Project Tips & Guidelines

~ Wyatt Lindquist

If you are stuck on a part of the project, some of the details in here may help. Note that all examples, and any suggestions, are merely that. They do not have to be implemented in the way specified; further, in many cases, the examples would not be sufficient in a complete project. They should just spark your creativity and help you move along if you are having trouble. If you are already doing it differently, it is very likely that that is perfectly fine as well. The project is intentionally broad to allow you to come up with your own protocol and implementation.

Part One – GPIO Drivers

XINU already has a *gpio.h* header, but it can't be used as-is. Notice that the comments in that header reference "Cisco" buttons and LEDs. Further, if you *grep* the Xinu directory, you will find no references to *gpioLEDOn* or the other functions. This is because that header was actually defined for another platform Xinu runs on – a MIPS router! As such, you will notice the resources you find online discussing BBB GPIO use different base addresses and register orders. Still, the existing header can be useful for understanding how GPIO works.

Looking at BBB/GPIO resources, you will notice that only some of the expansion headers on your board are tied to GPIO pins, and they are not in a specific order. Only some of the internal GPIO pins are tied to the expansion headers. To control the GPIO, you must know the actual pin number, not a number related to the expansion header.

Further, since the BBB is 32-bit, rather than setting a single pin per memory mapped address, multiple pins are grouped together into *ports*, which are then mapped to a single address, per port. Each bit in one of these addresses controls a single GPIO pin. So, in C, you must utilize bit manipulation to set individual pins.

The GPIO address space has a lot of registers, but the ones of most interest are *Output Enable, Data In,* and *Data out*. To access these registers, you must add the respective register offset to the GPIO base register for the needed port.

The *Output Enable* register specifies the "direction" of a set of pins. This is how you set a pin to be either input or output. The *Data In* register holds the current values of each pin with an IN direction on a port. Note that the values are undefined for pins set as OUT. The *Data Out* register is written to to set the output values of a pin on a port. Similar to *Data In*, this register is only defined for OUT pins.

Notice that the above discussion is exclusive to *digital* inputs and outputs. To get input from an *analog* device, some of the expansion header pins must be specially configured. On the BBB pinout, some of the headers are labeled AIN0, AIN1, etc. These are pins that are dedicated to the ADC.

To control the ADC, a different base address will be used, with a programming method independent from the digital one. As the BBB ADC is an integrated part of the processor, the best place to get information on this would be a datasheet for the AM335x, the BBB MPU. Pay special attention to chapter 9, on control modules and pad control, and chapter 12 (the ADC is integrated with the touchscreen controller).

Note: The BBB ADC supports a voltage range up to **1.8V**, around half of the actual BBB voltage (**3.3V/5V)**. DO NOT CONNECT ANY VOLTAGE HIGHER THAN 1.8V to the BBB ADC. This includes the other GPIO outputs and the power pins. Doing so may damage your ADC. When connecting a sensor, make sure (in its datasheet) that the output is within range. Some chips may take in a larger voltage, like 5V, and only output an analog range of, say, 0V – 1V.

Part Two – High-Level I/O Interface

Once your lower-half drivers are implemented, you can start thinking of your high-level I/O interface. This should be a new set of functions that, when used, will use the low-level driver functions to accomplish the task in a way that is more appropriate for the externalized IoT environment.

These functions should be callable from standard Xinu code. Keep in mind that for the later parts of the project, this interface, or a new one, will need to be callable remotely (over Ethernet), as a form of RPC, for example. In this method, a packet will arrive specifying the requested function, along with its arguments. Xinu could then send back a response packet. If your functions involve concepts such as handlers, special care would need to be taken to make sure the data gets back to the original remote caller.

Part Three – The DDL File, Parser, and Generator

Your DDL file will describe the connections for a SPECIFIC configuration of your BBB/sensors combo. This configuration will be known before hand, so there is no need to consider auto-detection / hot-swapping. You DDL file can be XML, JSON, or a format of your choosing, as long as it is human-readable.

The next step is to *parse* your DDL into something that can be used by Xinu: driver code. Note that the DDL will likely also be parsed into something for the edge node (more on this later).

