put_char(volatile unsigned char *port, unsigned char pin, char txchar){CODE_HERE;}
void put_string(volatile unsigned char *port, unsigned char pin, char *str){CODE_HERE;}

int main (void) {
     CLKPR = (1 << CLKPCE);              CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
     set(serial_port, serial_pin_out);  output(serial_direction, serial_pin_out);
     static int size = 0;               static char buffer[BUFFER_SIZE] = {0};          static char chr;
     // repeat forever
     While (1) {
          get_char(&serial_pins, serial_pin_in, &char);
          buffer[size++] = chr;
          if (size == (BUFFER_SIZE-1)){size = 0;}
          put_string(&serial_port, serial_pin_out, “hello.ftdi.44.echo.c: you typed”);
          put_string(&serial_port, serial_pin_out, buffer)
          put_char(&serial_port, serial_pin_out, 10); // new line
     }
}
```

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)!



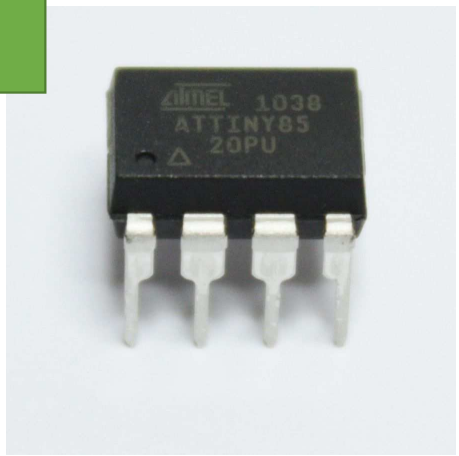
C Code
(.c, .h)

Byte Code
(.o)

Hex Code
(.hex)

Lets 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 and avr 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

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

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

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

$(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
```

Here's the best part – as long as you don't include big external libraries (or simply copy and paste them into your code at the top) you won't have to ever touch the MAKEFILE beyond the type of board and file name! Thanks Neil :-)

We did it! That’s Neil’s code explained line by line!

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?

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.

