A Guide to PygmyOS

Warren D Greenway

September 16, 2012

Pygmy OS is a simple, small, yet scalable operating system for embedded systems. Its primary goals are to be clean, simple, and efficient. Pygmy is written in boiler-plate C for the sake of efficiency and simplicity. Pygmy also follows simple naming conventions and allows the OS to be integrated with the end application in a staged manner, from simply wrapping the hardware in a user friendly set of APIs, to providing advanced tasking, clock, utility, command processing, file system, graphical and messaging services.

At the end of this document you will find an API reference. In the following chapters Ill be including a basic overview of the API as it applies to the examples presented. Currently, I have no intention of including source code with these tutorials. From a practical perspective, you dont learn when you copy and paste. You need to drill the basics in. With that thought in mind, however, Ill be relying on brevity to avoid applying too much load to your patience and typing skills.

Lets start with a basic description of your main.c source file.

#include “pygmy_profile.h”
void main(void) { while(1){ ; } }

So, lets start at the top. For each Pygmy project there should be a file named pygmy_profile.h that contains the profile information for your specific platform, or hardware configuration. The contents of pygmy_profile.h will look something like this:

```
#pragma once

#include "pygmy_sys.h"
#include "pygmy_port.h"
#include "pygmy_com.h"
#include "pygmy_audio.h"
#include "pygmy_adc.h"
#include "pygmy_file.h"
#include "pygmy_lcd.h"
#include "pygmy_gui.h"
#include "pygmy_nvic.h"
#include "pygmy_rtc.h"
```

```
#include "pygmy_string.h"
#include "pygmy_web.h"

#include "profiles/pygmy_nebula.h"
```

The first line is for the compilers pre-processor only, it keeps the header file pygmy_profile.h from being included multiple times. The next group is the Pygmy API includes. The only file included in the API sources is pygmy_profile.h, which means it is critical to your project, and that every API used must be listed. It may be useful to note that Pygmy OS is designed to be compiled with size optimization and including all APIs will not cause a decrease in performance or an increase in code size. In light of this fact, it is generally a good idea to include all the APIs to avoid complications with missing resources as your project evolves.

The last line is the most interesting. In this case the included file is the standard hardware definition for the the Pygmy Nebula board. If a "shield" or "backpack" module or modules are to be used with the main board, they will be included just below the primary board include file. All Pygmy profiles for modules and platforms (main boards) are stored in the folder pygmy/profiles/. The file pygmy_profile.h must be in the same folder as main.c. One other note on pygmy_profile.h is that it can be used as the header file for main.c.

Lets get back to our main.c and finish our analysis. The main function is void and void. No parameters are passed to main or returned by main. Dropping out of main will return to the startup function and an endless loop. Despite this fact, it is still recommended to insert an infinite loop at the end of main. Generally nothing will be executed in the loop.

Now that we have gone over a basic main source file, we look at a few examples.

Chapter 2 – Getting the Blink:

Goal: Blink an LED
Platform: Pygmy Nebula
API description:
pygmy_port
void pinConfig(u8 ucPin, u8 ucMode)
void pinSet(u8 ucPin, u8 ucState)
void pinGet(u8 ucPin)
pygmy_sys
u16 sysInit(void)
u16 newSimpleTask(u8 *ucName, u32 ulTimer, PYGMYFUNCPTR EventHandler)

Its a good idea to have balance before taking a step. Its a good idea to learn to walk before you try to run. In line with this reasoning, well start by turning an LED on, LED0 to be specific. LED0 and LED1 are the two user LEDs on

the Pygmy Nebula Board. By setting the pin to OUTPUT and setting the pin HIGH, the LED will power on.

```
#include “pygmy_profile.h”

void main( void )
{
pinConfig( LED0, OUT );
pinSet( LED0, HIGH );

while( 1 ){ ; }
}
```

Loading and booting this simple program will light up LED0 on the board and wait in the loop. As you can see, the program is incredibly simple. Making the LED blink will be our next task. There are many ways to accomplish this, all reasonably simple. In the example we will use the tasking system to blink the LED. This approach will be much like driving a tack with a sledge hammer. There is a reason for using this specific approach, we will be demonstrating with this simple example how to use the timing and tasking systems. Once you start coding for Pygmy, youll quickly find the tasking system to be a powerful tool to solve a wide variety of problems.

```
#include “pygmy_profile.h”

void blink( void );

void main( void )
{
sysInit( );
pinConfig( LED0, OUT );
newSimpleTask( “blink”, 100, blink );

while( 1 ){;}
}

void blink( void )
{
if( pinGet( LED0 ) ){
pinSet( LED0, LOW );
} else {
pinSet( LED0, HIGH );
}
}
```

Got that? You now have a task (thread) running in the background. First, sysInit() initializes the task and messaging systems. Second, pinConfig()

is called to set LED0 to output. Third, newSimpleTask() is called to spawn a thread named "blink", with an interval of 100 miliseconds; the function blink() will execute every 100 miliseconds.

