Lab Writeup and Documentation

Informal writeup and code docs

By Loren Ashfield, lashfield@ucsb.edu

Loren Ashfield 2024

Table of contents

Getting Started 2
Setting Up Embedded Development with CLion 4
Initial C++ Algorithms 5
Rewriting into C Algorithms 6
STM32H7: Initial Benchmarks 8
STM32H7: Further Developments 11
STM32H7: Some Init Functions 14
STM32H7: FreeRTOS 16
Porting to ATSAMV7 17
Data Processing Scripts 18
HiFive Inventor Board 19

Getting Started

When I first started I had very little idea about what embedded development was or how it worked. I started out by watching a YouTube tutorial series (https://www.youtube.com/watch?v=gL8OoS9E1rw) provided by STM going over the basics of CUBE IDE, their proprietary development engine. Following that tutorial, I created a few simple programs for the STM32 board, one counted from 0 to 7 in binary using the 3 output LEDs and another was a simple reaction time game. This taught me the basics of embedded development and how to control the functions of the board using the hardware abstraction layer.

Code snippets from original STM32 program:

countBinary(): LEDs counting from 0 to 7

```
void countBinary(){
    for (int value = 0; value < 8; value++){
        if (value & 1)      // Turn pins on
            HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
        if (value & 2)
            HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
        if (value & 4)
            HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
        HAL_Delay(500);

        if (value & 1)      // Turn pins off
            HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
        if (value & 2)
            HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
        if (value & 4)
            HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
        HAL_Delay(500);
    }
}
```

reactionGame(): Produces a random delay after which you must click the button within 200ms

```
void reactionGame(){
      win=0;

      int r = rand() % 2000;
      HAL_Delay(r + 1000);
      if (win==1){
          HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
          HAL_Delay(1000);
          HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);

          win=0;
          return;
      }

      HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);

      HAL_Delay(200); // Time (in ms) given for player to react

      HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
      if (win==1){
          HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
          HAL_Delay(1000);
          HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
      } else {
          HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
          HAL_Delay(1000);
          HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
      }


      win=0;
}
```

Setting Up Embedded Development with CLion

In my opinion, the manufacturer supplied proprietary IDEs are far inferior to standard IDEs. Especially because I was no longer using the HAL and did not need the auto generated code, I wanted to get away from CubeIDE. Towards the end of the STM32H7 testbench development I managed to set up CLion with embedded development which in my opinion is far superior. I'll go over it here.

One of the benefits of using CLion is that it works for most consumer microprocessors, meaning as I switch from the STM32 to an ATSAMV7, the transition should be fairly seamless.

Setup

For the most part, I followed this guide by JetBrains, the creator of CLion.

Embedded Development in CLion Docs
(https://www.jetbrains.com/help/clion/embedded-overview.html)

Since they explain it very well, I'll just go over a few small points. I use GCC to compile the programs and downloaded the Arm GNU Toolchain. For all the processors I'm using, the "AArch32 bare-metal target (arm-none-eabi)" version is the right one. I then downloaded OpenOCD for the debugger. If you're using bare metal programming, you can ignore anything they're saying about integration with CubeMX and `.ioc` files. `CMakeLists.txt` files can be fairly easily found on GitHub for the specific processor you're using and only small modifications need to be made. All in all, you have to jump through a few more hoops, but you save up for it with a much more streamlined development process.

Initial C++ Algorithms

After writing the first programs in CubeIDE I kind of understood more of how microprocessors worked, and I started writing algorithms to test on the processor. I settled on three different simple tests to test a range of different processor functions: bubble sort, matrix multiplication, and SHA256 hashing. I wrote the original algorithms in C++, complete with multithreading to run simultaneous tests for data verification. The program also used std::chrono to time each test run and check for timing delays.

For example, the runFunction() code is provided below:

```
template<typename T>
void runFunction(std::function<T()> func, std::promise<T> promise,
std::chrono::time_point<std::chrono::high_resolution_clock>&
start,
std::chrono::time_point<std::chrono::high_resolution_clock>& end)
{
    try {
        start = std::chrono::high_resolution_clock::now();
        promise.set_value(func());
        end = std::chrono::high_resolution_clock::now();
    } catch (...) {
        promise.set_exception(std::current_exception());
    }
}
```

Each thread calls this function, which in turn runs and times a test function.

As it turns out the memory management and optimizations of these algorithms was too complex, and therefore it wasn't good for benchmarking the processor. From here on I program everything in C.

