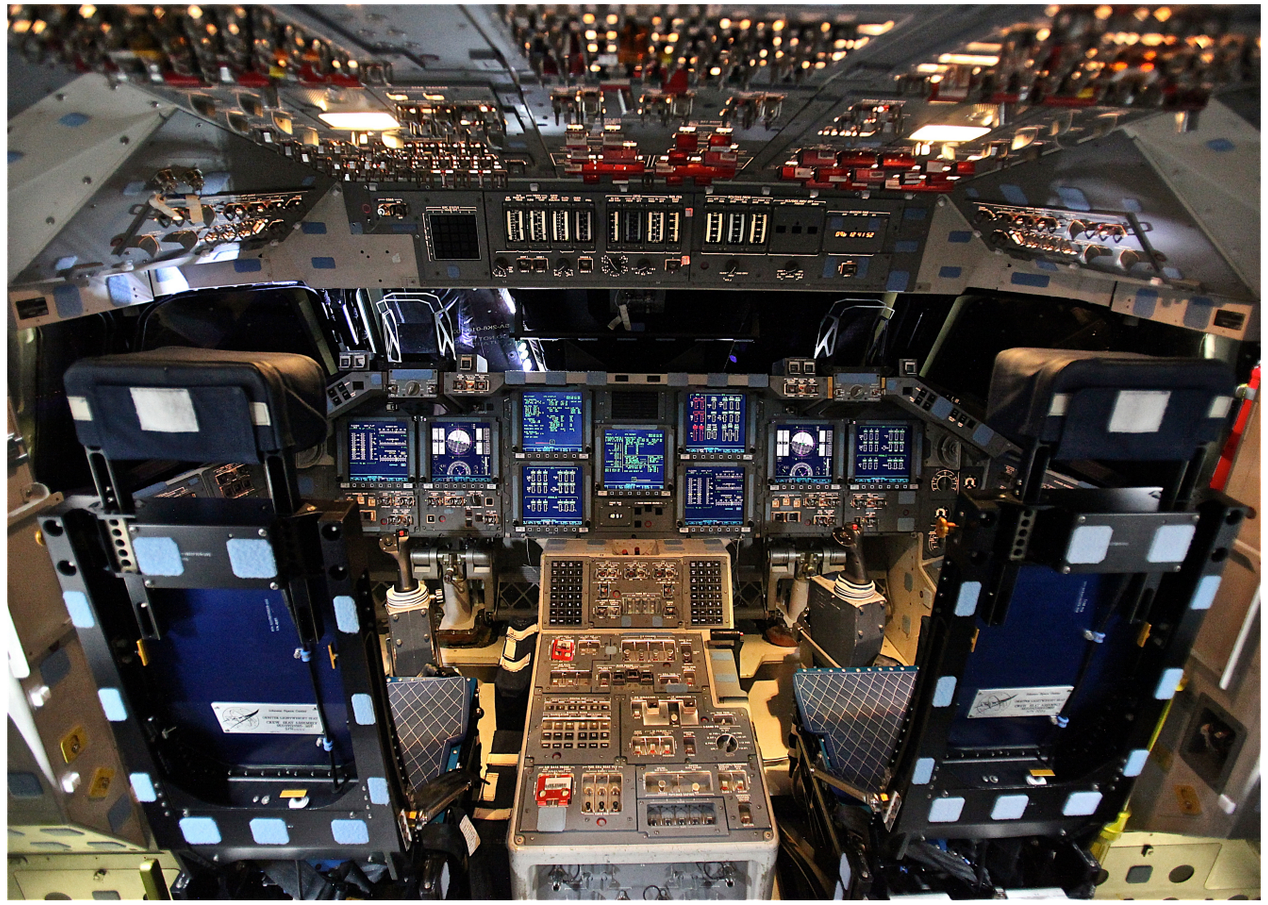
A USB JOYSTICK CONTROLLER

Hempstick User's Guide

v.1.0.3



Jonah Tsai

Copyright

Cover photo is provided by Mr. Steve Jurvetson under Creative Common Attribution 2.0 Generic license (CC By 2.0).

Preface, chapter, and section title photos are attributed to NASA.

Preface

The name Hempstick is simply a play of my screen name, Hempstead. My screen name simply came from the street name my apartment used to be in Pittsburgh.

Hempstick is a general purpose USB HID controller. It came from the desire to produce my own custom USB physical controllers like instrument panels, rudders, joysticks, etc., but I cannot find a suitable USB controller board on the market. For instance, I want one that will read Thrustmaster's Cougar/ Warthog sticks. Also, I want it to be able to read Melexis' MLX90363 Hall Effect Sensors in digital SPI mode and more, but again, I cannot find one either. So, ever as a born tinkerer, I write my own.

Some might say, there sure would be some available software out there that already does USB joysticks/ controllers! Why should I bother with Hempstick at all? For instance, Arduino has one! Yes, it does. But the existing ones I found are way too simplistic. They use a simple loop to do all the work, i.e. read the ADC, buttons status, report them to the USB host, then sleeps for a specified amount of millisecond and do the loop again, forever. What's wrong with that?

Nothing, if that's all it does. What happens if you want the report rate to be 1000Hz, and also need to read 15 SPI Hall Effect Sensors and each report can take up to 5 ms, then read all the buttons, and 8 ADC channels? You can no long meet the time constraint of 1ms (1000Hz) report rate. What if I also want to read gyro, accelerometer, and magnetometer for head tracking in the future? Can you make the MCU do all the above within 1ms? Even if your MCU is fast enough to do all those, your sensors might not! In essence, we need multi-tasking for more complex USB controller that can do many other things other than just reading buttons and ADC values.

That calls for an preemptive multitasking operating system. Usually, for embedded systems, that would be a Real Time Operating System (RTOS). I have selected the FreeRTOS as the OS for the Hempstick, because it's free, it's simple, and it's supported by the Atmel

Software Foundation, which supports the MCUs I am most familiar with.

So, the Hempstick in the background has a task reading the configured ADC values, another task reading the TM stick if configured, and yet another task reading buttons. What happens when we also want to read multiple MLX90363? Well, there is another task running in the background reading the MLX90363s. What more sensors? Add more tasks!

In essence, it is designed to be modular, flexible and expandable. Sure, there are joysticks that do all these (well, TM's Warthog sure does all these!), but I have yet to find one that is OpenSourced.

Sure, that was the original idea of writing Hempstick -- I wrote it for myself! But in the long process of learning digital design, the more I learn the more I realize that there is no magic in designing with today's highly integrated MCUs. There is just a tremendous amount of low level programming, and programming is something I have a knack of and make a living out of it. Why couldn't anybody do it? Well, not everybody is born with a knack in programming, not to mention not everybody wants to spend evenings carefully reading the dry, voluminous (1000+ pages per MCU is the norm), and often confusing and unclear MCU specification sheets. But what if the programming part and the careful reading of MCU specification sheets are taken out of the equation? After all, regular desktop and server operating system APIs do exactly that! What if I can make a design that is modular and flexible and hides all the gory details (if not all) of controlling the MCU and its peripherals from the non-programmers? Can I apply good software engineering design to the construction of the Hempstick and isolate the details into their own modules so that it not only becomes easy for me to cobble together different USB controllers and make it simple to configure all those sensors and peripherals needed to construct a USB controller?

That is, the Hempstick software is also designed from the ground up to allow the non-software-engineers to do some modifications and produce customized USB firmware.

Read Chapter 1 the Super Quick Start Guide to see how easy it is for non-software-engineers to compile and burn a USB firmware into a blank SAM4S XPLAIN Pro board and produce a functioning Hempstick!

Once you get to see how easy it is for you to build a Hempstick from the source code and have it up and running, I hope you will be itching for custom modifications and producing some USB controllers that you can call your own, like putting your own call sign on the stick!

You can use it to build your own instrument panels, flight instruments, helo collectives, or simply turn your old game port rudder into a USB rudder. Or may be, you want to do what I often say -- “With this, I can turn a wooden stick into a USB controller.” It’s up to you.

Chapter 1

Overview

The Hempstick is an OpenSource general-purpose USB HID joystick firmware.

It is designed to be open, modular, flexible, and suited for simple modifications, like buttons, ADC, TMStick, and modifications of USB VID/PID/device name, etc. by end users to produce custom USB joystick controllers.

The end users producing custom USB firmware? Yes, you read me right. And by end users, I mean Joe-Six-Packs. And, I really mean the end users modifying the source code, compiling a new firmware and burn it onto the embedded MicroControllers (MCUs) themselves.

But I am not a programmer! You may scream! No, don't be scared, and I am not crazy. By modifying the source code, I really mean modifying the configuration, like which pin goes with which button, and maybe a little simple boilerplate assignment statements that everybody can copy-paste-n-change, and then press a button in the Integrated Development Environment (IDE) to compile and burn the the firmware to the MCU.

Trust me, anybody can do it.

Furthermore, if you have a knack in programming, and you really want to understand how it works, modify it, producing custom Hempstick controllers, or adding new features, the Hempstick is designed to be modular to be able to accommodate that. For this, I did not design it for you. I designed it like this for myself so I can add new features I want without much fuzz. You just get to tag along and benefit from it.

Features

1. Supports both OpenSourced Arduino Due/X board, and some selected Atmel Evaluation Kits hardware boards.
2. Max. 16 Analog Digital Converter channels.
3. Max. 32 buttons (Windows' DirectInput limitation).
4. Able to read buttons onThrustMaster's Cougar/Warthog sticks.
5. 12-bit hardware ADC, improved to 14-bit with oversampling.
6. Software average noise filter for ADC.
7. Built-in hardware debouncers and pull-up resistors.
8. No ghosting, no shadowing, because no button matrix arrangement, i.e. you can press all 32 buttons simultaneously if you want. We simply connect each button directly to an MCU pin. That is, we brute force the problem away with bigger MCUs.
9. Software debouncers for TM sticks.
10. Built-in pull-up resistors.
11. Contains a Real-Time Operating System (FreeRTOS), so that it can accommodate more complex features and provides flexibility for more modular design. Allows me to design it in a modular way to abstract out the configuration for end users to

modify them without requiring deep embedded programming knowledge in order to produce a custom joystick controller.

12. End users can easily change the configurations like USB VID/PID, manufacturer, device name, button/ADC assignments etc. to produce truly their own custom controllers.

Planned Features

1. Read and program MLX90363 Hall Effect Sensors in pure digital SPI mode (current works well with analog output Hall Effect Sensors using ADC).

2. PWM output for controlling LED dimming.

The matrix problem arise mainly due to using small MCU without enough pins to serve the number of buttons. It's a very cleaver way of doing more with less. But it has the ghosting and shadowing problems. Well, there is no free lunch, like they said!

We basically solve the problem by using bigger MCUs that have enough digital pins to connect to the number of MCUs we want. This approach has some additional benefits.

1. Modern MCUs' digital pins these days come built-in with hardware debouncers. So, by connecting each button directly to a digital pin, we eliminate the need to wire up external hardware debouncers, or using a software debouncer in the firmware.

2. Modern MCUs also come with pull-up resistors for each of their digital input pin. By connecting each button directly to a digital input pin, we also eliminate the need to wire up external pull-up resistors.

These simplifies the end-users wiring tremendously. Big deal? You say? Not such a big deal really I would agree. But it does save a lot of wirings. You can just directly connect a button to a digital input pin and the ground pin. Total 2 wires instead of having to wire the digital input pin to a breakout board with resistor arrays on it, and then wire it up to your buttons. You cut the wiring by half, and saves a breakout board.

Cutting the wiring job by half, you basically also eliminated the chance of wiring error by half! That is a big deal for me for any non-trivial pit building project!

How difficult is it to get a Hempstick up and running?

Not difficult at all. For a pre-configured Hempstick, all you have to do is the followings.

1. Purchase an Arduino Due/X board (USD $50), or an Atmel SAM4S XPLAIN Pro board (USD $39), not from me!

2. Download and install Atmel Studio (free as in beer).

3. Download libHemp.zip, Hempstick.zip, and unzip them.

4. Connect the board via USB.
5. Launch Atmel Studio, open the Hempstick solution, select the desired pre-configured project, and press the “Run” button.

That’s it! I am sure you are capable of installing a software, download some zip files, connect a board via USB ports, open a file, and press a “Run” button! If you can’t even do these, then Hempstick is not for you.

Of course, after the USB Joystick controller board is burned, you will still need to wire it up to all your pots, sensors, buttons, etc. Those, you are on your own. I can provide some guidance, but since I don’t know your “custom” design, I really can only provide some general recommendations only. But I will provide detailed pin outs for pre-configured Hempstick that I support.

In the first few following chapters, I will first provide step-by-step instructions quick start guides, with screenshots, to install needed software, and produce a working Hempstick boards.