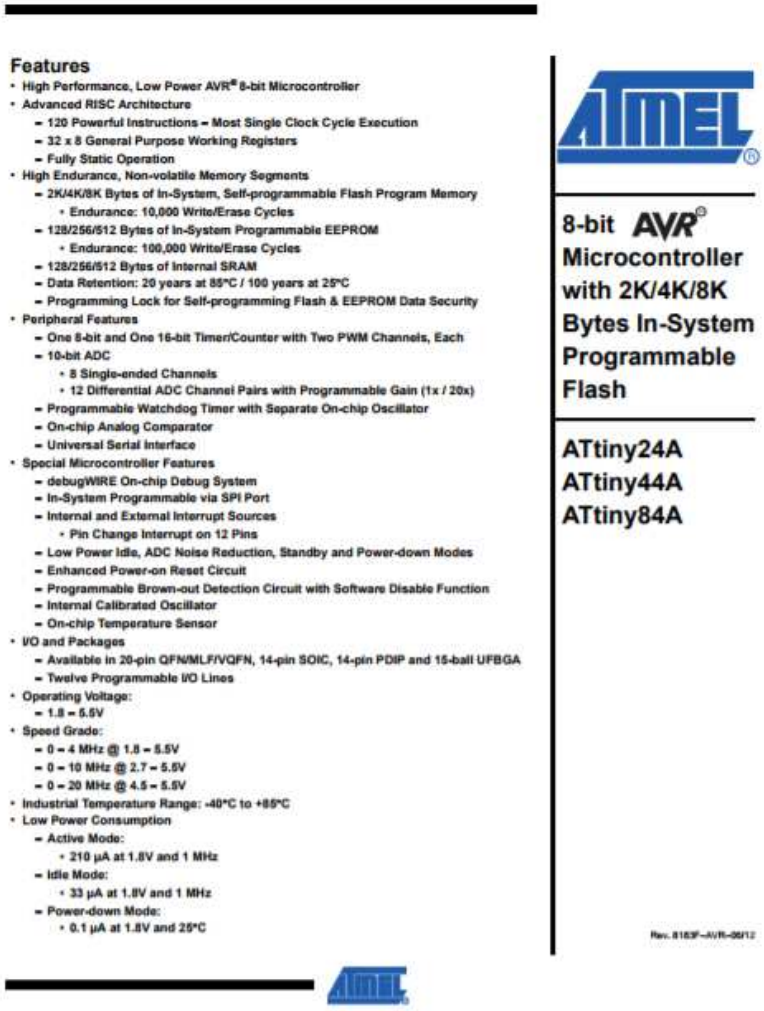
Timers and Clock Registers

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

Interrupts

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

ATMEL

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

ATtiny24A
ATtiny44A
ATtiny84A

Rev. 8183F–AVR–06/12

ATMEL

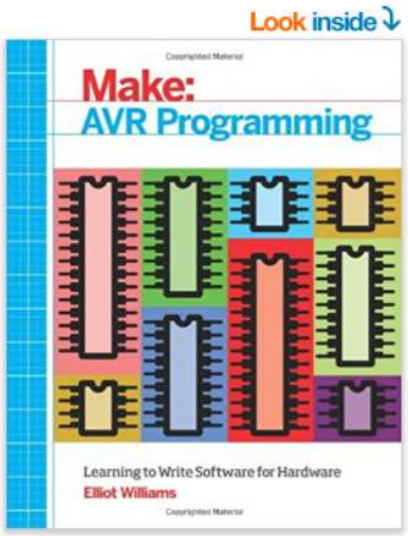
And so so so much more (e.g. ADC) so read up!
:-)