Rewriting into C Algorithms

Since the memory management using C++ classes like vectors was not ideal, I started programming in C. I rewrote the simple test algorithms and also added a merge sort. I figured since merge sort was a very memory intensive process, it would be more likely to produce useful memory errors when bit flips were created in registers. There is not much to show with these algorithms as they are just standard known algorithms, therefore I am not adding them here.

When I was writing the C version of these algorithms I also started a simple fault injector that randomly flipped bits from a preset range. Keep in mind at this point I was still testing and running all these algorithms on my computer therefore the memory and processes were all run by my computer's processor.

The faultInject() function is given below:

```
void injectFault(void *data) {
    // Cast the data to a byte array
    unsigned char *bytes = (unsigned char *)data;
    size_t byteIndex = rand() % 10000;
    unsigned char bitIndex = rand() % 8;
    bytes[byteIndex] ^= (1 << bitIndex);
}

// Fault injector thread function
void *faultInjector(void *arg) {
    void **args = (void **)arg;
    void *data = args[0];

    while (1) {
        injectFault(data);
        usleep(100000); // Limit fault injection rate
    }
}
```

The function basically just takes all the available data in the program and randomly starts messing stuff up. It may or may not have been able to access data outside the program,

it's probably fairly dangerous. The issue with this fault injector was that it never really affected the runtime of the program because I didn't know where to target the bit flips, so I just made it cover a huge range of memory.

I don't really remember the outcome of this program, it was a few weeks ago now. I didn't develop it further though because after this I started coding the benchmarks for the specific processor instead of just running them on my computer.

STM32H7: Initial Benchmarks

My initial process for writing the STM32 benchmarks was to take the algorithms I wrote and copy-paste them into a fresh CubeIDE project. This worked fine at the beginning, but I was still a little stuck. I didn't have any examples of what a radiation testing benchmark looked like, so I was unsure of how to develop the program further.

At this point I came across a handful of radiation testing programs created by the Los Alamos National Laboratory, and from here on out all my work was heavily influenced by their benchmarks.

LANL Benchmark Github (https://github.com/lanl/benchmark_codes?tab=License-1-ov-file)

Their code was extremely simple and light, which is a feature I tried to replicate from here on out. I was also unsure prior to this point on what to output during the tests, but after seeing these examples I emulated their style. These tests were written for a Texas Instruments MSP430F2619, therefore I had to adapt the code to work with the STM32. One thing that stood out to me was that, instead of using a HAL (hardware abstraction layer), these tests directly managed the memory of the chip. I never liked the HAL on the STM32 because the amount of code made the compile time slow and clunky, and it also felt like it was moving a lot of unnecessary register bits. I didn't know it was possible to avoid the HAL altogether, but these tests showed me it was. With this I dove into the world of bare metal embedded programming.

Converting the LANL code

At this point I more or less threw out everything I had so far and just started trying to get the LANL programs to run on my processor. I knew nothing about bare metal programming and even barely anything about processor architecture, registers, etc. so I was truly starting from scratch. My first attempts were just throwing the code into chatGPT and trying to get out a usable answer. Since chatGPT probably has all the documentation for these processors in its memory, it was actually half decent at creating a jumping off point. Without it, it probably would have taken me at least another week to figure it out. The general structure of the code was the same as for the TI processor, just the macros had different names and some items had to be configured differently. A comparison of initializing the UART module (`initUART()`) for the two processors is shown on the following page.

TI UART Initialization (From LANL code)

```
P3SEL = 0x30;                      // P3.4,5 = USCI_A0 TXD/RXD
UCA0CTL1 |= UCSSEL_2;              // SMCLK
UCA0BR0 = 104;                     // 1MHz 9600; (104)decimal =
0x068h
UCA0BR1 = 0;                       // 1MHz 9600
UCA0MCTL = UCBRS0;                 // Modulation UCBRSx = 1
UCA0CTL1 &= ~UCSWRST;              // **Initialize USCI state
machine**
//IE2 |= UCA0RXIE;                 // Enable USCI_A0 RX interrupt
```