If you are familiar with embedded programming or general software programming, you may want to just skimp through the quick start guides in the first few chapters just to see what needs to be done for producing a pre-configured Hempstick.

In the later chapters, I will provide instructions on how to modify and produce custom Hempsticks that you can call your own!

Chapter 2

Hempstick Super Quick Start Guide

This Super Quick Start Guide will show you what you will need and do to create a Hempstick board from a blank Atmel SAM4S XPLAIN Pro board.

This chapter contains step-by-step instructions on what software to download and install, as well as what needs to be done to compile from source code and burn the resulted firmware to the board, resulting in a fully functional Hempstick board.

Section 1

What You Need

For the hardware board for the Super Quick Start Guide, we choose the Atmel SAM4S XPLAIN Pro board for the reason that it contains an Atmel EDBG (Embedded Debug) chip allowing us to program the on-board MCU (MicroController Unit) without an additional hardware programmer or debugger.

Hardware Needed:

- An Atmel SAM4S XPLAIN Pro board, http://store.atmel.com/PartDetail.aspx?q=p:10500344;c:100113#tc:description, USD $39.
- 2x micro USB to USB cable.

Software Needed:

- Windows OS (XP, Vista, or Win7, preferably Win7).
- Atmel Studio, http://www.atmel.com/tools/ATMELSTUDIO.aspx.
- Hempstick source code, at GitHub libHemp-master.zip and Hempstick-master.zip.

Section 2

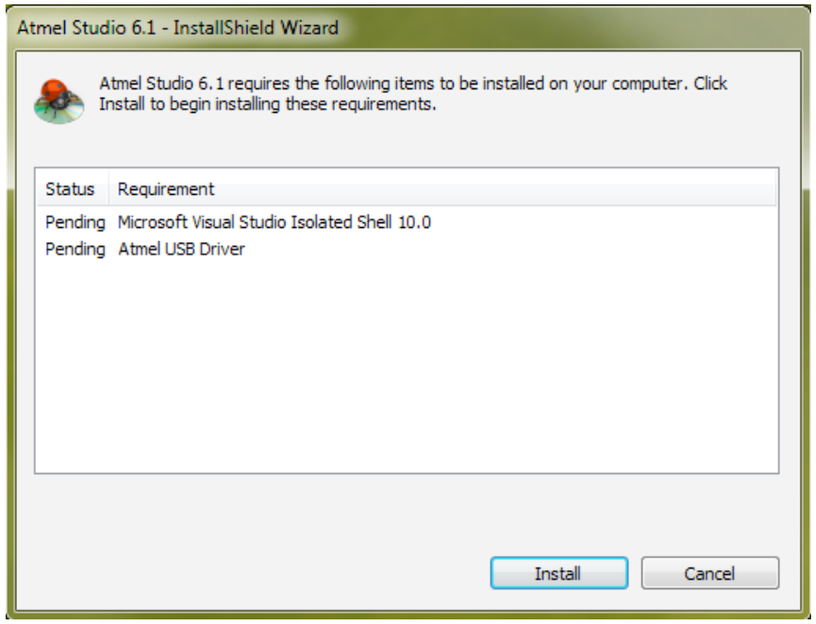
Installing Atmel Studio

Atmel Studio is an IDE (Integrated Development Environment). It contains the full suite of development tools needed to edit, compile, program, and debug firmware on Atmel MCUs. For the purpose of this Super Quick Start Guide, that is all the software you will need, in additional the your favorite browser.

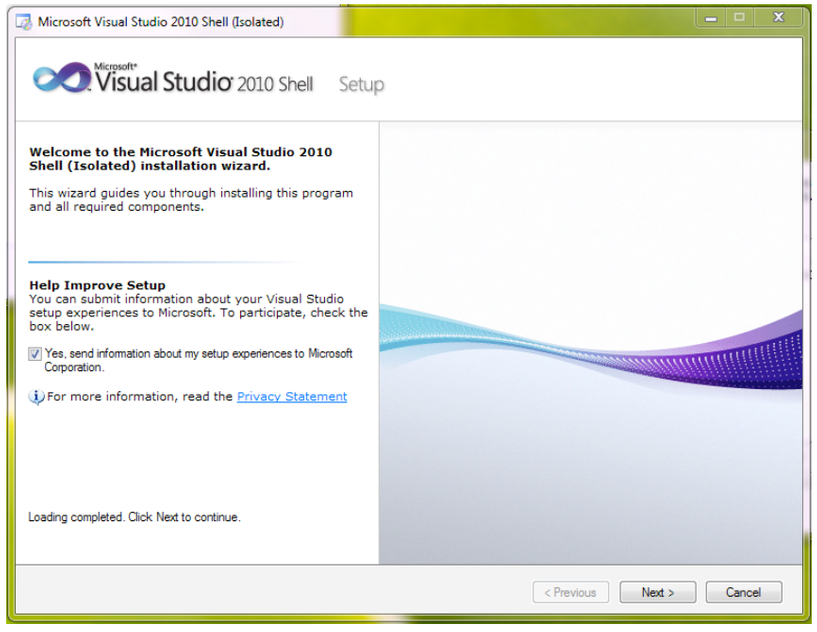
This section will show you the step-by-step instructions on how to install Atmel Studio. If you are familiar with it, then skip over this section.

Once you have downloaded the latest Atmel Studio software (registration might be required for the download), launch the installer.

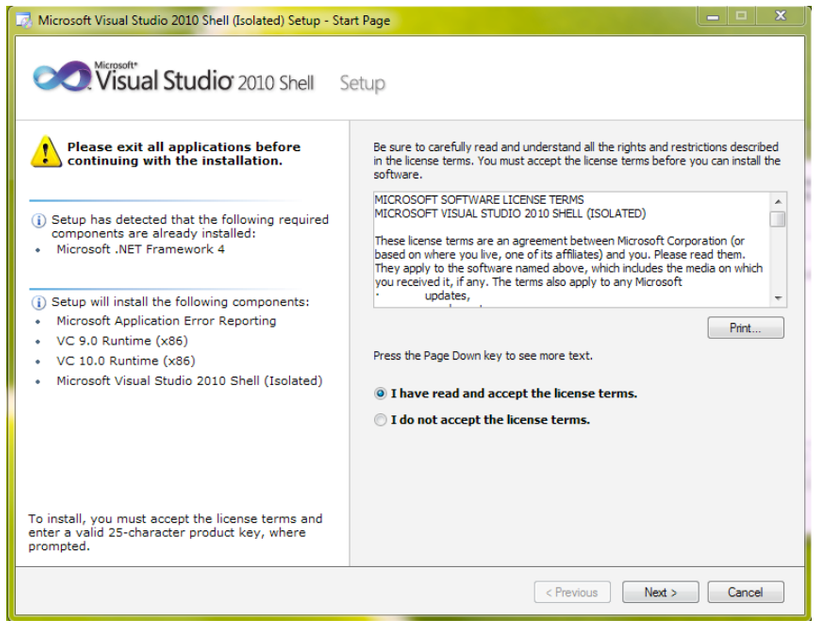
1. Here, it says that it will need to install two additional software. Click **[Install]**.



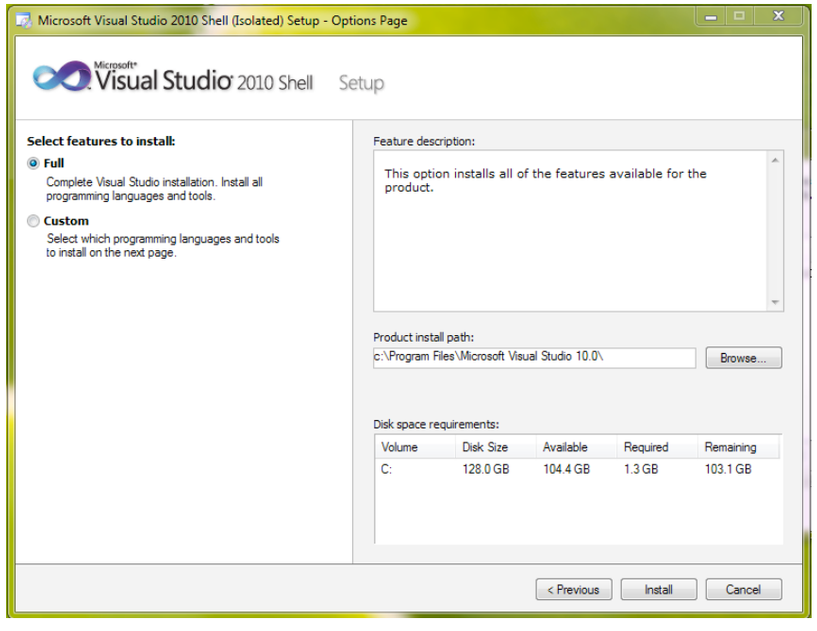
2. Uncheck the **[Yes, send information to the mothership...]** and click **[Next]**.



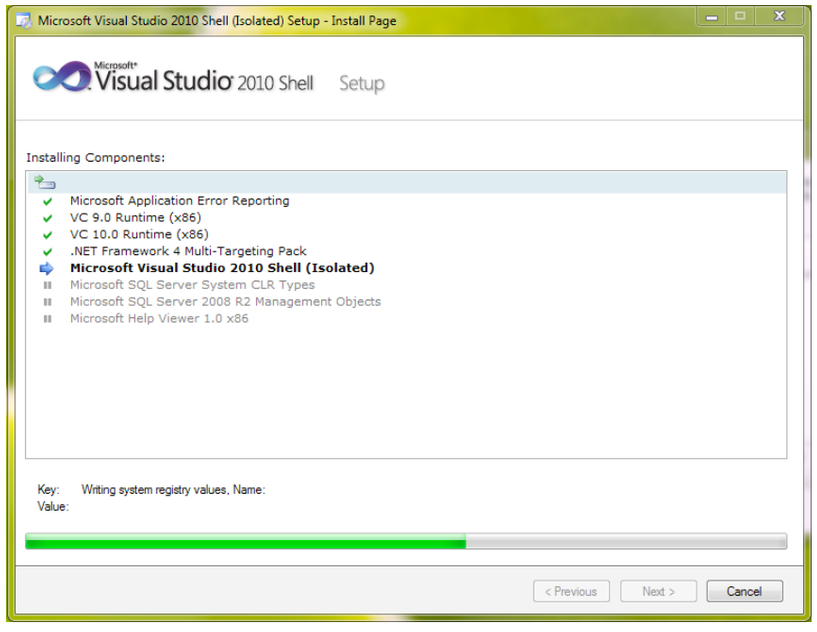
3. Check **[I have read and accept the license terms.]** and click **[Next].**



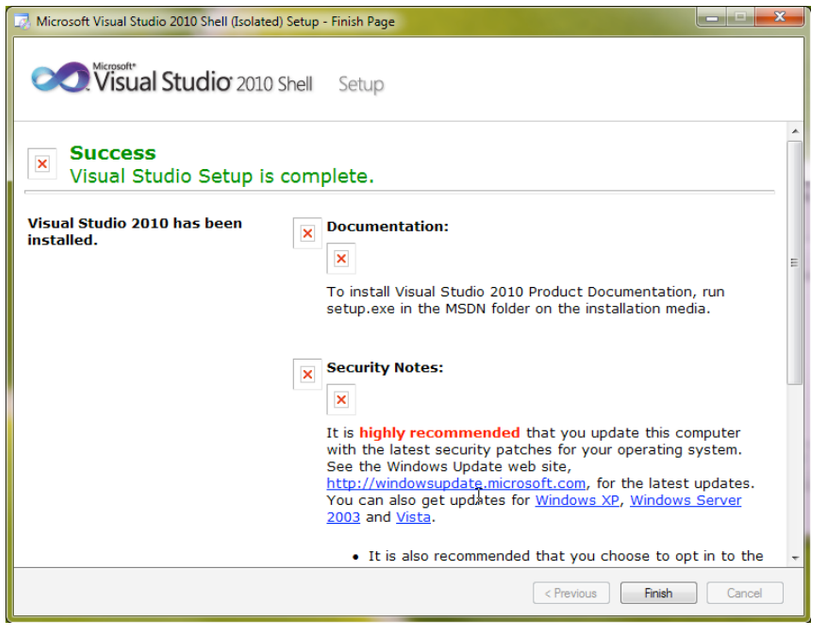
4. Check **[Full]** and click **[Install]**.