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

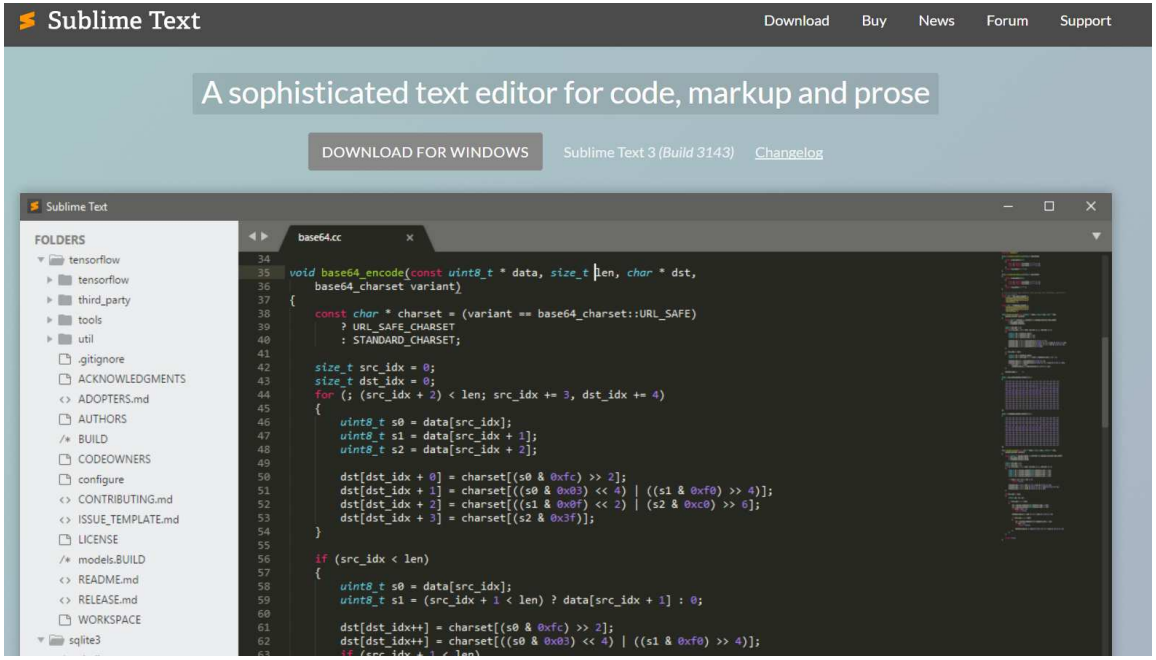
Add to Cart

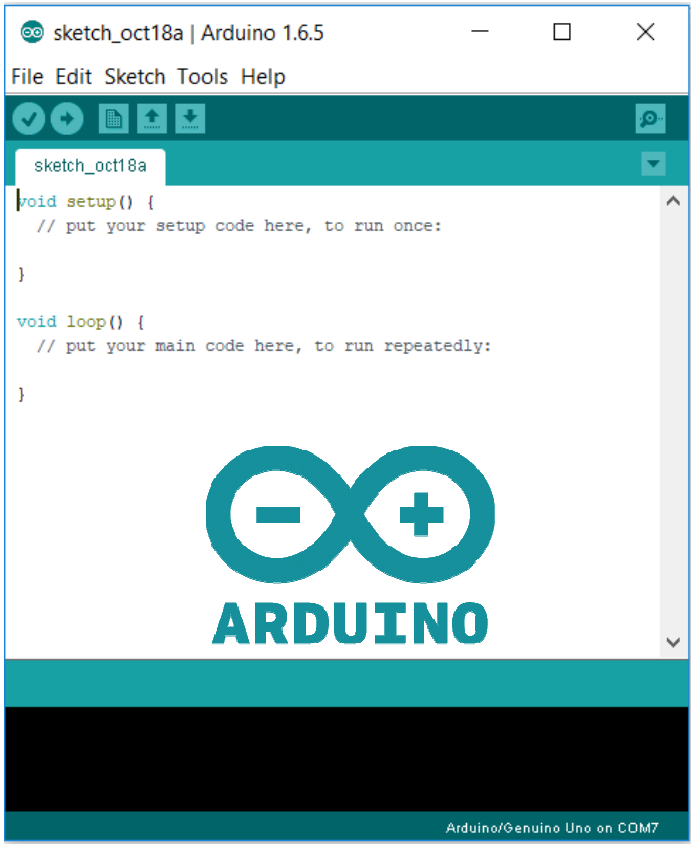
Turn on 1-Click ordering

Ship to:

Brian Plancher- Somerville - 02144

Possible Lightweight Editors to Use (IDE)

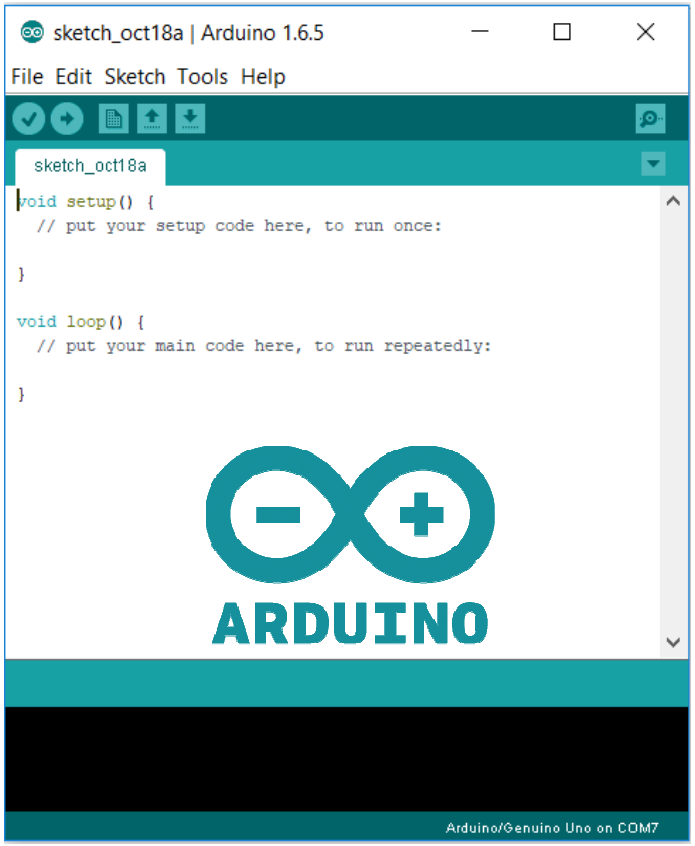




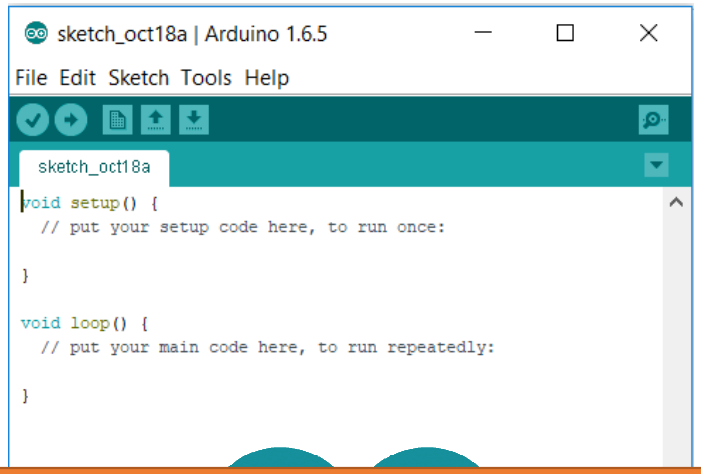
Everything is harder on windows → Linux VM

Possible Lightweight Editors to Use (IDE)