Final Version of the STM32 UART Initialization

```
// Enable the clock for USART3 and GPIOD (USART3 TX/RX pins)
RCC->APB1LENR |= RCC_APB1LENR_USART3EN;
RCC->AHB4ENR |= RCC_AHB4ENR_GPIODEN;

// Configure PD8 and PD9 as alternate function for USART3 (AF7)
GPIOD->MODER &= ~(GPIO_MODER_MODE8 | GPIO_MODER_MODE9); // Clear
mode bits
GPIOD->MODER |= (GPIO_MODER_MODE8_1 | GPIO_MODER_MODE9_1); // Set
to alternate function mode
GPIOD->AFR[1] |= (7 << GPIO_AFRH_AFSEL8_Pos) | (7 <<
GPIO_AFRH_AFSEL9_Pos); // Set AF7 for PD8 and PD9

// Configure USART3
USART3->CR1 &= ~USART_CR1_UE; // Disable USART3
USART3->BRR = 64000000 / 9600; // Set baud rate to 9600
USART3->CR1 = USART_CR1_TE | USART_CR1_RE; // Enable transmitter
and receiver
USART3->CR1 |= USART_CR1_UE; // Enable USART3

// Wait for the USART to be ready
while (!(USART3->ISR & USART_ISR_TEACK) || !(USART3->ISR &
USART_ISR_REACK));
```

Initially I would just chatGPT the code a hundred times and throw it into the processor to see what worked and what didn't. I also found a few very helpful examples of STM32 bare metal implementations which I linked below.

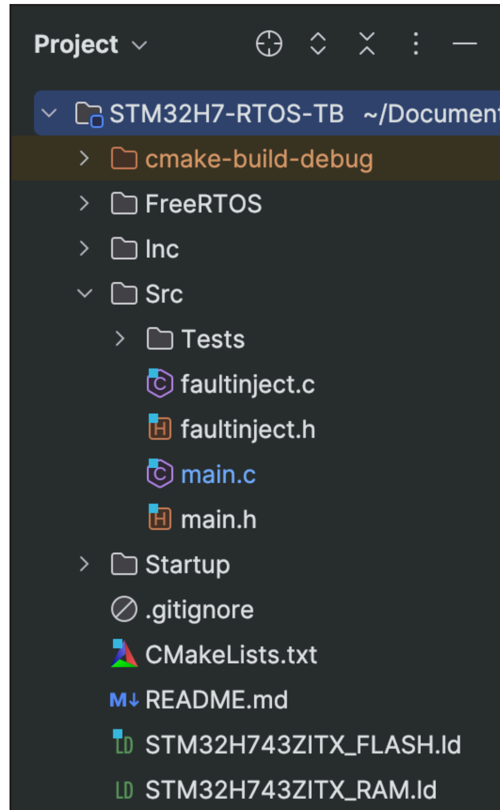
Source Links:

1. Vivonomicon (https://vivonomicon.com/2020/06/28/bare-metal-stm32-programming-part-10-uart-communication/)
2. Martindoff's project (https://github.com/martindoff/bare-metal-stm32h743-HIL)

These implementations were either for slightly different model processors or were implementing things in different ways than I was. For example, Martindoff was using the same board as me but was running his system clock from the HSE (High Speed External) oscillator, an optional add-on which my board didn't have. Instead, I was using the HSI (High Speed Internal) oscillator. Looking at all these examples, reading documentation, and troubleshooting the extremely half-baked chatGPT code over the course of a few days threw me into the deep end of embedded programming, but it actually all came together and started to make sense. With half an understanding of how I actually did it, I was able to enable the GPIO, System Clock, and USART of my STM32 board, and I was able to run the LANL code.

STM32H7: Further Developments

After I got the basic STM32 function initialized with the bare metal setup, I was able to run basically any test algorithm using this initializer code. I separated the tests from the main file, where `main.c` holds the initializer code and any algorithm can be added to the `Tests` folder then called by main.



File Structure of final STM32 Testbench

This made it super easy to run all kinds of algorithms on the processor, either ones I wrote myself or ones I found online. I altered the native C `printf()` function by writing a custom `__io_putchar()`, therefore any algorithm that used `printf()` would instead send the data over UART, making importing algorithms to my program really easy.

```
int __io_putchar(int ch){
    USART3_TransmitChar(ch);
    return ch;
}
```

I then started working on the fault injector in order to actually test these algorithms. For the actual injector function, I modified an already written function sent by Almudena. It chooses a random bit within the STM32's r0-r12 registers and flips it with an inline assembly XOR.

The harder part of getting the fault injector to work was enabling the board's timer. The injector had to run on a set time frame, therefore the board needs to be counting ticks since it's been initialized. I ended up using TIM1 on the board to run the clock.

At about this point is when the bare metal programming really clicked and I understood what I was doing. I learned how to read STMs very helpful documentation, and even though it still took troubleshooting, I finally understood what was going on instead of just throwing code at a wall to see what stuck.