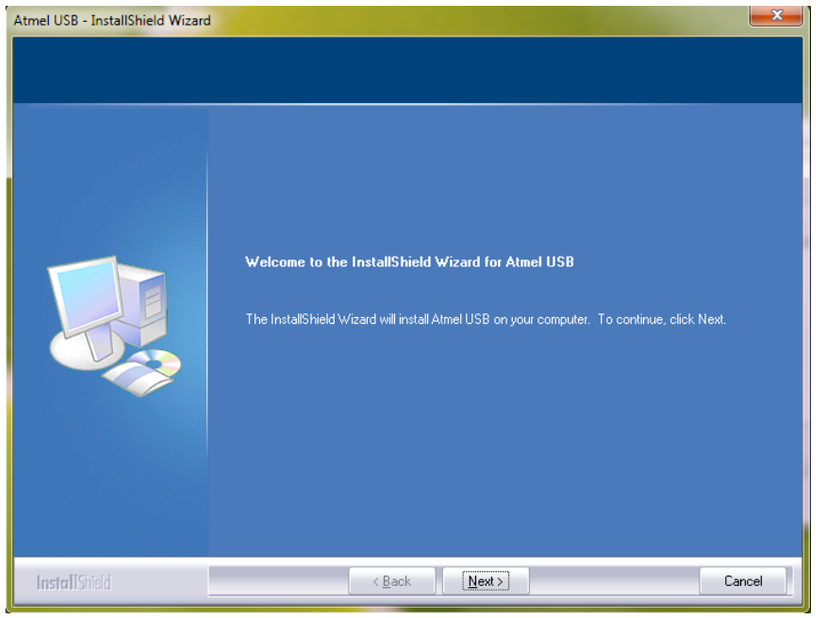
5. This will install everything Visual Studio Shell needs including the proper version of .NET if you don't already have it. Wait...



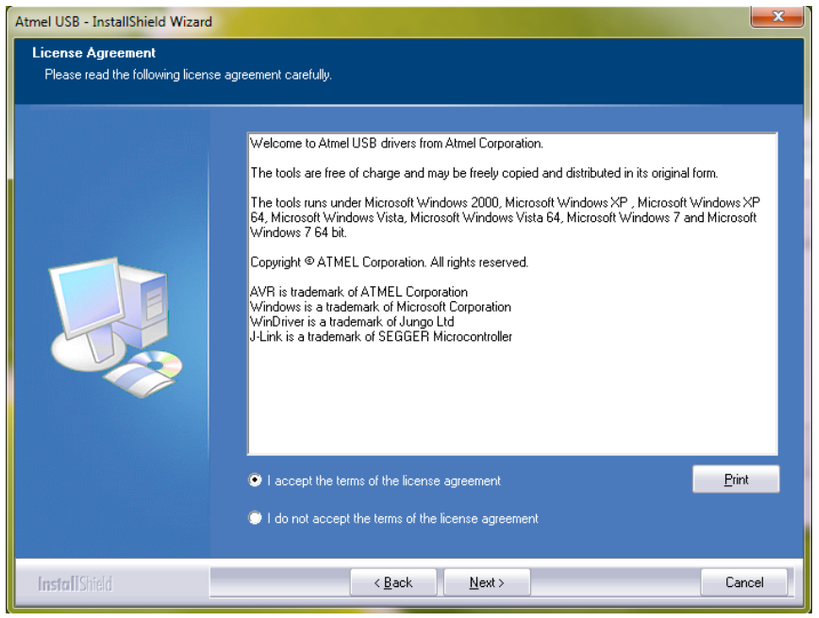
6. Visual Studio Shell installed correctly. Don't worry about the broken graphics. It is ok. Click **[Finish]**.



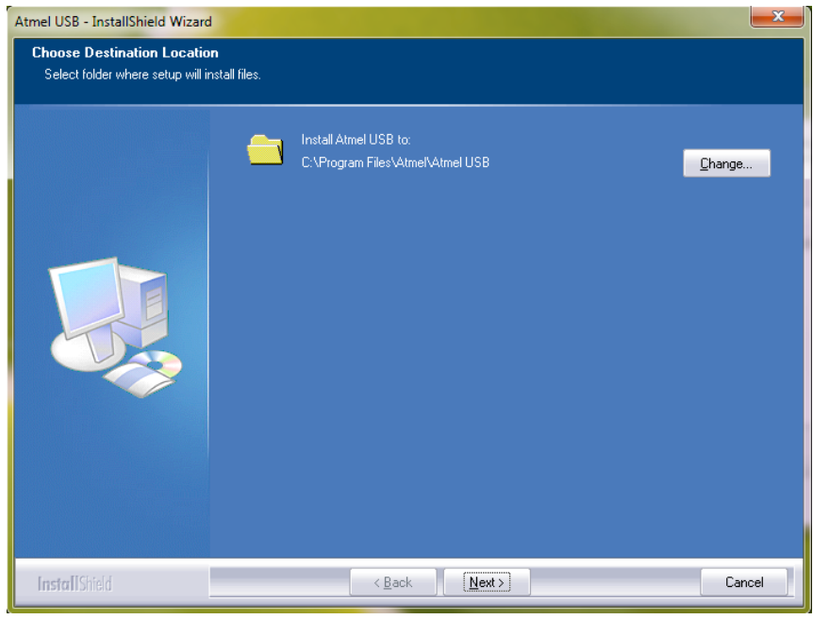
7. Now starts the Atmel USB driver installation. Click **[Next]**.



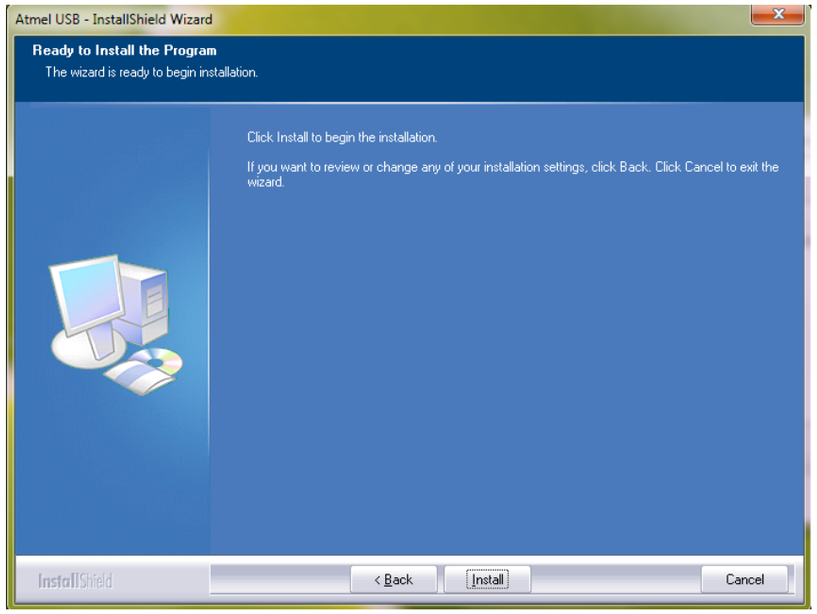
8. Check **[I accept...]** and click **[Next]**.



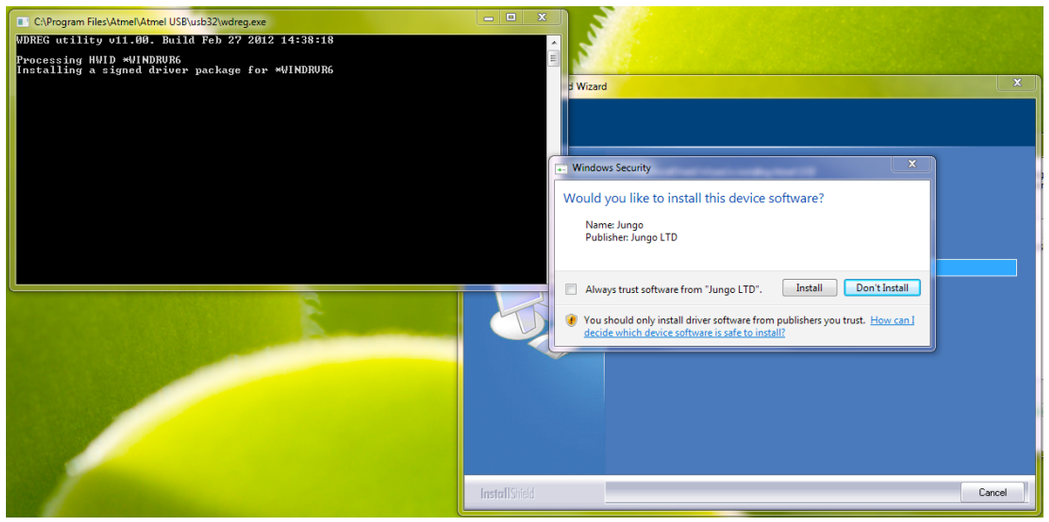
9. Accept the default location, and click **[Next]**.



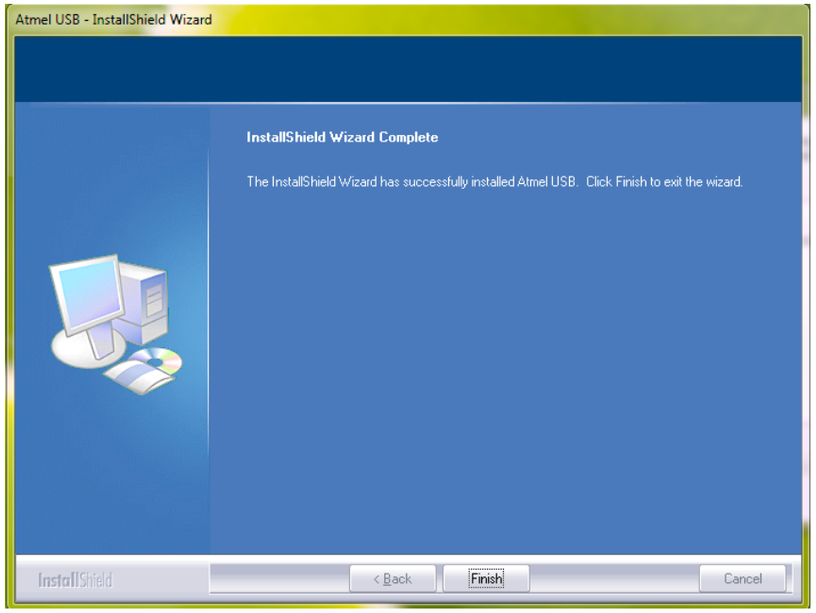
10. Click **[Install]**. This will launch a command line window and install a Jungo/WinDriver USB driver. Jungo is what Atmel uses to write their custom USB driver(s).