When blink is executed, pinGet() returns the current state of LED0, and toggles the pin state. The task sill execute indefinitely and requires very little resources to execute. You can of course start multiple tasks. You can blink LED1 in opposition to LED0 in this way.

Ill provide another LED blinking example below that alternates the two user LEDs using two threads (yes, it obviously could be done with the one thread). Note in this case that instead of using a static variable to store the state of the LEDs, we are using pinGet(). Due to this fact, we must setPin() one of the LEDs LOW to start them out opposite to eachother. Why? Simply because the port pins default to HIGH.

```
#include "pygmy_profile.h"

void blink1( void );
void blink2( void );

void main( void )
{
sysInit( );
pinConfig( LED0, OUT );
pinConfig( LED1, OUT );
pinSet( LED1, LOW );
newSimpleTask( "blink1", 100, blink1 );
newSimpleTask( "blink2", 100, blink2 );

while( 1 ){;}
}

void blink1( void )
{
if( pinGet( LED0 ) ){
pinSet( LED0, LOW );
} else {
pinSet( LED0, HIGH );
}
}

void blink2( void )
{
if( pinGet( LED1 ) ){
pinSet( LED1, LOW );
} else {
pinSet( LED1, HIGH );
```

```
}
}
```

Chapter 2 – PWM:

Goal: Blink an LED
Platform: Pygmy Nebula
API description:
pygmy_port
u8 pinPWM(u8 ucPin, u32 uiFreq, u8 ucDutyCycle)
pygmy_sys
u16 sysInit(void)
u16 newSimpleTask(u8 *ucName, u32 ulTimer, PYGMYFUNCPTR EventHandler)

Now that you have LEDs blinking, its time to take the next step. Our next goal is to have LED0 flash instead of simply blink. By this, I mean that we will linearly increase and then decrease the light intensity. How will we accomplish this goal? Simple. We will use PWM (Pulse Width Modulation) to vary the average DC voltage.

With PWM you are basically switching the power from high to low at a given frequency for a specific percentage of the time to vary the average voltage. Fifty percent duty cycle would result in approximately fifty percent of the supply voltage at the pin being toggled.

A real description of PWM is beyond the scope of this document. However, if you plan on using PWM for partial voltage generation or hobby servo positioning, you should go read up on Wikipedia at the least.

Pygmy makes PWM simple. All you have to do to set PWM on a timer pin is: pinPWM(pin, frequency, dutycyle);. The pin will be configured to the correct mode and the peripheral clock sources configured and the timers configured automatically. The PWM will continue on the pin or pins selected until the pin mode is changed or the timer disabled. Even if every timer output is setup as PWM and actively being driven, there is no load on the CPU caused by this activity.

Before we look at sample code, it is high time to review the Pygmy pin naming convention. This is simple in arrangement and will only take a moment to review.

Dx(D0, D1, etc.) Digital IO
Ax(A0, A1, etc.) Analog Input/Digital IO
Tx(T1, T2, etc.) Timer Output/Digital IO
TAx (TA0, TA1, etc.) Timer Output/Analog Input/Digital IO

Using this convention, the pin name tells you what limitations the pin has for usage. The pin name on the board is the same name you will use in the function calls, or alternately the GPIO name for the pin (PA11 for LED0, as an example). Additional functions may exist on the pin in question, review

the ST Reference for the MCU in question for additional details if the standard behavior isnt what youre looking for.

