STM32L4 Bootloader Prototype Documentation

Emma Berry

Overview:

I have developed a functional bootloader prototype, capable of updating or replacing any on-board application. My prototype uses two STM32L476VG DISCO boards, one acting as host and the other as a receiver. Porting the bootloader to any STM32L4 device should be relatively easy; you'll just have to change certain memory addresses and function calls.

The bootloader is designed to receive and flash applications as Intel Hex files *only*. These are significantly larger than compiled binaries, but make the flashing process simpler and only writes the necessary data to flash. In short, the bootloader parses Intel Hex files and turns them into binaries before writing them to flash, throwing out any unnecessary metadata afterwards.

Currently, the compiled .Hex file is about 25 KB, meaning the final binary is about half that at ~12.5 KB. This includes everything the bootloader needs to function, including its own independent library. The file structure is as follows:

- *Project.uvprojx* - The project file for Keil. Converting this to CubeIDE should be trivial.
- *main.c* - The bootloader logic itself, along with other essential functions, reside here.
- *Mylibrary.c* - All HAL functions (along with some additional flash functions) reside here. Currently a bit of a mess, but it works. The bootloader relies *only* on this file and Mylibrary.h, along with stm32l476xx.h. Any additional functionality will probably require additions to this library, especially with any new interfaces.
- *Mylibrary.h* - Definitions, structs, etc for the library above.
- *UART_SetConfig.c* - Currently empty. Was going to move all UART functions from Mylibrary.c to here, but ran out of time. Would just make things a bit more organized.
- *SysClock.c & SysClock.h* - Configure PLL clock. Sets PLL source to HSI and PLL as SysClock source.
- *Startup_stm32l476xx.s* - Modified startup file to support Assembly Programs without the use of libraries. Essentially a smaller startup file that removes the need for certain libraries.
- *HAL_Drivers* - Currently-unused folder containing certain HAL files. Originally going to be integrated into the bootloader before I decided to create my own library instead.
- *Pin_Connection.txt* - Notes on GPIO pin assignments for various STM32L476 DISCO functions. Mostly unused.

Keep in mind that this bootloader was made *without* the use of CubeMX. This may make initial development more difficult, but the program is much smaller and more efficient because of it. There is an older version of the bootloader from a CubeMX file, but it doesn't contain the full functionality of the current prototype.