11. Click **[Install]**.

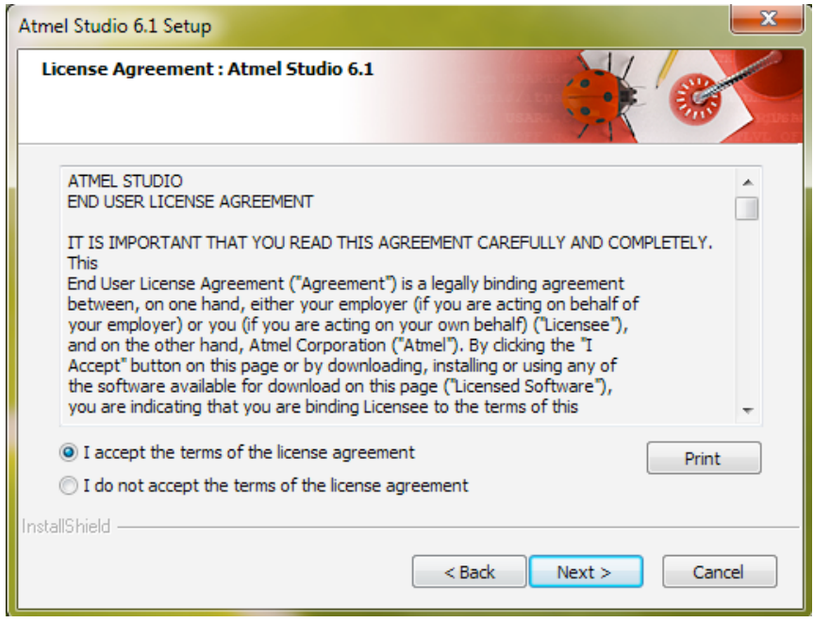


12. Finished. Click **[Finish]**.

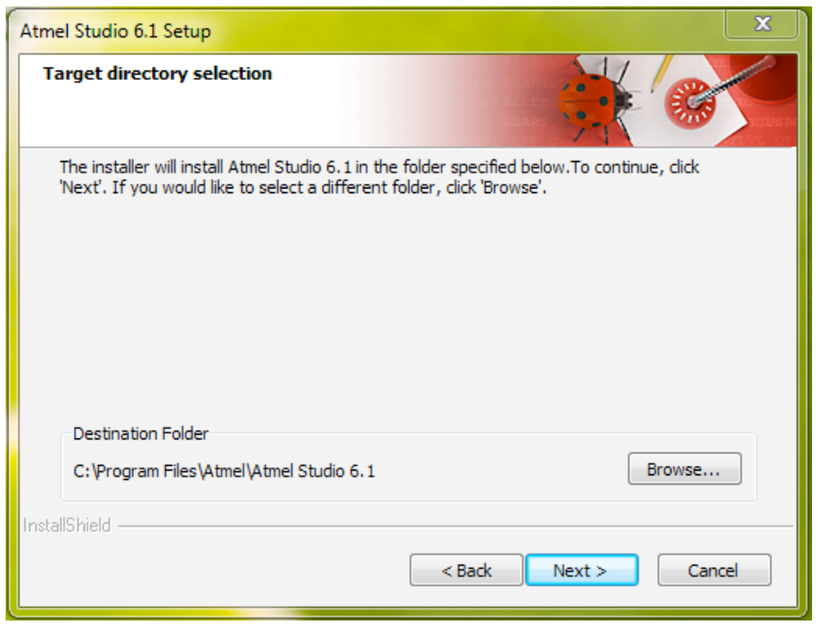


13. Now, we really start the installation of Atmel Studio. Click **[Next]**.

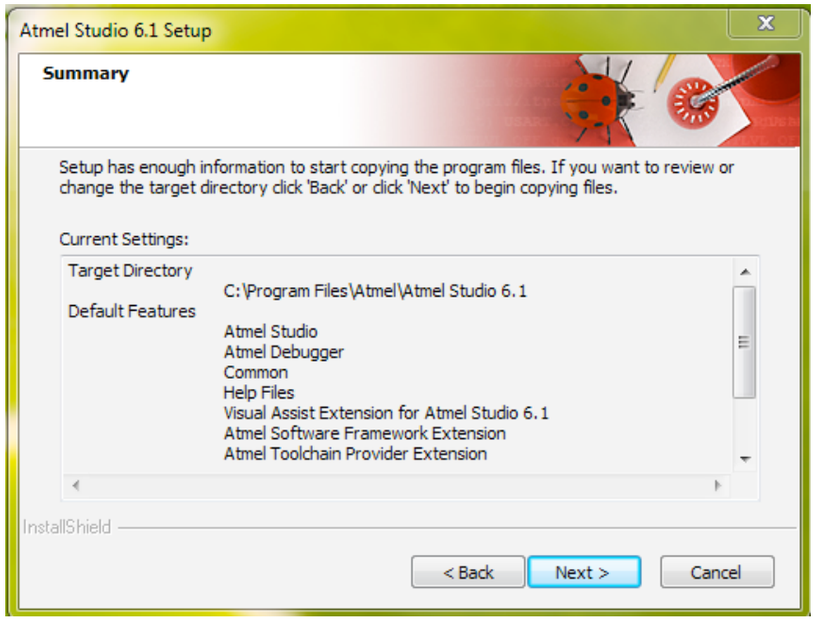
14. Accept the license agreement and click **[Next]**.



15. Default location is good, click **[Next]**.



16. Last chance to change your mind..., click **[Next]**.



17. Just click **[Finish]** without check the option, unless you know what you are doing.

Done. The installation of the Atmel Studio is completed. You should see a lady bug icon on your desktop. That's the Atmel Studio's shortcut.

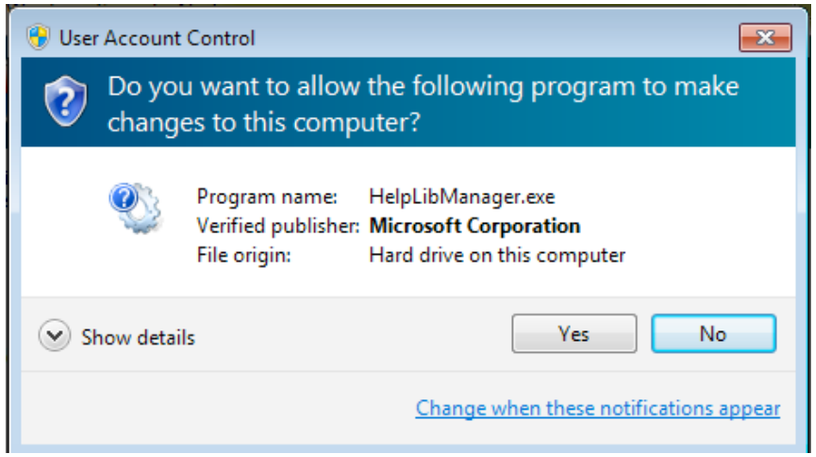
MAKE SURE YOU RESTART THE WINDOWS OS, OTHERWISE THE NEWLY INSTALLED USB DRIVER MAY NOT WORK!

Starting & Working With Atmel Studio

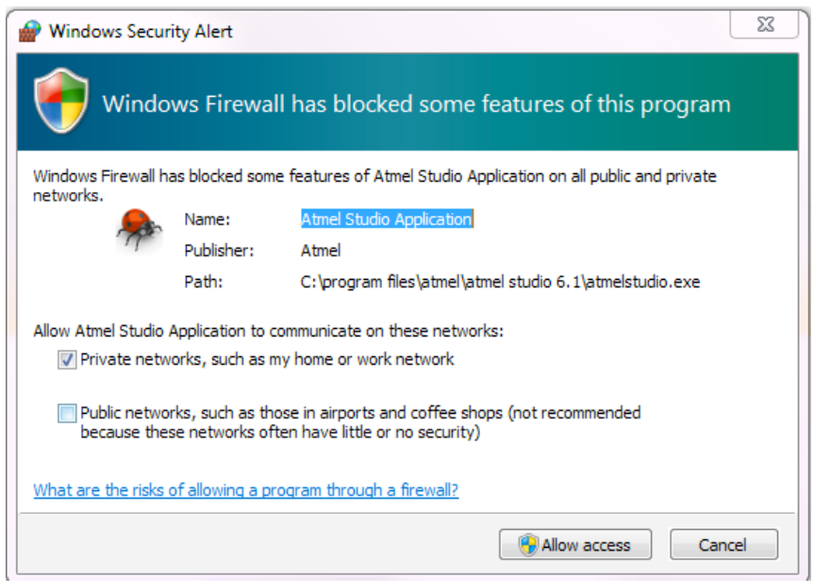
Once the Windows OS is rebooted, you should find that you have a red Lady Bug icon on your desktop. That is the Atmel Studio icon. Launch it.

The first time you launch Atmel Studio, there is a little bit of setup to do, in particular if you are on the newer Vista or Windows 7. These have very nagging security model to the point of significantly reducing productivity, particularly in the hands of clueless and draconian system administrators (I have seen such a “corporate security policies” with which you have to run EVERYTHING as Administrator, even for a text editor, which defeats the whole point of the multiple levels of the security model!).

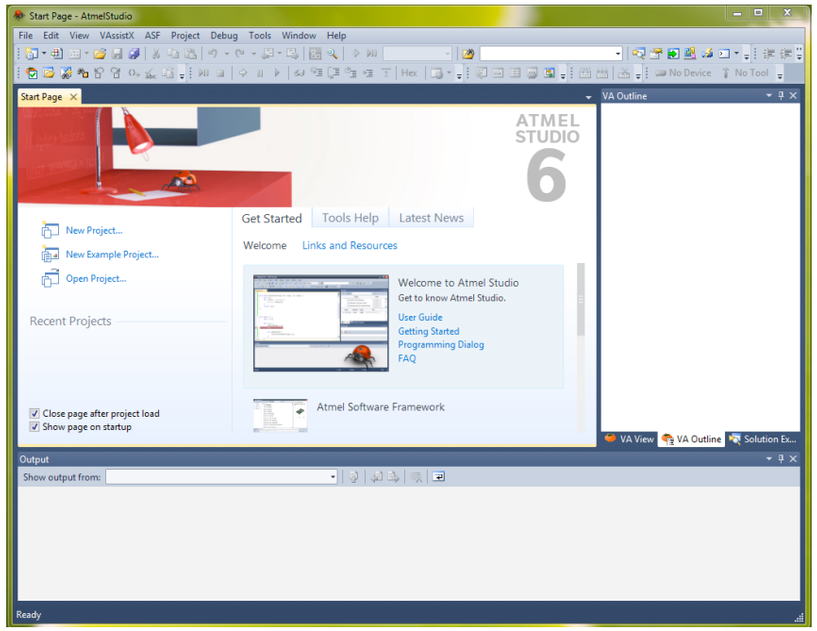
Suffer the UAC first! Say **[Yes, dear]**.

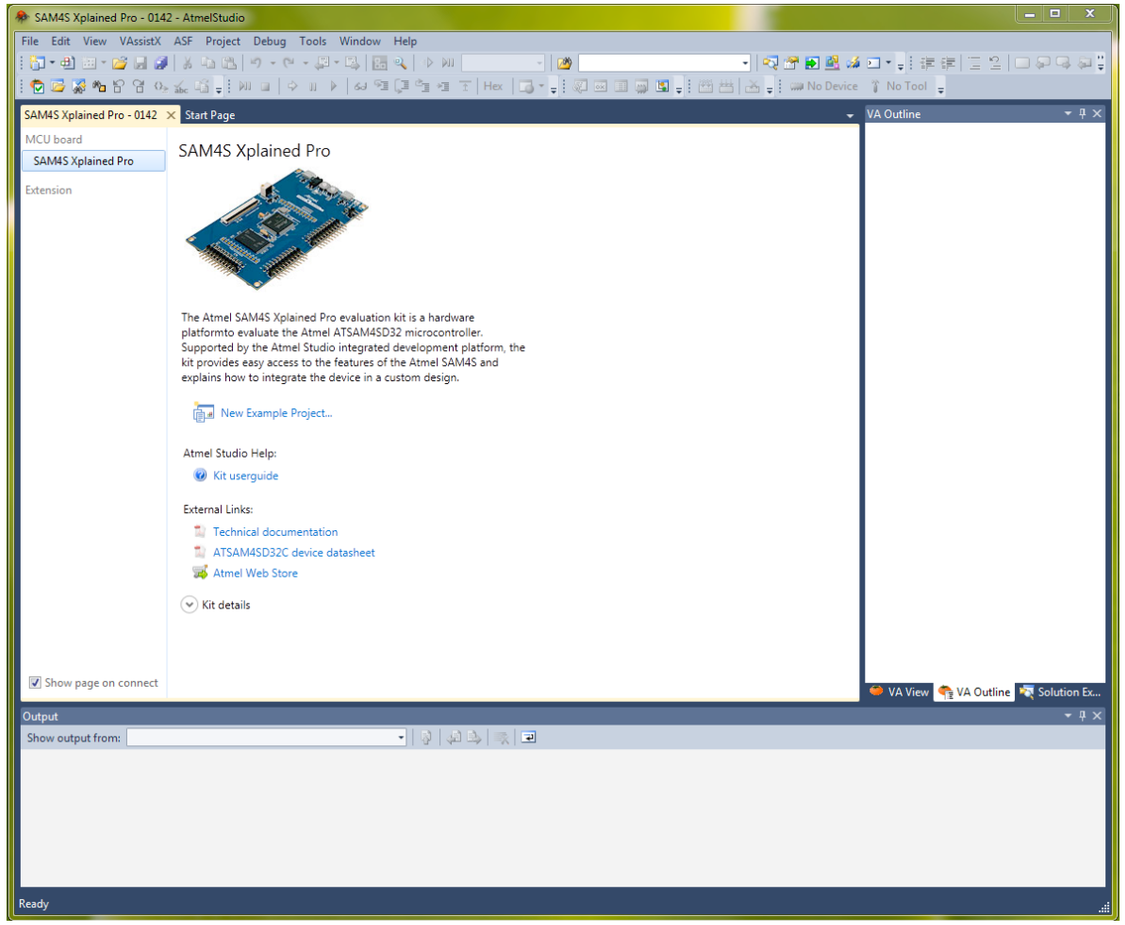


Now feel the wrath of Windows Security! Click **[Allow Access]**.



You should now see the main window of Atmel Studio showing up, with a Start Page.



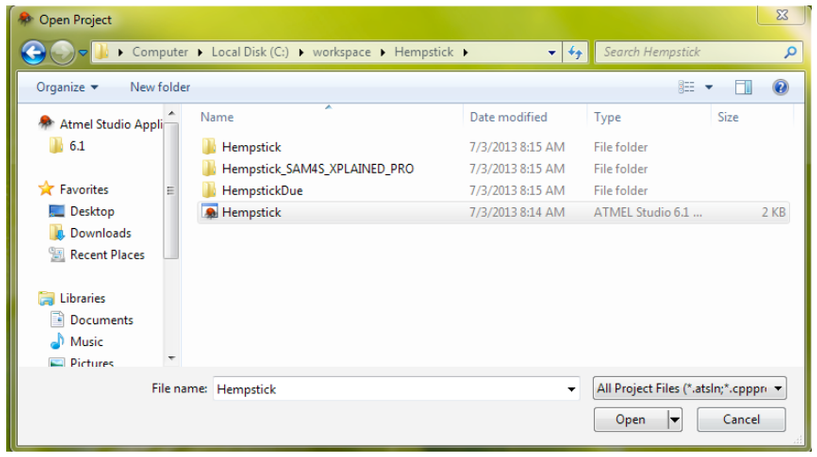


Now, connect the SAM4S XPLAIN Pro board's Debug USB port to the host computer's USB port.