Everything is harder on windows → Linux VM

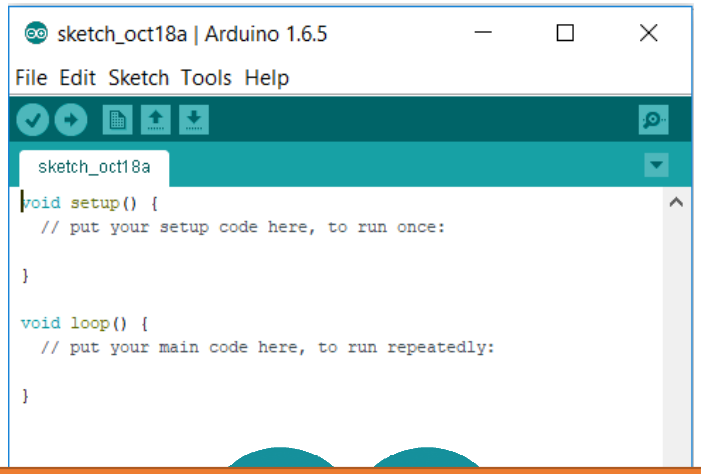


Arduino is a board, form factor, libraries, IDE, bootloader, and headers!



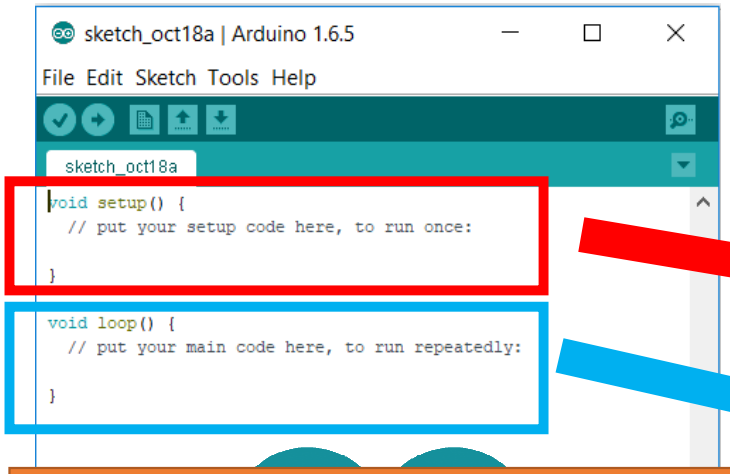
1. It does a ton of #include and #define behind the scenes

Arduino is a board, form factor, libraries, IDE, bootloader, and headers!