As an example, consider the DDL on the next page. Your DDL format does not have to be the same; in fact, this example is lacking in many aspects. In the example, I define “devices”, which are higher-level abstractions of a combination of pins. Each pin then contains a number of attributes describing its functionality. Note that the attributes describe both high-level functionality (the name/id) and low-level functionality (the pin number and logic level).

my-example-config.ddl

```
<ddl>
     <device>
          <name>LED_rgb</name>
          <pin>
               <id>gnd_r</id>
               <index>0</index>
               <mode>out</mode>
               <level>high</level>
          </pin>
          <pin>
               <id>gnd_g</id>
               <index>1</index>
               <mode>out</mode>
               <level>high</level>
          </pin>
          <pin>
               <id>gnd_b</id>
               <index>3</index>
               <mode>out</mode>
               <level>high</level>
          </pin>
     </device>
     <device>
          <name>button</name>
          <pin>
               <id>sense</id>
               <index>7</index>
               <mode>in</mode>
               <level>low</level>
          </pin>
     </device>
</ddl>
```

You may write your parser in any language of your choosing (it will be ran before Xinu is compiled). I would suggest choosing a higher-level language that already has a parsing library for something like XML or JSON.

Once you can parse your DDL, you can use that information to generate some part of the driver. Consider the above example. In this case, I decided that my high-level interface will simply support reading and writing individual pins (based on a device id, which is also generated by the parser). Therefore, I declare a *skeleton* of my two functions:

Note: This is just an example; this is not the specific part of the driver you *must* generate. You may for example, just generate a complete .c file. Overall, you may do it in any way of your choosing – just remember the instructions state it should generate "as much of the code of the device drivers as possible", as in, you shouldn't be manually creating code for individual device configurations.

pio.c

```
bool pread(int dev_id)
{
	switch (dev_id)
	{
		#include <ddl_io_in.inc>
		default:
			assert(false);
	}
}

bool pwrite(int dev_id, bool data)
{
	switch (dev_id)
	{
		#include <ddl_io_out.inc>
		default:
			break;
	}
}
```

Notice how each function is simply a bare minimum, with the implementation to be defined externally. When Xinu is compiled, it takes the output generated from the DDL parser, and #include's (copy-pastes) it into my high-level device code.

On the next page, I show the example generated output *dd_io_out.inc*. Notice how it is just a portion of valid C code that completes the switch statement when it is included.

ddl_io_out.inc

```
case 0:
     gpio_write(0, data);
          break;
case 1:
     gpio_write(1, data);
          break;
case 2:
     gpio_write(3, data);
          break;
```

Once the file is included when Xinu is compiled, the high-level driver functions *pread* and *pwrite* are completed and operational for that specific DDL configuration.

Note: To generate the DDL output before compilation, look into adding a new Makefile rule to the Xinu file *compile/Makefile*. This rule can depend on the input DDL and trigger you parser binary to create the output. By making the existing rule depend on this new rule, the DDL will always be parsed before compilation.

Part Four – Setting up the Edge & Cloud

Once your Xinu modifications are complete, you can begin setting up your IoT configuration. The most basic setup will consist of: BBB(s), a computer, and a router. In this setup, both the edge and the cloud would be running as separate applications on the computer, connected to the BBB(s) with the router. Note that the router is used solely for connecting the BBB(s) to your computer; it does not act as the edge.

The edge and the cloud should continue abstracting the device details, making the low-level functionality simple and intuitive to control at the user application level, while still showcasing your new high-level functions.

The edge and cloud may also require some output from the parsed DDL to function and communicate with its nodes. Consider my above example; in my system, my Xinu instances only understand pin numbers, and my cloud only understands string names. I can use my edge, with information from the DDL, to facilitate this communication.

On the next page, I show the output that my edge will receive from the DDL.

ddl_edge_glue.inc

```
0 : LED_rgb.gnd_r
1 : LED_rgb.gnd_g
2 : LED_rgb.gnd_b
3 : button.sense
```

Here, the edge knows both the device ID and the name of each pin. This will allow it to translate the communications that originate between the cloud and the nodes. Once again, note that the edge does not have to work this way - this is merely an example. The main purpose of the edge is to facilitate the communication between devices and the cloud.

Finally, the cloud will handle interaction with the entire system. This is the layer in which your demo application would interact with. For example, using the above DDL and support, the demo may provide three buttons to control the RGB LED and a line of text showing the status of the button. Interacting with the RGB LED may cause a packet to be sent from the cloud to the edge, which the determines which BBB/pin the LED component is located on, finally sending it to the appropriate board with the relevant state information. When the button is pressed, the edge may be polling or using events to notice this change, upon which a packet could be sent to the cloud, updating the demo app text.

In your real demo application, the app should showcase the power/flexibility/generalness of your high-level Xinu interface. For example, you may be able to cause an LED to blink at a specified interval, or use a button to automatically control the state of an LED on a different board. You should focus on creating a unique/creative interface that works well for your devices in an IoT context, as it is the largest component of the project, upon which most of the rest depends.