The connection of Debug USB port to the host computer should trigger the Windows OS to automatically find the appropriate driver, which we installed during the process of installing the Atmel Studio. You shouldn't need to do any manual step for it. If everything goes fine, you should see a new SAM4S XPLAIN Pro tab showing up on the main window. We don't need any of these two tabs, close them if you want.

Section 4

Building Hempstick



We finally have all the tools we need setup and arrive at where the real work begin -- building a Hempstick board.

We have three tasks to do.

1. Download the latest Hempstick source code.
2. Compile it and burn the firmware to the board.
3. Test the board.

Download the latest Hempstick Source Code

Go http://www.hempstick.org to download http://www.hempstick.org/download/libHemp.zip and http://www.hempstick.org/download/Hempstick.zip. Then extract the two zip files into the **same folder**.

It is important to put both in the same folder in a convenient location with short path name. If you don't, you will need to modify the build settings in the project, which is beyond the subject of this Super Quick Start Guide.

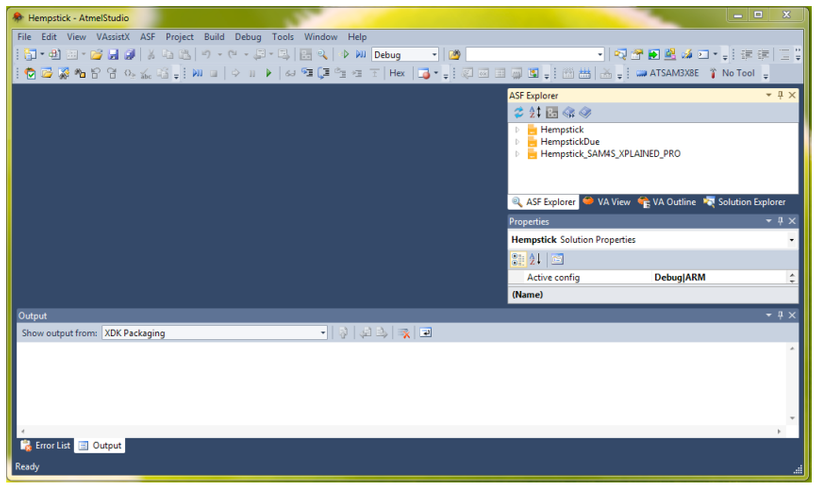
I would suggest that you put it in something like c:\workspace\. From here on, I will use c:\workspace\ to refer to the location of the source code.

Compile & Build Hempstick

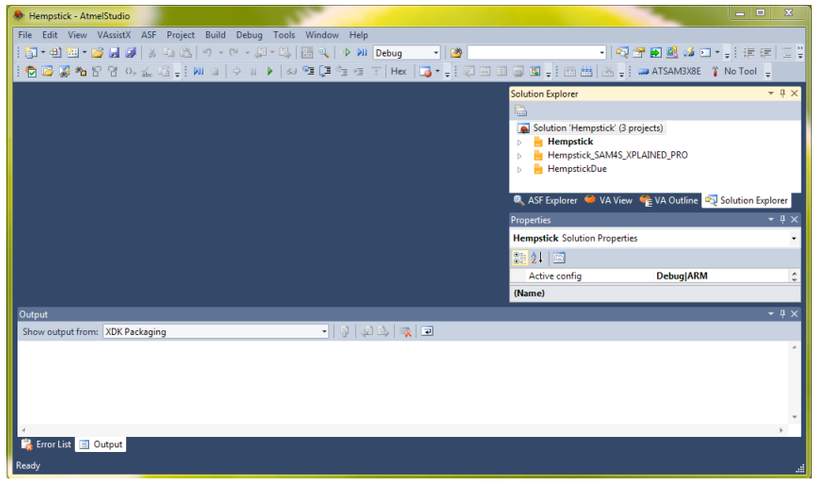
In the main window of Atmel Studio, click the following in the menu: **[File]** -> **[Open]** -> **[Project/Solution]**.

Navigate to c:\workspace\Hempstick\, and select the Hempstick solution, as shown above.

The first time you launch Atmel Studio and load up a solution, it may automatically show you the ASF Explorer pane on the upper right hand corner. This pane is not very useful for our purpose. Go to the lower right hand corner of the pane, find the tab called Solution Explorer, and click it.



Now select the Hempstick_SAM4S_EXPLAINED_Pro project in the Solution Explorer pane, and click  button in the toolbar on the top, right below the menu bar.



This will do the followings.

- Compile everything needed for compilation in the selected project.
- If everything goes well, load the resulted firmware binary and burn it to the selected board.
- Run the newly burnt firmware.

If you see warnings on the bottom output pane. Don't worry. It's normal.

If everything goes well, you should see first the green status LED blinking very fast. This is the indication of activities on the EDBG bus. Then, if the firmware runs successfully, you should see the orange LED0 starting blinking at a rate of 1Hz. The LED0 blinking at 1Hz is a feature in Hempstick to indicate that the embedded FreeRTOS is executing tasks correctly.

Now, now.... you have just made a Hempstick board.

I am not kidding that it is easy, did I?

Test The Board

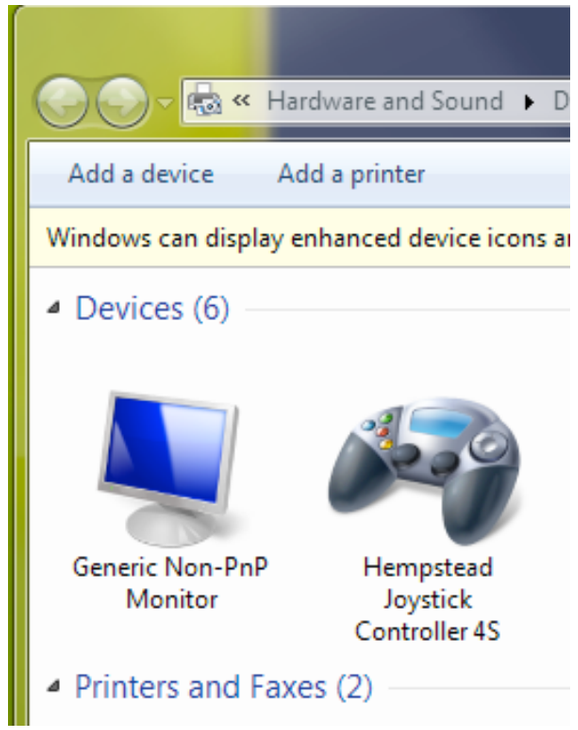
Now we need to run some basic joystick tests.

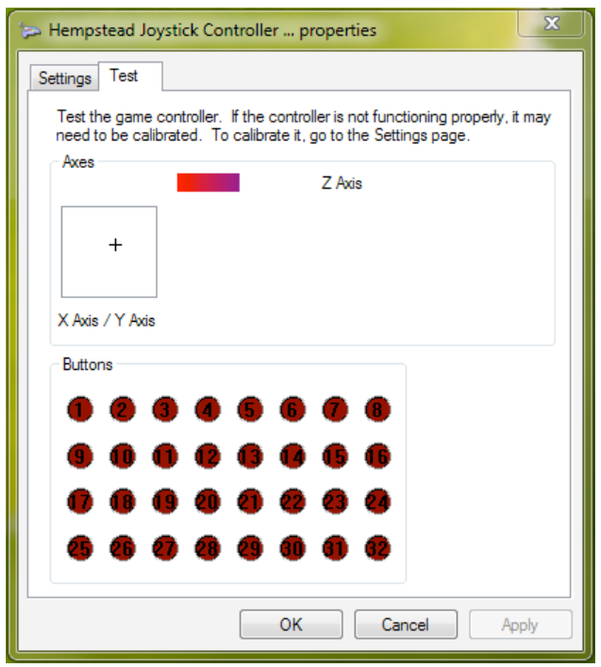
Plug in another micro USB cable to the USB port on the SAM4S XPLAIN Pro board marked SAM4S USB and connect the other end to your host computer (same one as the one your run Atmel Studio is fine).

This should trigger a driver installation on your host Windows OS. SInce the Hempstick is a USB HID joystick device, Windows come with default driver, which will work just fine.

Now, go to **[Start]** -> **[Devices and Printers]**, and you should see the Hempstick showing up as a game controller.

Select the **[Hempstead Joystick Controller 4S]** -> **[R-click]** -> **[Properties]**.





Section 5

Epilogue

Although we just showed you how simple it is to make a Hempstick board, you would still need to wire up the board for your controller function. The pin assignments will be shown in other chapters.

Also, note that, the EDGB built into the SAM4 XPLAIN Pro board, although saves you from having to obtain a hardware programmer/debugger, it is nevertheless a very new technology from Atmel. It communicates with the host computer via USB, and the USB driver is very twitchy. You will need to shut down the EDBG in a very specific order or it will never work again until you reboot the PC (and the shutdown of the OS will get stuck; that bad). Here's the proper way of shutting down the EDBG.

1. If you are in any debugging session, STOP the debugging session.

2. Unplug the Debug USB cable from the SAM4S XPLAIN Pro board.

3. NEVER ever rewire the pins while plugged into Debug USB port, or it will cause the EDBG (or driver, I don't know which). to freeze and then the Atmel Studio will never be able to communicate with it correctly until reboot. This is particularly true, if you plug or unplug a ground pin!

Until Atmel fix this problem, you should follow the above procedure every time you are up for rewiring the board.

Chapter 3

Pins & Wiring

A Hempstick is useless unless you wire up something, like ADCs (Analog-Digital Converter), Hall Effect Sensors, buttons, and switches.

This chapter gives you some of the OOTB (Out-Of-The-Box) pin assignments and wiring instructions.

A Few Words About Pins & Wiring

The Atmel's new series of ARM chips, like the one used in the SAM4 XPLAIN Pro board, are 3.3V devices and unfortunately, they are not 5V tolerant. Meaning, if you wire up 5V to a pin directly, you risk frying the MCU chip!

5V vs. 3.3V Power Supply

Since the USB bus is a 5V bus and a lot of TTL era sensors are 5V devices, and the Atmel evaluation boards or Arduino boards usually come with both 5V and 3.3V power pins, you must be very careful on whether you power the sensor with 5V or 3.3V. For some sensors that require 5V, like some older analog Hall Effect Sensors from Allegra, you will need some voltage level converters. In some situations, a simple resistor-based voltage divider would suffice. For more complex 5V logics, full bi-directional logic converters may be needed. On the other hand, programmable Hall Effect Sensors like the MLX90333/MLX90316 can be programmed to only output up to 3.3V, eliminating the need for a voltage divider.

The bottom line is that the Atmel ARM MCU pins may get fried if your sensors feed them 5V.

Fortunately, for the run of the mill buttons and potentiometers that we commonly used in joystick and controllers, wiring them up directly to the on board 3.3V power would do just fine.

Pull-up Resistors & Debouncers

In addition, for buttons and switchers, usually some hardware debouncer and pull-up resistors would be needed.

Fortunately, the MCUs supported by the Hempstick all have built-in pull-up resistors and hardware debouncers on chip and they are all by default enabled when configured for buttons and switches so you do not have to wire up additional pull-up resistors and debouncers.

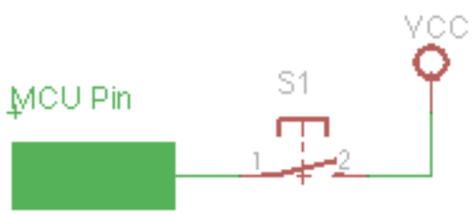
What are the pull-up resistors and debouncers then?

Pull-up Resistors

Imagine that an MCU pin is configured to read 0 or 3.3V and give the firmware reading the voltage as either 0 or 1 digital values. Now, we wire the pin to a momentary button.