I came across a 3300-page manual from STM detailing every bit and every register inside their STM32H7 line of processors. The documentation for each register mostly looks like this:

45.4.2 IWDG prescaler register (IWDG_PR)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PR[2:0]		
													rw	rw	rw

Bits 31:3 Reserved, must be kept at reset value.

Bits 2:0 **PR[2:0]:** Prescaler divider

These bits are write access protected see *Section 45.3.6: Register access protection*. They are written by software to select the prescaler divider feeding the counter clock. PVU bit of the *IWDG status register (IWDG_SR)* must be reset in order to be able to change the prescaler divider.

000: divider /4
001: divider /8
010: divider /16
011: divider /32
100: divider /64
101: divider /128
110: divider /256
111: divider /256

Note: Reading this register returns the prescaler value from the V_{DD} voltage domain. This value may not be up to date/valid if a write operation to this register is ongoing. For this reason the value read from this register is valid only when the PVU bit in the IWDG status register (IWDG_SR) is reset.

IWDG_PR Register Map in Datasheet

Basically the header files for the chip contain a bunch of macros which each flip specific bits in the registers. Whether these bits are on or off changes the states or configurations

of parts of the chip. This document lets you look up a macro from the provided header files, and then read exactly what changing each bit does. For example take the image on the previous page. My IWDG initialization contains the line `IWDG1->PR |= IWDG_PR_PR_2;`, which sets the `IWDG_PR_PR_2` macro to the `IWDG1_PR` register. Mousing over the macro shows it holds the value `(0x4UL << (0U))`. This line means this macro inserts the hex value 4 into the register with a 0 bit shift. In other words, when applied to the `IWDG1->PR` register, it sets `100` to `[PR]2:0`. You then check the documentation to see if it is the right functionality, and you're done. Using the documentation you can find the right macros, find the initialization process, and find the effects. The IDE's macro autocomplete is also an extreme time save.

Once this whole register management part clicked for me programming this initialization became a world easier. There are even very intuitive naming conventions and everything that make a ton of sense once you get used to it.

After the TIM1 was set up, I initialized IWDG1 (one of its registers is seen in the photo) which is a built-in watchdog for the board. This made it so that the board would automatically reset itself if the watchdog counter is not being reset, which only happens if the program freezes due to an error. Finally, I initialized the RNG module of the board so that the injector could flip truly random bits at truly random times. Since this module uses an on-board oscillator to generate random numbers, the numbers are different every time even after a watchdog reset. This module was extremely finicky for some reason and kept giving me errors, but I was able to get it to work after setting it to run using a PLL (Phase Locked Loop) I set up. I still don't really know why it was acting up, I spent hours reading documentation, but my workaround is fine.

Link to STM32H7 Documentation (https://www.st.com/resource/en/reference_manual/rm0433-stm32h742-stm32h743753-and-stm32h750-value-line-advanced-armbased-32bit-mcus-stmicroelectronics.pdf)

STM32H7: Some Init Functions

For reference, some initialization functions from the main.c file are given below:

RNG_INIT()

```
void RNG_Init(void){
    // Enable the RNG clock
    RCC->AHB2ENR |= RCC_AHB2ENR_RNGEN;

    // Use PLL clock bc for some reason default doesn't work
    RCC->D2CCIP2R &= ~RCC_D2CCIP2R_RNGSEL_Msk;
    RCC->D2CCIP2R |= RCC_D2CCIP2R_RNGSEL_0;

    // Enable the RNG peripheral
    RNG->CR |= RNG_CR_RNGEN;

    // Wait until a random number is ready
    while ((RNG->SR & RNG_SR_DRDY) == 0);

    srand(RNG->DR); // Set seed for error generator
}
```

TIM2_Init()

```
void TIM2_Init(void){
    // Enable clock for TIM2
    RCC->APB1LENR |= RCC_APB1LENR_TIM2EN;
    // Enable TIM2 IRQ in NVIC
    NVIC_EnableIRQ(TIM2_IRQn);

    // Set prescaler to 63 (which will result in a 1 MHz timer
clock frequency)
    TIM2->PSC = 63;
```

```
    // Set auto-reload value to 999 (for a 1-second overflow)
    TIM2->ARR = 999;

    TIM2->EGR |= TIM_EGR_UG;
    // Enable update interrupt
    TIM2->DIER |= TIM_DIER_UIE;

    // Enable the timer
    TIM2->CR1 |= TIM_CR1_CEN;
}
```