1. It does a ton of #include and #define behind the scenes
2. It has a bootloader that auto does the compile and program

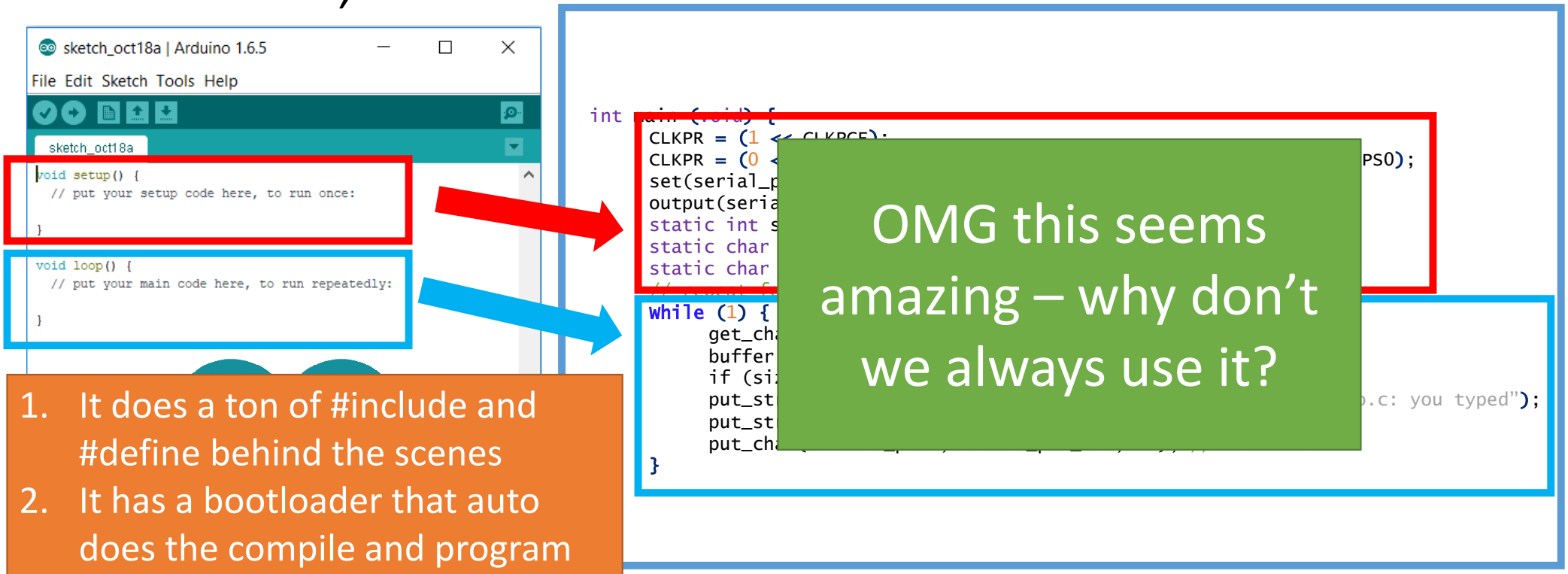
Arduino is a board, form factor, libraries, IDE, bootloader, and headers!




```
int main (void) {
CLKPR = (1 << CLKPCE);
CLKPR = (0 << CLKPS3) | (0 << CLKPS2) | (0 << CLKPS1) | (0 << CLKPS0);
set(serial_port, serial_pin_out);
output(serial_direction, serial_pin_out);
static int size = 0;
static char buffer[BUFFER_SIZE] = {0};
static char chr;
// Print function
While (1) {
      get_char(&serial_pins, serial_pin_in, &char);
      buffer[size++] = chr;
      if (size == (BUFFER_SIZE-1)){size = 0;}
      put_string(&serial_port, serial_pin_out, "hello.ftdi.44.echo.c: you typed");
      put_string(&serial_port, serial_pin_out, buffer)
      put_char(&serial_port, serial_pin_out, 10); // new line
}
```

1. It does a ton of #include and #define behind the scenes
2. It has a bootloader that auto does the compile and program
3. It wraps up the do once and while loop code into nicely named functions

Arduino is a board, form factor, libraries, IDE, bootloader, and headers!



1. It does a ton of #include and #define behind the scenes
2. It has a bootloader that auto does the compile and program
3. It wraps up the do once and while loop code into nicely named functions

Arduino is unfortunately very memory intensive which requires a nicer IC!

Product Overview

Digi-Key Part Number	1611-ATTINY44V-15SSTCT-ND
Quantity Available	**4,849** Can ship immediately
Manufacturer	Microchip Technology
Manufacturer Part Number	ATTINY44V-15SST
Description	IC MCU 8BIT 4KB FLASH 14SOIC
Manufacturer Standard Lead Time	12 Weeks
Detailed Description	AVR AVR® ATtiny Microcontroller IC 8-Bit 8MHz 4KB (2K x 16) FLASH 14-SOIC

Price & Procurement

Quantity 1

1611-ATTINY44V-15SSTCT-ND

Customer Reference

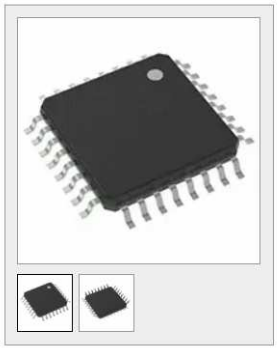
Add to Cart

All prices are in USD.

Price Break	Unit Price	Extended Price
1	0.52000	$0.52
25	0.44000	$11.00
100	0.37700	$37.70

Submit a request for quotation on quantities greater than those displayed.

Documents & Media



Product Overview

Digi-Key Part Number	ATMEGA328PB-AURTR-ND
Quantity Available	**20,000** Can ship immediately
Manufacturer	Microchip Technology
Manufacturer Part Number	ATMEGA328PB-AUR
Description	IC MCU 8BIT 32KB FLASH 32TQFP
Manufacturer Standard Lead Time	7 Weeks
Detailed Description	AVR AVR® ATmega Microcontroller IC 8-Bit 20MHz 32KB (16K x 16) FLASH 32-TQFP (7x7)

Price & Procurement

Quantity 2000

ATMEGA328PB-AURTR-ND

Customer Reference

Add to Cart

All prices are in USD.