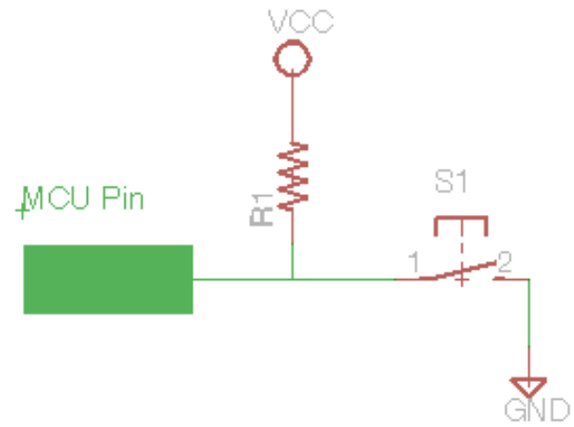
A momentary button has two pins, when you press the button, it connects the two pins electrically, and when you release it, it disconnects the two pins, sort of (more on this in the debouncer topic).

How do we wire it up to the MCU pin then? We could connect one button pin to the Vcc (3.3V), and the other pin to the MCU pin, like so.



So, if the button is pressed, the 3.3V Vcc will be directly connected to the MCU Pin making it sense 3.3V and reads logic 1. Good? No! What happens when the buttons is not pressed? Is the MCU Pin going to read 1 or 0? Nobody knows. It is called a *Floating Pin*.

To solve this problem we usually connect the pin the following way.



This way, when the button is not pressed, we guarantee that the MCU Pin will read 1 and when the button is pressed, it reads 0. The only trouble is now that when the button is pressed the MCU read 0 and when it is not pressed, the MCU reads 1, exactly the opposite of what we would have thought. We can easily revert that in software. No worry, and Hempstick source code does that for you.

Note that, we need to chose an R1 resistor value suitable for the circuit so that it does not sap too much power, whether the button is pressed or not. When not pressed, the MCU would have a very small internal resistance and consume some neglectible power, and when pressed, the MCU pin gets pull to the ground 0V, and it will consume Vcc/R1 amp. The common value to use is 10K ohm or 100K ohm for such circuits.

Fortunately for us, the Atmel MCUs, like I mentioned earlier, come built-in with a 100K ohm pull-up resistor for each I/O pin (Input/Output pin) and they can be activated by software.

So, there is no need for you, the end user of Hempstick, to wire up pull-up resistors. The Hempstick firmware will configure the MCU to do so for buttons.

Debouncers

When we switch on a light switch, we tend to think that the switch contact connects the wire and the current flows, the light goes on. That's it.

On a macro conceptual level, yes, that is true. But not so when the MCUs are capable of reading thousands or millions of values per second. The contact actually connects and disconnects a couple of times during a few millisecond time frame. This is called *Switch Bounces*. We humans cannot see it, but the MCUs can. How many times it bounces and how fast depends on a lot of factors, including material, force applied, physical design of the switches, etc. There is no simple and fast rule for it. A lot of it depends on empirical tests in order to determine the timing required to take out the bounces.

Let me put it another way. I don't know anybody designing switches with certain bounce characteristic, nor do I know any switch or button that comes with specifications of its bounce characteristics. Maybe some special military application or NASA switches come with such spec., I certainly have never seen one.

You don't want to press the trigger of your joystick one time and fire several missiles during a millisecond, would you? Rapid fire is nice for StreeFighters, not for simmers. Even for the StreetFighter game, this behavior is not very desirable, I mean, you cannot control how many bounces it does so can't you reliably pull off some awesome combo kicks from Chun-Li correctly. No good! The bounces must go, and if rapid fire is desirable, it must come from a more controllable and predictable mechanism.

There we come to the subject of *Debouncers*.

I will not bore you with the theory of debouncers. Let's just say that there are two kinds of debouncers in general, software and hardware debouncers.

To use a hardware debouncer solution, you obviously have to buy it, and wire it up yourself. To use a software debouncer, you have to write the software to, say in a few milliseconds, read the same pin's value a few times, and only if the value stabilizes do you count it as a button press or release, and ignore all the transient bounces.

Again, the Atmel MCUs the Hempstick support all have such hardware debouncers built-in for each pin and can be activated by software configuration. Hempstick does that for you so you do not have to wire up your own hardware debouncers!

The Hempstick is also capable of reading ThrustMaster (TM) Cougar/Warthog sticks. Unfortunately, the TM sticks are not equipped with hardware debouncers. So, the Hempstick, when configured to read TM sticks, it will internally perform software debouncing for you. Again, you don't need to do anything.

Note that, whether it is a software or hardware debouncer, nothing is perfect. They don't get to filter out all bounces of all types. They only filter out the most commonly occurring type of bounces.

The good news is that for human interaction, you most likely will not see any of these "glitches" from Hempstick. But I cannot guarantee that you will never see them.

About Residual and Floating Values of Unwired ADC Channels

In the default configuration of the Hempsticks, we configure at least 8 *ADC (Analog-to-Digital-Converter)* channels for each controller. This gives you X, Y, Z, Rx, Ry, Rz, Slider, and Dial for Windows' maximum 8 ADC channels.

However, if you do not wire up all ADC channels, say you only wire up X, Y, Z, and Rx. The rest of the unused channels may get residual values from wired up channels. This is because, there is only one ADC module, and all the 8 channels "multiplexes" on the same ADC hardware. It is like a time-sharing scheme in using the computer CPU; we sample one channel at a time, one after another, and if we sample them fast enough, it looks like 8 channels are all sampled concurrently all at once to the end user. However, there is also only one "sample-and-hold" hardware in the ADC module. So, since you did not wire up Ry, Rz, Slider, and Dial channels, they still get sampled after the X, Y, Z, and Rx, and the sample and hold hardware may have some residual charges left in the "hold" part of the hardware from the previous sampling of wired up channels. That, would appear on the unwired channels. And you would see some random values showing up on these unwired channels. The residue values seem to be decreasing in magnitude by the sampling order of the channels; but the sampling order is not deterministic (but mostly in the order of ADC channel number.).

That is a very unfortunately side effect of accommodating users who do not have the technical skills or unwilling to spend the time to produce a customized firmware for themselves.

If you do not wish to see these floating and residual values, then you have two choices.

1. Wire the unused ADC pins to ground with a resistor, say 100K ohm.
2. Modify the Hempstick configuration to disable the unused ADC pins.

Method #1 above is basically to ground the unused ADC. BTW, in electronics, leaving an unused pin floating is usually a bad thing. The resistor is needed because when the sample and hold circuit in the ADC samples, internally there is a capacitor holding some small charge gets connected to the ADC pin. If you wire the unused ADC pin directly to the ground, when the ADC switches to that channel and starts to sample, the left over charges from the previous channel will flow directly into ground without any resistance and it might cause some unknown large current for a short time damaging the circuits. The capacitor here is usually a very small one so the charge is small, but it is not specified. So, we wire up a high value resistor to ground, serving as a safe discharge path to dump the charge inside the capacitor.

Another way of solving this “problem” is that you could wire the unused ADC pins to the 3.3V voltage supply. Again, it would be advisable to put a resistor between the unused ADC pins and the 3.3V, even though I did try wiring them up without any resistor and nothing blew up. Sure, the sample-and-hold capacitor in the MCU for sure has a very small value, and it probably has some internal stray resistance value, but we don’t know how much there is. It’s never wise to rely on unknown values or wire a charged capacitor to ground or power supply directly without any resistance. You might not destroy it immediately, but you could be shortening the life of the ADC module or even the MCU itself. Although it is in general not a good idea to leave pins floating, wiring charged capacitor directly to ground is an even worse offense!

Even if you don’t do anything and leave the unused ADC pin floating, it won’t do any harm. It just looks funny.

We will discuss how to customize Hempsticks and produce your own very custom controller of your own in some other chapters.

Section 2

Hempstick 4S Pin Assignment

The Hempstick 4S uses an Atmel SAM4S XPLAIN Pro board. The board has 3 extension connectors, EXT1, EXT2, and EXT3, plus a Spare, a PIOD, and an LCD connector. The latter three are not used by Hempstick. So, in this section, we will only list the pin assignments used by the OOTB pins. In addition, there is also a PWR connector providing external power of rectified 5V, USB 5V, and a regulated 3.3V.

If you wish to use the unused connectors for more input, you will have to customize the Hempstick 4S configuration. A word of caution though. Some of the pins on the unused connectors are “shared” with the EXT1, EXT2, and EXT3 connectors. Meaning, some of the pins are internally connected to some of the pins on the EXT1, EXT2, and EXT3 connectors!

Even the EXT1, 2, and 3 connectors have shared pins. The SPI pins, MISO, MOSI, etc. Be careful when you reassign these pins!

Power Pin Assignment

PIN	Function	Description
EXT1-19	GND	
EXT1-20	Vcc	3.3V
EXT2-19	GND	
EXT2-20	Vcc	3.3V
EXT3-19	GND	
EXT3-20	Vcc	3.3V
PWR-1	VEXT_P5V0	External 5V Input
PWR-2	GND	
PWR-3	VCC_P5V0	Unregulated 5.0V
PWR-4	VCC_P3V3	Regulated 3.3V

ADC Pin Assignment

You must wire the potentiometers to the 3.3V Vcc, GND, and the assigned ADC input pins, in a three-wire configuration. Usually, the output voltage pin would be the middle pin of the 3 pins on the potentiometer, and the Vcc and GND wired to the other two pins, depending on the direction of rotation you wish to have.

For other analog sensors like the Hall Effect Sensors, you must ensure that the sensors' analog output never exceed 3.3V. For instance, for an MLX90316, you could program the internal response curve to output 0 to 3.3V with a linear response.

For other 5V Hall Effect Sensors that do not have internal response curve configuration, you could use a simple resistor voltage divider for them. I will discuss this arrangement in some other chapter/section.

ADC Pin Assignment

PIN	Function	Description
EXT1-3	X	ADC Channel 0
EXT1-13	Y	ADC Channel 8
EXT1-14	Z	ADC Channel 9
EXT2-3	Rx	ADC Channel 4
EXT2-4	Ry	ADC Channel 5
EXT3-3	Rz	ADC Channel 13
EXT3-4	Slider	ADC Channel 14
EXT3-14	Dial	ADC Channel 7

There are up to 16 ADC channels on the MCU. However, some of the channels are used for other purposes or are not routed out on the SAM4S XPLAIN Pro board. We support only 8 ADC channels for this board.

The XPLAIN Pro series of boards are designed to have the same functionality for each extension connectors, each having the same number of ADC channels, SPI, etc. That's why some of these pins and channels seem to jump around randomly at the first sight. Oh, well, they are a bit arbitrary, nevertheless, there is some logic in there.

TMStick Pin Assignment

TMStick Pin Assignment

PIN	TMStick Pin	Function	Description
	1	Vcc	
EXT2-8	2	Serial-Parallel	RF
EXT1-8	3	Clock	RK
EXT1-4	4	Data	RD
	5	GND	
EXT2-9		Clock	TIOA2, connect to EXT2-8, TMStick Pin 2

The TMStick internally has 3 8-bit cascaded JAM-Type Parallel-to-Serial shift registers. What this means is that it has total 24x inputs from buttons. When pin 2 is not asserted low (usually high), it allows whatever values to go into the register. Once pin 2 is asserted low, it locks whatever value is in the register at the time. Then it requires 24 clock pulse train to shift out the 24 values one at a time. That is, it locks the 24x parallel inputs and shift them out serially.

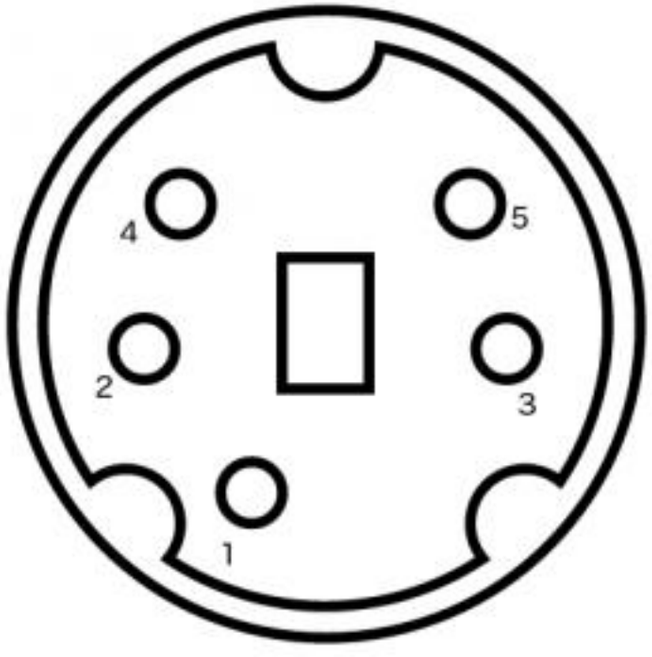
This requires three facilities in order to make it work, in addition to the Vcc and GND pins, they are:

- A trigger low periodically to lock the parallel inputs.
- During the period that the trigger is asserted low, we must generate 24x clock pulses to shift out the 24x values.
- A pin to read this 24x bits of data and arrange them in the a desired order and format.