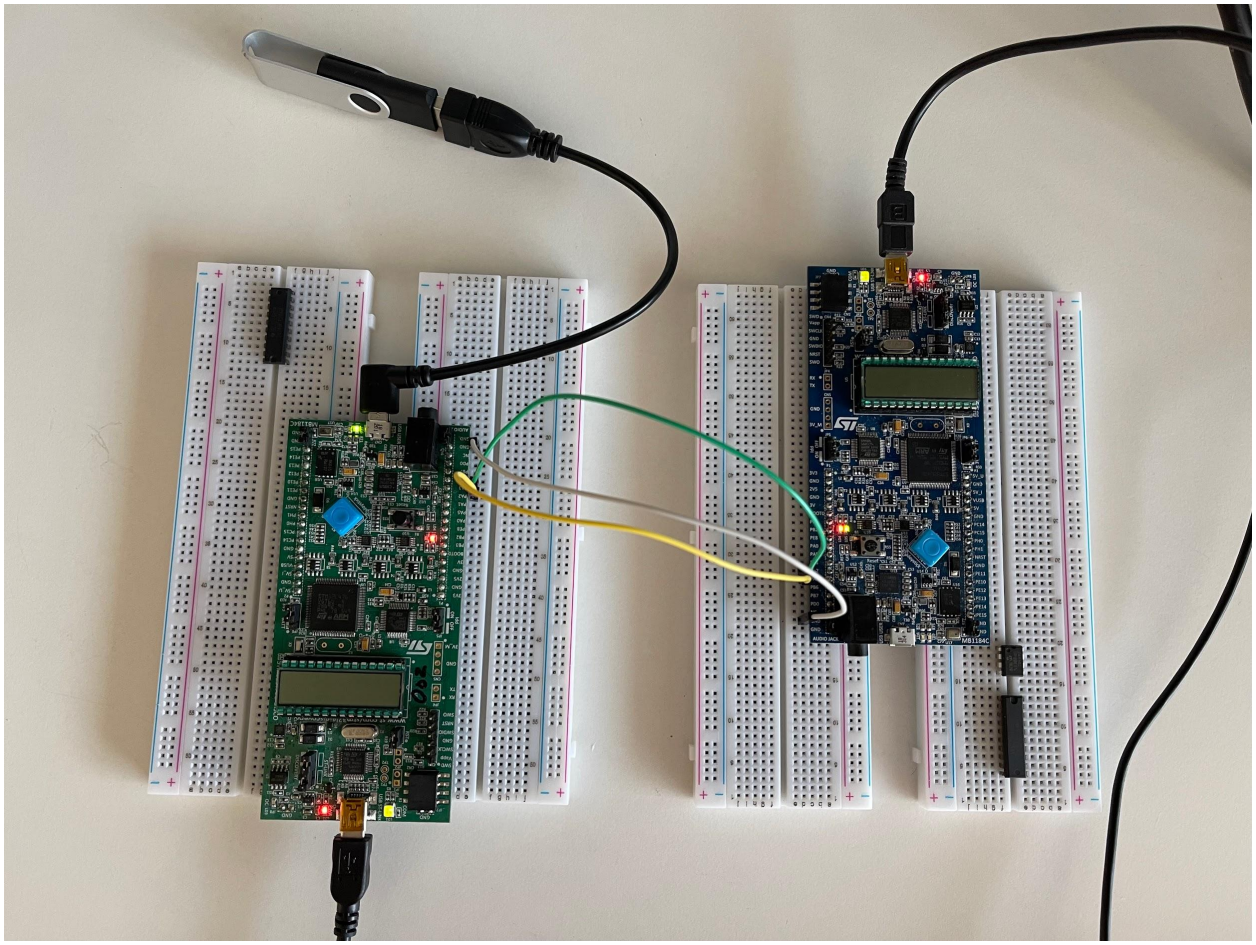

Figure 1: Visual overview of current prototype. I didn't have time to draw schematics :(

Host Requirements:

The current bootloader prototype is configured to only accept UART calls directly from another STM32L476 board. The host program is pretty straightforward; it's a CubeMX generated project which loads the .Hex file from a USB (via USB OTG FS) drive into an array, confirms the receiver (board with the bootloader) is requesting the file, and sends it over. The current build of the host program and bootloader are using a 16KB sample application for testing. Larger programs will require some modifications to the host and receiver code, mostly in modifying array sizes. I.E. any array holding the entire update at once would need to be resized accordingly from the current size of ~16,000 bytes.

Replicating such functionality over LORA will be difficult. The original plan was to have the RAK3172 module send the update data over UART to the STM32L432KB board, which this essentially mimics, but the current prototype just uses a USB drive (with FatFS drivers) instead of LORA for retrieving the file. For the time being, perhaps one board could be flashed with the .Hex file and connected to the receiver board (again, the one with the bootloader) over UART on a breadboard? Just a thought.

The current host build has full use of the HAL libraries, so implementing new functionality or moving things around should be pretty easy. Neither the host nor the receiver currently make any use of LL libraries.

The host program is currently configured to use pins PA2 and PA3 as USART2 TX and RX respectively. These will need to be converted to the relevant pins on your board.

As an aside, it may be easier to start with two DISCO boards before porting (I'd check eBay or just ask Dr. Wolfe if he can send some). This would let you get the prototype up and running pretty quickly.