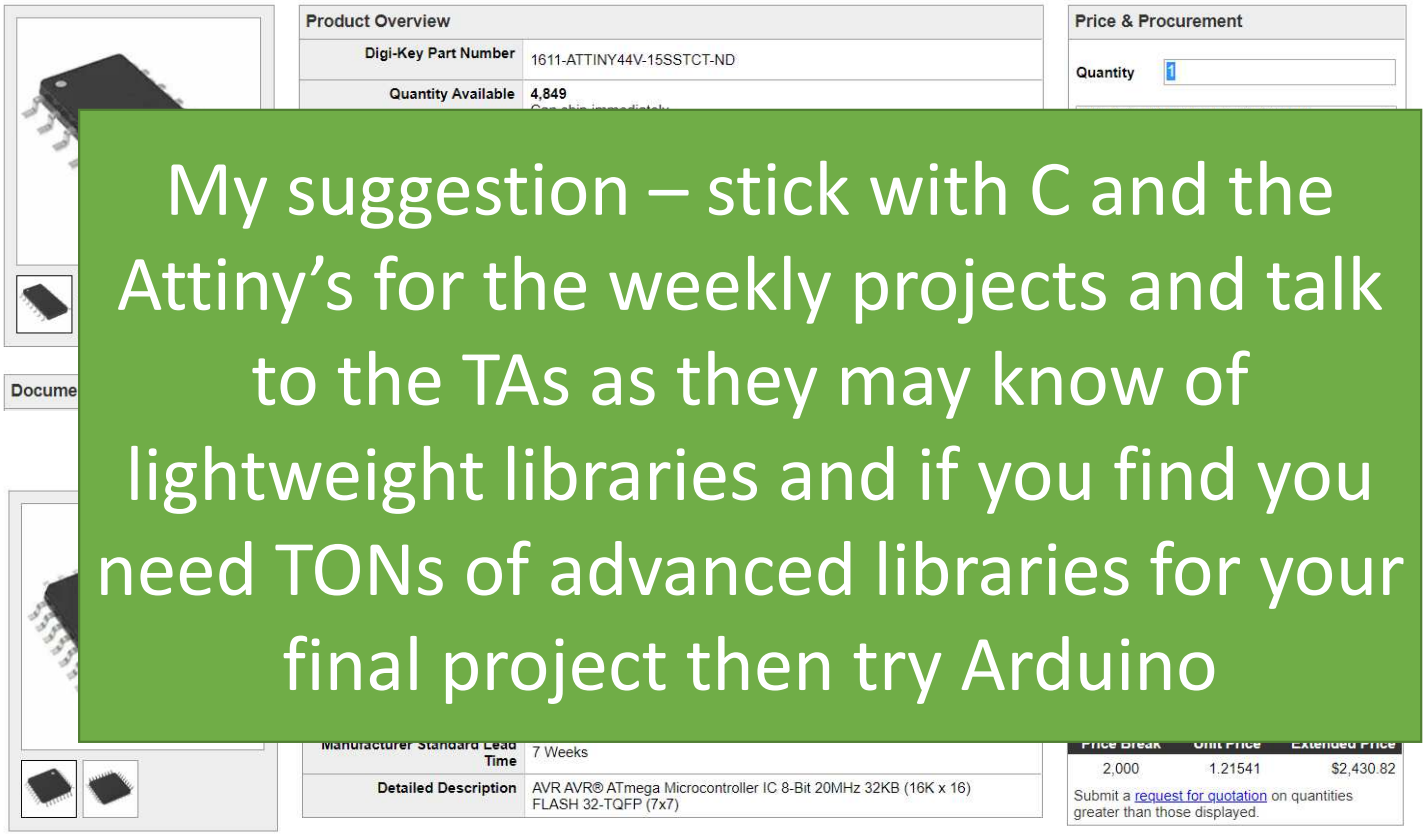
Price Break	Unit Price	Extended Price
2,000	1.21541	$2,430.82

Submit a request for quotation on quantities greater than those displayed.

We can buy ATTinys in bulk for 40 cents while the lowest price I could find on digikey for an ATmega328P (the Arduino chip) was $1.20

Plus you have to solder way more pins and take up way more space on your board!

Arduino is unfortunately very memory intensive which requires a nicer IC!



My suggestion – stick with C and the Attiny's for the weekly projects and talk to the TAs as they may know of lightweight libraries and if you find you need TONs of advanced libraries for your final project then try Arduino

We can buy ATTinys in bulk for 40 cents while the lowest price I could find on digikey for an ATmega328P (the Arduino chip) was $1.20

Plus you have to solder way more pins and take up way more space on your board!

And we’re totally 100% done!

Questions?