The Hempstick 4S uses an MCU built-in Timer-Clock TIOA2 to generate the periodical trigger pulses at the rate of 1KHz. During each of the 1ms trigger period, it uses the MCU internal SSC module to generate 24 clock pulses at the rate of at least 500KHz to shift out the 24 data bits.

To connect to a TMStick, you must connect the EXT2-9 pin to EXT2-8 and then connect the combined pin to TMStick's pin 2. You must also connect the TMStick pin 3 & 4 according to the table above. In addition, you must also connect the 3.3V Vcc and GND pins. DO NOT CONNECT 5V to the TMStick! You might damage the MCU by connecting 5V to the TMStick, although the TMStick will be fine with it (TMStick would out put 5V!).

TMStick Pin Out



TMStick Color Code for Cougar

Pin	Color
1	Brown
2	Red
3	Orange
4	Yellow
5	Green

TMStick Color Code for Warthog

Pin	Color
1	Black
2	Brown
3	Red
4	Orange
5	Yellow

Please note that the color code of the wires listed here is decoded from my sample of Cougars and Warthogs. I cannot guarantee that all Cougars and Warthogs are wired this way. You use this color code at your own risk!

Button Pin Assignment

Please note that when TMStick is activated, it occupies button 0 to 23 positions. By default, Hempstick 4S has the TMStick activated.

Since Windows' DirectInput USB driver is only capable of handling up to 32 buttons, although USB is capable of up to 128 buttons, you are left with 8 additional buttons to assign.

Position in Windows	Position in Hempstick Internally	Cougar Stick Function	Warthog Stick Function
1	0	Trigger 1st Stage	Trigger 1st Stage
19	1	N/C	CMS Push
20	2	N/C	N/C
2	3	WPN/REL	WPN/REL
3	4	Nose Wheel	Nose Wheel
4	5	Pinky Shift	Pinky Shift
5	6	Master Mode Control	Master Mode Control
6	7	Trigger 2nd Stage	Trigger 2nd Stage
HAT N	8	Trim Up	Trim Up
HAT E	9	Trim RWD	Trim RWD
HAT S	10	Trim Dn	Trim Dn
HAT W	11	Trim LWD	Trim LWD
7	12	TMS Up	TMS Up
8	13	TMS RWD	TMS RWD
9	14	TMS Dn	TMS Dn
10	15	TMS LWD	TMS LWD
11	16	DMS Up	DMS Up
12	17	DMS RWD	DMS RWD
13	18	DMS Dn	DMS Dn
14	19	DMS LWD	DMS LWD
15	20	CMS Up	CMS Up
16	21	CMS RWD	CMS RWD
17	22	CMS Dn	CMS Dn
18	23	CMS LWD	CMS LWD

When the TMStick is activated, here is the list of the buttons assignments.

Section 3

Hempstick Due Pin Assignments

The MCU used in Arduino Due/X is a SAM3X8H. It is no different from any other more recent ARM chips from Atmel; they are all 3.3V devices. So, as usual, do not wire 5V supply to your sensors, unless you can limit the output of the sensors to 3.3V max. output.

ADC Pin Assignment

PIN	Function	Description
A7	X	ADC Channel 0
A6	Y	ADC Channel 1
A5	Z	ADC Channel 2
A4	Rx	ADC Channel 3
A3	Ry	ADC Channel 4
A2	Rz	ADC Channel 5
A1	Slider	ADC Channel 6
A0	Dial	ADC Channel 7

TMStick Pin Assignment

TMStick Pin Assignment

PIN	TMStick Pin	Function	Description
	1	Vcc	
62 (A8)	2	Serial-Parallel	RF
64 (A10)	3	Clock	RK
63 (A9)	4	Data	RD
	5	GND	
11 (Digital 11)		Clock	TIOA8, connect to A8, TMStick Pin 2

Button Pin Assignment

The default Hempstick Due has the TMSTick activated. So, the same as the pre-configured Hempstick 4S, the first 24 buttons are assigned to the TMStick. For the button assignment, please see Buttons Pin Assignment in Section 2.

Again, since Windows DirectInput only supports up to 32 buttons for USB controllers, we are left with 8 buttons. Here are their pin assignments.

PIN	Position in Windows	Position in Hempstick Internally
22	18	17
23	19	18
24	20	19
25	21	20
26	22	21
27	23	22
28	24	23

Chapter 4

Customization

Most of the customization of Hempstick are done through either configuration header files or some .c files that contains only data (no code).

In this chapter, we will discuss some common customizations. But we will not discuss code changes.

Top Level USB Customizations

Configuration File Sets

The directory src/config/ contains the active configuration file sets. The compilation will use this set of files.

However, there are sub-directories under the src/config/ directory that you will not see in the Atmel Studio projects. Do not add these directories or files into the project.

These are the sets of configurations file sets for certain Hempstick configurations. Each directory contains one particular set. For instance, src/config/Cougar/ contains the configuration files somebody customized to work for Cougar, in this particular case, I did.

If you just want to use a pre-configured set, copy all the files from the directory into src/config and overwrite the existing ones. Then compile burn the firmware.

If you are going to customize Hempstick, I recommend you find a set of configuration files closest to what you want and then customize that one.

The active configuration file set that come with Hempstick whenever you check out the newest version will be whatever I happen to be using last.

Change Manufacturer & Product Name

The simplest and most visible change the users want would most likely be to put their Call Sign on their custom USB controller.

There it is.

Open src/confg/conf_usb.h. Scroll down and find the following two lines.

```
#define  USB_DEVICE_MANUFACTURE_NAME      "Hempstead"

#define  USB_DEVICE_PRODUCT_NAME          "Hempstead Joystick Controller 4S"
```

Change them to whatever you like. But, please, stay in 256 character length!

Change USB Vendor ID & Product ID

Alright, for the USB uninitiated, the Vendor and Product IDs require a little explanation.

Every USB device comes with a Vendor Id & Product Id. When a USB device is plugged in to a computer, the USB Hub on the computer notifies the OS of the plug-in event. The OS then interrogates the device for its USB descriptor, which describes what the device contains what function. The OS then uses this information to find and load the appropriate driver (if any) to communicate with the device.

In the USB descriptor, there may be a USB device class number. If that exists, the OS “should” use the built in class driver for the device. For our purpose, the Hempstick is always a USB Human Interface Device (HID) class device and a Joystick sub-class. The Windows (and other OSes) will then use a built-in USB HID Joystick driver for Hempstick.

If there is no USB class indicated in the USB descriptor, then the VID & PID are used to search the OS for appropriate driver. On a Windows OS, this is the collection of the .inf file installed with the driver. A .inf file contains, along with other information, the VID & PID, as well as the name of the driver .dll file. Once a .inf file is located, the driver is loaded into the OS kernel to start communicating with the USB device.

How do you get a VID? Well, the USB consortium controls the distribution of the VIDs. You either join as a member of consortium, or you pay about USD $2,000/year for a non-member VID. Under each VID, you may assign whatever PIDs you wish to your products.

Even though the OS does not use the VID & PID to locate the appropriate driver for HID devices, VID & PID are still required to be programmed in and reported by the device via the USB descriptor.

Thus, some manufacturers use this VID to exclude the support of other vendor’s products in their drivers.

You cannot just program your Hempstick to any VID/PID pair!!! The OS will remember it and may cause collision with other vendors’ VID/PID!

But, it’s your computer, as long as you are careful to avoid the collision, do you care? Say, you use CH Rudder’s VID/PID. As long as you don’t plug in CH rudder in your computer, you will not have collision.

Under a USB HID class, there are sub-classes of devices, like mouse, keyboard, etc. Hempstick is a Joystick sub-class device. A USB HID class driver is a generic class driver. In the USB

interrogation process, the Hempstick must also submit a USB Joystick HID report descriptor. This HID report descriptor describes what which button is at which byte, which bit, which axis is at byte what, and how long. We will come back to this later in more advanced customization. Suffice it to say that this HID report descriptor is how we define how many buttons, and how many axes there are in the USB joystick we are plugging in.

Back to VID/PID.

If you use other manufacturer's VID/PID. What happens?

Well, remember that since the Hempstick is an HID Joystick sub-class device, the Windows generic HID Joystick driver is used. And the USB HID report descriptor the Hempstick reports determines how many buttons and how many axes there are. Since the driver is a generic one, the what so called "driver" from your vendor, like ThrustMaster, CH, or Saitek are not really "driver driver", they are just user land applications, not a kernel land "driver driver", and they must go through the generic driver to find out the capability of the device plugged in. That is, they get whatever we report in the HID report descriptor.

So, say, if you program your Hempstick to be a T.16000M joystick. What happens? Well, if they write their "driver" correctly, then all the buttons & axes will work right. Because a SIM does not care whether it's a T.16000M joystick or not, it only care whether it's a DirectInput device. And the SIM queries the OS via DirectInput API to find what joysticks are available, and what capability each joystick has, it gets what our HID report descriptor says too.

However, each vendor's "driver" might be programmed to know specifics of a particular device, based on the VID/PID. So, inside their "driver", they might not be able to "refer" to the Rx axis, because the particular device does not have it. For instance, a T. 16000M joystick does not have the Rx axis. So, inside TARGET, you cannot program the Rx axis. But, inside your SIM, you can still use the Rx axis.

Choose your VID/PID carefully to fit your needs.

To change the VID/PID, find the src/conf/conf_usb.h again, and find the following lines.

```
#define  USB_DEVICE_VENDOR_ID          0x44F

#define  USB_DEVICE_PRODUCT_ID         0xB10A

#define  USB_DEVICE_MAJOR_VERSION      8

#define  USB_DEVICE_MINOR_VERSION      0
```

Change them to what you like.

While we are at it, let's talk about the major/minor numbers.

Each time you change the USB descriptor, or HID report descriptor, the Windows OS remember that, associate with the VID/PID/major/minor numbers. So, if you change your descriptors, it would be wise to also “bump up” the major/minor numbers. Otherwise, the Windows OS might continue to use the old descriptors from its internal cache and your newly updated Hempstick will not work well with the old descriptors the OS assumes it’s using!

Now, is it legal to use others’ VID/PID? I am not a lawyer, but what I understand is this.

The USB Consortium does not own numbers! They own their trademarks. One of that is the USB logo. To qualify using the USB logo and trademark, your devices must pass a USB test suite. One of the test criteria is the VID/PID. You must program it to use your own VID!

But, as a private citizen, do you care whether your own custom Hempstick carries a USB logo, or claim USB compliant? I guess you do not.

Moreover, USB developers routinely use others’ VIDs. Otherwise, you may not start your development before you obtain a VID. That’s just not good for business.

Change The Buttons & Axes Assignment

The Hempstick comes with a default buttons & axes assignments for each board. By default, it comes with 64 buttons & 8 axes. And each different board has different pin assignments mapped to each USB report position (i.e. USB HID buttons & axes) because each board has different layout of the pins.

I try to make the default configuration as general as possible, but there is no such thing as one size fits all in embedded programming. You will need to be able to remap these. Hempstick makes it easy for you to do that. There are two files to change. They are:

- src/config/conf_hempstead.h
- src/config/conf_hempstead.c

I know, I know... naming the files with your own online handle... EGOTISTIC BARSTARD!!! Programmers are not very inventive in naming things. We run out of names all the time. I originally put conf_hempstead.h and .c there as a place holder before I came up with the name Hempstick, and intended to change the conf_hempstead.h .c files to match the name of the controller, but the name stuck because after the fact it’s a bit difficult to change when the IDE, VisualStudio-based Atmel Studio does not have a good refactor tool like Eclipse has.... It’s commonly referred to as Design Inertia. ;-(If you do manage to change them to conf_hempstick.h .c safely, please let me know, I will gladly take

your changes. But I would still need to modify the documents to match. *sigh*

The Number of PINs configured and Buttons

Find the following lines in the src/config/hempstead.h.

```
#ifdef CONF_BOARD_SAM4S_XPLAIN_PRO

#       define CONF_NUM_PINS
        15

#       define CONF_TOTAL_NUM_BUTTONS
        32

#       define LED0_GPIO
        PIO_PC23_IDX

#       define CONF_SSC_CLOCK_SOURCE_ID
        ID_TC4

#       define CONF_SSC_CLOCK_TC
        TC1

#       define CONF_SSC_CLOCK_CHANNEL
        1

#elif defined(CONF_BOARD_ARDUINO_DUE)

#       define CONF_NUM_PINS
        22

#       define CONF_TOTAL_NUM_BUTTONS
        28

#       define CONF_SSC_CLOCK_SOURCE_ID
        ID_TC8

#       define CONF_SSC_CLOCK_TC
        TC2

#       define CONF_SSC_CLOCK_CHANNEL
        2

#endif
```

Note the lines marked in red. You only need to concern yourself with the section that corresponding to the board you are using.