Besides the generic IO pins, there are also special purpose pins that may also be used as general purpose IO if desired. Some examples of these special purpose pins are as follows:

COM1_TXTransmit for COM1 serial port.
COM1_RXReceive for COM1 serial port.
COM2_TXTransmit for COM2 serial port.
COM2_RXReceive for COM2 serial port.
COM3_TXTransmit for COM3 serial port.
COM3_RXReceive for COM3 serial port.
DAC1Digital to Analog Converter Channel 1
DAC2Digital to Analog Converter Channel 2
MCOMaster Clock Out

Now that we have discussed the basics of pin naming, we may continue with our code example of PWM. Please note that LED0 and LED1 are significantly different electrically in that LED0 is connected to GPIO PA11 while LED1 is connected to GPIO PA12. You will find in the reference manual for the MCU (should you have the desire to read said manual) that PA11 is connected to Timer1 Channel4, while PA12 is not connected to a timer output at all. This means that the following sample code will only work with LED0. LED1 is strictly a digital IO.

```
#include “pygmy_profile.h”

void flash( void );

void main( void )
{
sysInit( );
newSimpleTask( “flash”, 100, flash );

while( 1 ){;}
}


void flash( void )
{
static s8 i = 0, cStep = 1;

i += cStep;
if( i > 80 ){
cStep = -1;
}
```

```
if( i < 1 ){
cStep = 1;
}
pinPWM( LED0, 8000, i );
}
```

Chapter 3 – COM Ports:

Goal: Send and Receive data using COM3
Platform: Pygmy Nebula
API description:
pygmy_com
void comConfig(u8 ucPort, u8 ucProtocol, u8 ucOptions, u32 uiRate)
pygmy_sys
void streamInit(void)
u8 streamSetRXBuffer(u8 ucStream, u8 *ucBuffer, u16 uiLen)
u8 streamSetPut(u8 ucStream, void *ptrFunc)
u8 streamSetGet(u8 ucStream, void *ptrFunc)
u8 streamGetChar(u8 ucStream)
void print(u8 ucStream, u8 *ucBuffer, ...)

LEDs are all well and fine, but once youre done blinking the user LEDs every way you can imagine, youll likely want to move on to something more useful. Being able to communicate through the COM ports allows you to dump debug info, upload or download files to memory, read sensor values, or implement a full-featured command interface. The good news is that serial communication is a breeze with Pygmy.

An important note here is that the Nebula boards running the F103 and F100 series MCUs do not have native USB. What this means in practical terms is that communications through the USB port on the Nebula boards is accomplished via the serial interface. Internally, COM3 is connected to a USB transceiver IC (MCP2200) which accomplishes the translation to USB in hardware. So, for these tutorials well be focusing on the serial communication interface using COM3.

Our first foray into serial communications will cover configuring the port and sending a string. There are low level and high level interfaces available in Pygmy, but well be focused on the high level interface.

```
#include “pygmy_profile.h”

void echoCOM3( void );

void main( void )
{
streamInit();
streamSetPut( COM3, putsUSART3 );
```

```
print( COM3, “Hello World” );

while( 1 ){ ; }
}
```

Lets start near the beginning and work our way back. The first function called, streamInit() configures the stream interface used by print() to defaults. The next function, streamSetPut(), selects the interface for the selected COM interface (in this case COM3). The second parameter is the function for output. The function for output can be user defined or one of the system provided defaults. In this case we are connecting the stream output for COM3 to putsUSART3(), which is a basic output interface for USART3 TX.

The final function call is the meat of the program. A call to print() with COM3 selected as the stream and a string to print will cause “Hello World” to be printed out the USB port. To see this, use a terminal program such as TerraTerm.

Now we must back track to what happens before the first function call. If you were especially observant, you may have noticed that we have yet to setup clocks, or, in this case, even the USARTs. The missing link is the bootloader.

Before transfering control to the user program, the bootloader configures USART3, the Watchdog, the clocks and SysTick timer are all configured. Since we are using USART3 connected to USB, we dont have to configure the port (or any other resources configured by the bootloader). The exception would be if we needed to change the baud rate, in which case we would call comConfig() as in the example below. Please note that in this example code comConfig() is being called to demonstrate usage, but isnt actually required.

```
#include “pygmy_profile.h”

void comtest( void );

u8 globalBuffer[ 128 ];

void main( void )
{
streamInit();
streamSetRXBuffer( COM3, globalBuffer, 128 );
streamSetPut( COM3, putsUSART3 );
streamSetGet( COM3, comtest );
comConfig( COM3, 0, 0, 230400 );

while( 1 ){ ; }
}

void comtest( void )
{
```

```
print( COM3, "%c", streamGetChar( COM3 ) );
}
```

Notice the use of comConfig() in this iteration. We have also added a function to handler received characters. There are many ways to customize the comports, but one of the most common will be a modified receive handler. In this case we are allowing Pygmy to receive and buffer the incoming chars in the buffer we allocated. Pygmy will call our custom handler comtest() each time a char is received.

Since this is nothing more than a demonstration, we are merely echoing the character back to the originating comport.