Ambystoma Labs™

Technical Article AL-3000, April 2014

Protecting your IP with Easy Code Read Protection

Abstract: This technical article discusses a simplified method of implementing code read protection (CRP). The reduced complexity of this implementation will allow more embedded design engineers to choose CRP as a way to protect their valuable intellectual property.

Code Read Protection (CRP) is a valuable tool for the embedded designer. However, it is not without its challenges to implement. This technical article describes a simple way to get the maximum protection offered by CRP while allowing for updates to software without a complicated software scheme.

Generally, one starts to think about CRP the first time they utilize a large company to populate/build their embedded design. Although contract manufacturers (CM) are a valuable resource for a budding hardware company, they should be prevented from having all the tools to reproduce your product. One quickly realizes that a CM has everything from board layout, schematic files and a bill of materials. Without CRP, a suitably savvy individual associated with your CM could easily pull your code from shipping product and copy your hard work. The decision to utilize CRP is clear, but implementation can be complicated.

Our example design uses an NXP LPC1100 series processor and the software was created in the LPCXpresso IDE, however the methods presented here will be applicable to many embedded processors offering CRP. CRP involves writing manufacturer specific binary sequences into flash that instruct a microprocessor to disable certain types of access to the flash memory where the embedded firmware is stored. NXP provides 3 levels of CRP protection and all of them will disable the Serial Wire Debug (SWD) port. Additionally, writes/reads to flash can only be done via UART0 In-System Programming (ISP) once CRP has been activated. (For CRP3 ISP is disabled and would need to be re-enabled in your code.)

To sum it up: if you choose to invoke CRP you are then presented with the problem of coming up with a UART0(ISP) based communication system with authentication and which sends an encrypted version of your code over to your product. As much as we like writing software, we just didn't feel like taking the time to do this. Additionally on our design, all ports except the SWD were in use so the decision to come up with a clever solution for CRP was easy since the ISP port was already occupied.

Hence, Easy CRP!

Before you proceed, decide exactly what you intend to do with CRP. Do you?

1. Want to protect your code.
2. Find it impractical and unnecessary to allow for field firmware upgrades.
3. Want to make sure that when you have 1,000 pieces of your product in inventory and suddenly come up with a game changing new feature, you can still update the code in your inventory, instead of scrapping it or waiting to build fresh product.

Is the answer yes to all? Then our method may be a good choice for you.

In this method, we invoke NXP's level 3 CRP. This means code cannot be read from or written to flash via any port. To facilitate performing updates to our product we provide a way within our software to instruct the device to erase its flash. Once the flash is erased, the microcontroller is back in a virgin state and can be programmed via the SWD or ISP ports.

That's it! We tell the device to keep everything secret with CRP 3 and then we give it a way to be told to erase its flash. Once the flash is erased and the microcontroller is power cycled (NXP devices read CRP bits during power up, not during reset) the device is in a virgin state and can be reprogrammed. Finally, we design the erase such that it cannot occur accidentally.

Now here is the code…

Again, this is NXP LPC1100 series using the LPCXpresso IDE. The first part sets the stage for CRP 3.

In driver_config.h:

```
#define CONFIG_ENABLE_DRIVER_CRP   1
//#define CONFIG_CRP_SETTING_NO_CRP    1
#define  CONFIG_CRP_SETTING_CRP3_CONSUME_PART   1
```

Next you want the erase to occur during start up and never again. This will prohibit accidental erasure in normal operation. Due to this, this part will be in main.

In main.c with the #includes:

```
#include "driver_config.h"

#define IAP_ADDRESS   0x1FFF1FF1
typedef unsigned int (*IAP)(unsigned int[], unsigned int[]);
static const IAP iap_entry = (IAP) IAP_ADDRESS;
```

Flash erase and retrieving the guid or device serial number require using the In Application Programming (IAP) commands. This is how NXP does it.

Then at the very beginning of main:

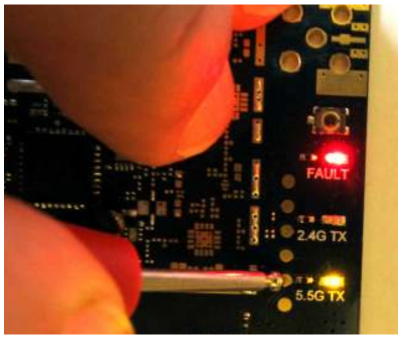
```
    int main (void)
    {
        uint32_t c, c2;  //General use counter variables
        uint32_t command[5], result[4];  //Used by the IAP commands


        //This is the part to erase flash for software update.
        //Look for P1_11 high
        //Clear personality bits
        LPC_IOCON->PIO1_10 &= 0b11111111111111111111101111000000;  //This is the Fault LED
        LPC_IOCON->PIO2_0 &= 0b11111111111111111111101111000000;  //This is the 2.4GTX LED
        LPC_IOCON->PIO1_11 &= 0b11111111111111111111101111000000;  //This is the 5.5GTX LED
        //Set pull down resistors
        LPC_IOCON->PIO1_10 |= 0b1000;
        LPC_IOCON->PIO2_0 |= 0b1000;
        LPC_IOCON->PIO1_11 |= 0b1000;

        //Set up the fault light (1_10) as indicator for flash erase operation
        LPC_GPIO[1]->DIR |= 0b10000000000; //This sets the data direction register for bit 10 of port 1 for output.
        LPC_GPIO[1]->MASKED_ACCESS[0b10000000000] = (0b00000000000); // We normally put all the zeros in the code just
        to keep track of which bit we are writing
        //Look for 5.5GHz light high and 2.4 GHz light low
    if ((LPC_GPIO[1]->MASKED_ACCESS[0b100000000000])&& !(LPC_GPIO[2]->MASKED_ACCESS[0b1]))
    {
        LPC_GPIO[1]->MASKED_ACCESS[0b10000000000] = (0b10000000000); //Set fault light on to indicate first part
        of erase procedure
        for (c=0xffffff; c>0; c--);  //Wait about 5 seconds

        //Look for 5.5GHz light low and 2.4 GHz light high
        if (!(LPC_GPIO[1]->MASKED_ACCESS[0b100000000000])&& (LPC_GPIO[2]->MASKED_ACCESS[0b1]))
        {
            LPC_GPIO[1]->MASKED_ACCESS[0b10000000000] = (0b00000000000); //Set fault light off to indicate second
        part of erase procedure
            for (c=0xffffff; c>0; c--);//Wait about 5 seconds

            //Look for 5.5GHz light high and 2.4 GHz light low
            if ((LPC_GPIO[1]->MASKED_ACCESS[0b100000000000])&& !(LPC_GPIO[2]->MASKED_ACCESS[0b1]))
            {
                LPC_GPIO[1]->MASKED_ACCESS[0b10000000000] = (0b10000000000); //Set fault light on to indicate
            last part and commitment of erase procedure
            for (c=0xffffff; c>0; c--);//Wait about 5 seconds.  At this point the device could be reset or power
            cycled to terminate erase procedure
            LPC_GPIO[1]->MASKED_ACCESS[0b10000000000] = (0b00000000000);//The fault light is turned off and the
            erase procedure has commenced.  We now wait a quasi random period of about 0 to 5 seconds before
            erasing the device.
            //This prohibits someone from anticipating when they could terminate the erase process and potentially
            extracting a portion of the code.

            //this part uses the guid serial number to create a quasi random wait period
            command[0] = 58;
            iap_entry(command, result);
            c=(result[1]&0xff0000)|(result[2]&0xff00)|(result[3]&0xff); //We take a byte of the last 3 guid words
            to form a quasi random number. For reference 24 bits high is a little over 5 seconds with 48MHz clock

            while(c) c--;

            //And we erase the flash!
            command[0] = 50;
            command[1] = 0;
            command[2] = 1;
            iap_entry(command, result);
            command[0] = 52;
            command[1] = 0;
            command[2] = 1;
            command[3] = 48000;
            iap_entry(command, result);

            //At this point the device is in a virgin state and the CRP bytes have been cleared.
            //For NXP microcontrollers the device must be power cycled in order for the cleared CRP state to be
            recognized.
            //Pointless to have a reset since flash is blank and the next command won't load.
        }
            }
        }
  //If we did not complete the erase command sequence, then we boot normally
```

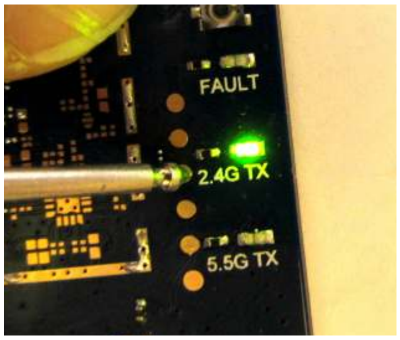
The rest of your code goes here.

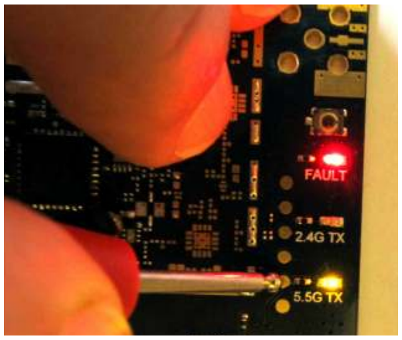
As you can see from the code, the process goes as follows:

1. On power up or reset, look for 1_11 high and 2_0 low. Set 1_10 high, then wait 5 seconds.
2. Look for 2_0 high and 1_11 low. Set 1_10 low, then wait 5 seconds.
3. Look for 1_11 high and 2_0 low. Set 1_10 high, then wait 5 seconds.
4. Set 1_10 low. Wait quasi-random period derived from quid serial number and then erase the flash.

Below is shown the sequence to erase the flash, if it were performed manually. On the example board, the I/O bits 1_10, 1_11 and 2_0 are available on test pads used for a factory programming fixture. Manual execution of the erase sequence is shown simply to better illustrate the process. Ideally, one would script this to run on their factory test/programming setup.



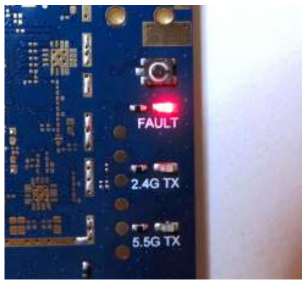




Use a probe with a series resistance to apply power to 1_11. Then apply power to the board or perform reset. 1_10 will illuminate indicating beginning of the erase sequence.

Within 5 seconds, apply power to 2_0 and wait for 1_10 to extinguish.

Within 5 seconds, apply power to 1_11 and wait for 1_10 to illuminate.



Once 1_10 illuminates, within 5 seconds remove power from 1_11. 1_10 will then extinguish and the flash will be completely erased within 5 seconds. After waiting the full 5 seconds, power cycle the board (it is not sufficient to perform a reset) and the device will be back in a virgin state able to be programmed through the SWD.

Richard J Moldovan

Studied Electrical Engineering at Florida Atlantic University. After graduating with a BSEE in 1999, he spent 15 years designing products in the wireless telecom, wireless infrastructure, wireless networking and defense electronics sectors. He is the founder and CEO of Ambystoma Labs Inc.

Email: richard.moldovan@ambystomalabs.com