The macro constant CONF_NUM_PINS must match exactly the number of entries in the following section in the src/config/conf_hempstead.c.

```
hw_pin_configuration_table g_hw_pin_conf_table = {

        .mutex = NULL,

        .pin = {

                {.pin = PIO_PC23_IDX, .conf =
HW_PIN_ENABLE_MASK, .mode = (PIO_TYPE_PIO_OUTPUT_1 | PIO_DEFAULT)},



                {.pin = PIO_PA0_IDX, .conf = HW_PIN_ENABLE_MASK, .mode =
PIO_PERIPH_B},
```

```
        ...
                {.pin = PIO_PA24_IDX, .conf = HW_PIN_ENABLE_MASK, .mode =
(PIO_TYPE_PIO_INPUT | PIO_PULLUP | PIO_DEBOUNCE)},


                },
        };
```

The number of {.pin ...} entries must match what is defined in CONF_NUM_PINS!!! If you count it wrong, the whole thing either come crashing down or some entries will not be read and the pins will not be configured correct! I might change this “inconvenience” in the future when I implement the feature of dynamic configuration at runtime via a USB thumb stick.

Configure MCU Pins as Buttons

This table of data are read at startup of the Hempstick configure the MCU pins. Each pin you wish to use must be configured because each pin of the Atmel MCUs are “multiplexed” to have multiple possible functions to save the number of pins that need to be brought out from the silicon die to outside of the IC package. If you don’t configure a pin, then it would use the default configuration and that is highly MCU dependent.

For instance, on the SAM4S MCU with 144 leads, the pin PIO_PA24 could be used for General Purpose IO (PIO), or it could be used for the following functions.

- RTS1
- PWMH1
- A20
- PIOD0
- GPIO

Meaning, that particular pin could be used for either Request To Send for Serial Port 1, or PWM High Side channel 1, or Address line #20, or PIO D0, or GPIO. But not all at once. It’s firmware controllable.

In addition, each pin when configured in a particular mode, might have several different options. What we are more concerned here is when you configure it as a GPIO pin for button inputs.

Take a look at the PIO_PA24 entry. When a pin is configured as GPIO pin, we first put the .conf=HW_PIN_ENABLE_MASK to tell Hempstick to enable that particular pin. Then there is the .mode.

The .mode tells the Hempstick to switch this pin to PIO mode, tell it to configure it as input, turn on the built-in pull-up resistor, and

then finally turn on the hardware debouncer. All button pins must be configured this way. Except the pull up resistor.

Some newer Atmel MCUs not only come with pull up resistors for GPIO pins, they also come with pull down resistors. Usually, you would want the pull up resistor to be turn on instead of pull down. It's common to use pull up resistors on IO pins. This is often referred to as negative logic. For buttons, you must either pull up or pull down, if you don't, the pin is called floating. A floating pin's status is undetermined. That is not good. For unused pins, you usually want to "ground" them to make them into a known state. If you don't use a pin and you don't pull up nor pull down, you cannot reliably read them.

The Hempstick uses the conventional negative logic for buttons. Meaning, if the pin has 3.3V, it's a logic 0. If the pin has 0V, it's a logic 1. Then the firmware reverses it to report to USB.

So, if you turn on the pull up resistor of a GPIO pin. Then you need to connect that pin to one side of the button, and the other side of the button to ground. This way, the pin is internally connected to the 3.3V by the MCU. So if the button is not pressed, the pin is 3.3V, and the Hempstick reports it as logic 0 (button not pressed) to USB.

If you turn on the pull down resistor, then you need to connect the other side of the button to 3.3V. This way, when the button is not pressed, the pin is 0V, and the Hempstick reports it as logic 1 to the USB. This is used in the usually on, press to break situation.

But which pin should be configured for buttons? You need to read the board's User's Guide. It will give you tables of pins names.

The above is for configuring each MCU pin for the purpose of GPIO pin as a button. We still need to tell it which pin maps to which button bit in the USB report.

Assign MCU Pins to USB Button Bits

A configured pin is just a configured pin on MCU. It is useless to us unless we tell it which pins are mapped to which USB report as button pressed.

Each of the button status in the USB report is one bit, either 1 or 0. '1' means button pressed, and '0' means button not pressed.

You may map any pin to any USB button (just don't map multiple pins to the same USB button!). Here's how you do it.

In the same file, src/config/config_hempstead.c, find the following section.

```
rtos_button_data_t g_rtos_button_data = {

        .data = NULL,
```

```
        .num_button = 0,

        .mutex = NULL,

        .rtos_internal_task_semaphore = NULL,

        .rtos_task_semaphore = NULL,

#ifdef ID_PIOA

        .ports[0].button_conf[0].flags =  0x0000,

        .ports[0].button_conf[1].flags =  0x0000,

                .ports[0].button_conf[24].flags =
RTOS_BUTTON_PIN_ENABLED_MASK, .ports[0].button_conf[24].data_position = 30,

        .ports[0].button_conf[25].flags =
RTOS_BUTTON_PIN_ENABLED_MASK, .ports[0].button_conf[25].data_position = 31,

        ...
```

On the Atmel SAM serious of ARM chips, the pins are organized in ports. Each port contains maximum of 32 pins. They are usually named PIO_A, PIO_B, PIO_C, ... That's why the pins are usually named PAxxx namely Port A pin xxx.

In the data structure above, you first enable the pin for mapping, by specifying RTOS_BUTTON_PIN_ENABLED_MASK, then, you tell it which USB button you wish to use for which pin, by specifying something like this, .ports[0].button_conf[24].data_position = 30. Here, port A must use .ports[0], port B uses .ports[1],... etc. Then you tell it the pin number with .button_conf[xxx], and .data_position = yyy, where .xxx is always the pin number, yyy should be the USB button number.

Repeat this for all USB buttons you wish to map. If you configure a pin, but does not provide the mapping. That's ok, no harm done, but you won't see the button pressed in the USB reports send to your computer. It's just that when you press that button, the Hempstick will spend the time to process it, but will never report it to the computer. Wasteful, but it's safe.

On the other hand, if you don't configure a pin, yet you provide the mapping to USB button, you would be reporting BS to the computer. In particular, if you configure a pin to other mode, say PWM, the behavior is undefined. When the programmer tells you the behavior is undefined, it usually means... don't do it, I don't know what will happen!

What happens when you configure a pin as GPIO and as input, but you connect it to some external signal instead of a button? Say, you have an external circuit that when pressed, it automatically sends out 5 on-off sequences? Well, you get 5 USB report that Chun-Li just executed a 5 bunch combo. Hempstick does not care what the signal source is. You can connect an input pin to a clock signal if you wish (as long as the signal passes the debouncer, or you turn off the debouncer for that pin). Then you would get a periodical USB button press reports. Note that, a full speed USB can only get 1,000 reports per second. So, if your

clock signal exceeds 1KHz, then you would still get 1K USB button reports, not necessarily on-off-on-off.... it could be on-on-on-off-off-on-.... Each status of the button will be the current status of the button when each USB report is requested by the computer. Don't do that.

Configure the ADC Pins

Usually, if a pin has ADC mode on it, that's the default mode when the MCU boots up. That's the case with Atmel ARM MCUs. But there is no guarantee! If you know your board, and your MCU has this behavior, then there is no need to configure the ADC pins. But if you are not sure, make sure you configure the ADC pins.

In the file src/config/conf_hempstead.h, find the following entry.

```
#define MAX_ADC_CHANNEL 16
```

That's the maximum ADC channel data structure entries the Hempstick will allocate. Make sure this number is larger than the total ADC channels you wish to use. Setting this larger than needed is ok. It just waste a bit of memory space. No harm done. Just make sure this number is not larger than the number of channels the MCU has.

If you need to explicitly configure an MCU pin to switch on the ADC mode, add similar entry in the src/config/conf_hempstead.c.

```
{.pin = PIO_PA2_IDX, .conf = .conf = PIN_ENABLE_MASK, .mode =
(PIO_TYPE_PIO_PERIPH_B | PIO_INPUT | PIO_DEFAULT)}, // ADC channel 0
```

Make sure you find the PERIPH_X mode in the MCU specification document (don't worry, you don't need to read the thousands of pages of spec sheet per MCU. There is usually a table somewhere in the beginning of the spec sheet). Sometimes they are peripheral B, sometimes A, sometimes they are called the extra mode and is automatically turned on at bootup time if you don't configure it in the case of SAM3X MCU on the Arduino Due/X board.

Configure the ADC Channel To USB Axes Mapping

Again, without specifying the mapping from ADC channels to USB axes, you get no USB report for the values.

There are two places that Hempstick needs your help to do this.

Go to src/config/conf_usb.h and find the following entries.

```
#define HID_JOYSTICK_REPORT_ADC_BYTE_OFFSET 8

#define HID_JOYSTICK_REPORT_TOTAL_NUM_ADC_AXES 8
```

The HID_JOYSTICK_REPORT_TOTAL_NUM_ADC_AXES tells the Hempstick how many total axes you specified in the USB HID report descriptor. The

HID_JOYSTICK_REPORT_ADC_BYTE_OFFSET tells the Hempstick where the byte offset of the axes report should start in the USB HID report should be.

I am sorry, I don't wish to write a USB descriptor parser just to find out these two pieces of information while you could easily count it and tell me!

If you don't know how to count that, then don't change the USB HID report descriptor! Leave it be! If you do know how to change the USB HID report descriptor, then you know how to count those , then help me and give that information.

To enable the ADC channels and provide the ADC channels to USB mapping, go the the file src/config/conf_hempstick.c and find the following entries.

```
rtos_adc_data_type g_adc_data = {

        .data = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

#ifdef CONF_BOARD_SAM4S_XPLAIN_PRO

        .channel_flags = {ADC_CHANNEL_ENABLE_MASK, 0, 0, 0,
ADC_CHANNEL_ENABLE_MASK, ADC_CHANNEL_ENABLE_MASK, 0,
ADC_CHANNEL_ENABLE_MASK, ADC_CHANNEL_ENABLE_MASK,
ADC_CHANNEL_ENABLE_MASK, 0, 0, 0, ADC_CHANNEL_ENABLE_MASK,
ADC_CHANNEL_ENABLE_MASK, 0},

#elif defined(CONF_BOARD_ARDUINO_DUE)

        .channel_flags = {0, ADC_CHANNEL_ENABLE_MASK,
ADC_CHANNEL_ENABLE_MASK, ADC_CHANNEL_ENABLE_MASK, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0},

#endif

        .channel_mapping = {0, UINT8_MAX,  UINT8_MAX,  UINT8_MAX, 1, 2,
UINT8_MAX, 3, 4, 5,  UINT8_MAX,  UINT8_MAX,  UINT8_MAX, 6, 7,
UINT8_MAX},
```

The .channel_flags = {....} is the channel enable flags for each ADC channel. The MCU might have one ADC hardware, but each can multiplex to multiple channels. You must provide exactly the number of entries in the table as is defined in

```
#define MAX_ADC_CHANNEL                16
```

The entries in the table starts from index 0 channel of the MCU's ADC module. To enable ADC 0 channel, put ADC_CHANNEL_ENABLE_MASK in the index 0 position, and so on so forth.

To specify which ADC channel maps to which USB axis, find the .channel_mapping table, and put in the USB axis number in each ADC channel position. For any ADC channel you do not wish to map, put UNIT8_MAX at that position.

Why do we need to provide ADC channel to USB axis mapping? Well, some boards are not designed specifically for SIM! For instance, the Atmel XPLAINED Pro boards are designed for developers as evaluation boards so they can quickly start writing

the firmware while their hardware designer are still designing and debugging the board (that's why they are cheap, because Atmel sells MCUs and they want you to develop using their chips easy and hook you on so when you go production, they make the big bucks in volume, instead of making small bucks by selling you thousand of dollars development boards like in the 1990s). Because of this, these boards are not exactly SIM friendly. They may not route out all the ADC/GPIO pins out on the board, and they may not collect all the ADC pins in one convenient place. And worst of all, they might have some peripherals like buttons, LEDs, POTs, LCDs, OLEDs, SRAMs etc. on board to demonstrate the capabilities of the MCU. So, some of these pins are already "used." You shall not have them.

That's the price to pay for the cheap evaluation boards.

So, to get 8 axes for Windows, you might have to use ADC0, ADC5, ADC7, ADC10, ADC.... and the pins maybe all over the board. If you want the convenience of all the ADC pins and digital pins are routed out in one place on the board, use the Arduino Due/X board. But the Arduino boards do not come with the embedded debugger like the SAM4S XPLAINED Pro board. You wold have to buy a hardware debugger (USD $99). And Arduino Due/X also comes with Arduino bootloader, which requires some trick to get it to be "burned" by the Atmel Studio IDE.

That, that, that, that, that, that's all folx.