IWDG1_Init()

```
void IWDG1_Init(void){
    // Enable LSI
    RCC->CSR |= RCC_CSR_LSION;
    while ((RCC->CSR & RCC_CSR_LSIRDY) == 0); // Wait for LSI to
be ready

    // Unlock IWDG1 registers
    IWDG1->KR = 0x5555;

    // Set prescaler
    IWDG1->PR |= IWDG_PR_PR_2;
    // Set reload value
    IWDG1->RLR = 0x0FFF; // Reload value

    while (IWDG1->SR == 0); // Wait for IWDG to be ready

    IWDG1->KR = 0xAAAA;

    // Enable IWDG1 registers
    IWDG1->KR = 0xCCCC;
}
```

STM32H7: FreeRTOS

Implementing was smooth, I just modified the FreeRTOS demo for this chip as recommended by the manual. I had to enable to FPU on the chip to get RTOS to work, but besides that it was pretty easy. I tried to keep the RTOS files fairly separate from the main program files so that it would be easily removable prom the program.

Link to FreeRTOS Demos (https://www.freertos.org/a00102.html)

Porting to ATSAMV7

The next chip I started working on was a ATSAMV71, and development really only took around a day. All I had to do was port my work over from the STM and rewrite all the memory access to work with the new chip. I started by downloading the library files for the chip from the manufacturer, and then just rewrote the main.c file using the macros provided by the new chip.

The only real issue I ran into the entire time was the chip not printing to the UART. After a lot of testing I believe it was because the standard `printf()` function uses more memory than the chip can allocate. To solve this, I added this implementation of tiny printf for embedded systems, and everything worked great.

Link to Tiny Printf (https://github.com/mpaland/printf)

I subsequently backported all improvements I made as well as tiny printf to the original STM testbench to maintain congruency between the two versions.

I added FreeRTOS to the chip after a lot of trouble. It seems like the main error came from minor variable name changes in updated versions of the chip support files.

Data Processing Scripts

In order to test the chips and process the resulting data, I wrote two quick scripts in Python. The first one collects the output of the UART into a `.txt` file, and the second one process the data and outputs the effect of each bitflip. The script does this by checking if the processor output an OK, DE, or WD, after each bitflip, and this data is then stored. OK signifies that the next line proceeded as normal, DE signifies a data error, and WD signifies a watchdog reset was forced.

As of now, the scripts and sample output files are available on a public github repo here.

Link to Processing Scripts (https://github.com/lorenashfield/Testbench-Processing)

HiFive Inventor Board

After I was done with the two M7 chips I had the option of either working on a M4 chip or a Risc-V chip and I decided to go with the Risc-V for something new. It ended up being very difficult, and although I was able to get it working on the proprietary software I was not able to get it working on a stripped down barebones version of the code.

This board was the HiFive Inventor board, a board that was designed to teach kids coding a few years back. This board I guess never made it very far and there are very few resources to go off of. The chip is a FE310-G003 chip, which is a slightly modified FE310-G002 chip which is in SiFive's more popular HiFive1 Rev B board, so most of the software and resources are cross compatible. I was able to get the chip working right away by using the Freedom Studio software from SiFive and using a test program built for a HiFive1 Rev B board. It would have been possible to just modify the code there and run whatever program.

I wanted to run stripped down version of the code without all the bloat and helper scripts added by the proprietary software, so I attempted to get development set up on my Mac. The first issue I ran into was that OpenOCD, which I was using to debug the software, did not support the flash memory chip on the HiFive inventor. I was able to get around this by modifying the OpenOCD software and compiling my own version, until I realized the latest, unpackaged version of OpenOCD on GitHub had support for this chip. I replaced my version with a version I compiled from the OpenOCD GitHub. As a side note, this board works natively with Segger J-Link GDB Debugger, but I was too lazy to download it until much later when I was desperate to get things to work.

The other thing I needed was a GNU Toolkit for Risc-V because the toolkit I was using was for ARM. I downloaded and compiled the latest version of riscv64-unknown-elf.

I think the main problems that kept arising stemmed from newer versions of these programs breaking older software. I noticed Freedom Studio used version 8 or 9 of the Risc-V toolkit while I was trying to use 13. 13 had different ways in which makefile arguments had to be defined, and I think because of that the startup code was processed differently. I could never get into the flash memory to actually run the program.

I also tried installing the Freedom E SDK to develop using their software development kit, but I was unable to build it with my version of Python, 3.12, since some features used in building it had since been deprecated. It is probably possible to get programs working

by downgrading software, but I really wanted to try to get it working using the latest software. I was unfortunately not able to before I ran out of time but please reach out to me if anyone gets it to work.