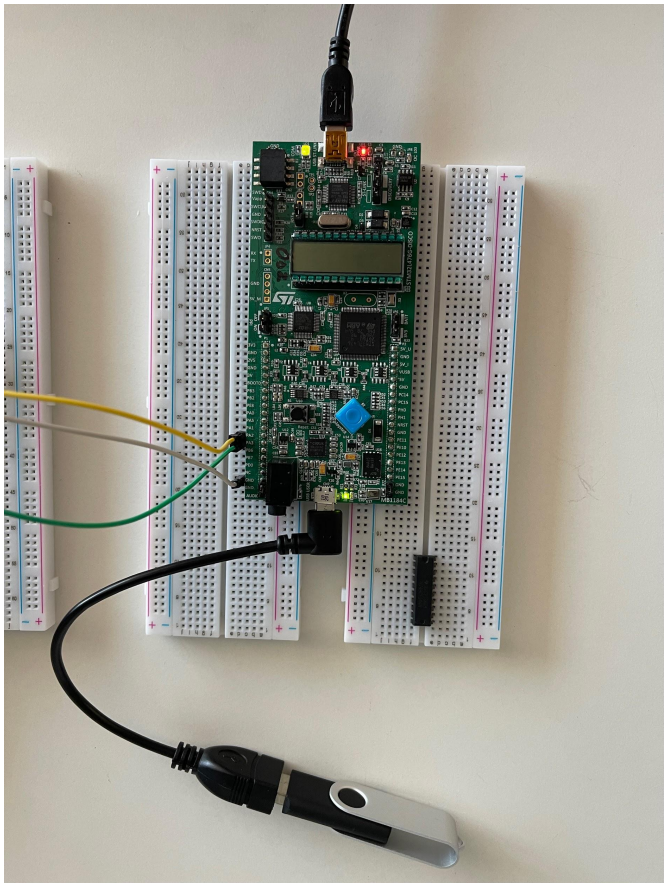

Figure 2: Visual overview of host board. Flipped upside down so you can actually read the pins

Receiver/Bootloader:

Note: The bootloader currently only accepts UART transfers in polling mode. UART interrupts are partially implemented, but don't currently work.

I'll cover some of the primary functions & structures here:

- struct HexIntel
    - Contains various char arrays and booleans to store a line of data, address, etc.
    - checksums currently unused
- hexCharToInt
    - Takes a given hex character from the r_update buffer and converts it to binary. Used to store two hex numbers in a single byte.
- Flash_Write
    - Takes a 32 bit address, a 64 bit data array, and a 16 bit integer (num of words) as input
    - Uses imported HAL functions and EraseInitStruct to write a given amount of data at a certain address
    - Writes one page at a time
    - Flash_Read_Data reads a given page and outputs it to any given buffer
- updateRequested:
    - Currently a little unstable, but seems to work ok
    - Polls host board for its version number
    - If the receiver version number doesn't match the host version number, receiver sends an ACK to host and returns true
    - Otherwise returns false
- Void jumpToMain
    - Loads defined address into stack pointer before shifting by 4 and moving to program counter (I think). Lets the bootloader jump to the main program.

Structure/sequence of main:

- After initialization, main reads the flash to determine the current version.
    - Was originally going to be in EEPROM, but didn't have time to implement this
- Assuming updateRequested is true, a temporary buffer is created to hold one page of data from r_update.

    - Since .Hex files have a bunch of metadata, the temp buffer is currently at 5600 bytes instead of 2048.
        - A page is 2 KB on the Discovery board. Unsure on the verdi board.
- Main then runs several checks
    - One: Polls host for start variable. Once it receives this variable, it's allowed to start parsing and flashing the data.
    - Two: Verifies that it is or isn't on the first page. If it is, it receives the whole update into r_update and transmits an acknowledgement back to the host (start_rec). If it isn't on the first page, it simply skips the above steps since it only needs to receive the update once. From here, the host's job is done and the bootloader will work with the r_update buffer to parse data.
- The temp buffer parses data from r_update, one page at a time. It converts all valid ASCII characters from r_update to binary and stores them.
- The temp buffer is subsequently copied to the data_page buffer.
    - This buffer is arguably redundant and could probably be removed.
- Main keeps track of how many lines of data temp has converted and stored in data_page, forcing it to stop once it reaches 255.
    - 256 lines of 8 bytes of data = 2048 KB
- From here, Main parses each line of data_page array for the record type and acting accordingly.
    - Check the .Hex wiki for more info on these records. It essentially either copies given data to hex_data_array (which is later flashed directly), writes down an address, or marks the end of the file.
- After parsing all the data, Main then stores the addresses of each line, copying them into a 32-bit array.
    - I only ended up using the first address of each array, so some of this code could probably be removed.
- Once all addresses have been copied to the array, Main writes to the flash
    - When writing the first page, I just used the MAINPROG address. I created another mainprog variable, but I'm pretty sure it's redundant.
    - If it's not writing the first page, it uses the first address of the hex_addresses array and subtracts 16. This removes any potential overlap or gaps in the code when it's written.
- If the page that was just written was *not* the last page, Main loops around and repeats the cycle.
    - If it was the last page, then the m_success flag will have already been set and Main will write down the new version number to flash.
- From here, the system has to be manually restarted, but upon restart, it *should* essentially boot straight into the new program.

While everything does function to varying extents, stability is not ideal and you *will* run into issues related to this. Luckily, the bootloader is relatively-easy to debug since it's only a few files. My code could use a bit of optimization and a lot of cleaning up, but I ran out of time to get through all of it.

Misc:

- Currently, both the host and receiver MUST be running at the same frequency for UART to work correctly. Not sure why, probably has something to do with polling.
- Mentioned before, but any additional functionality *will* require library modification. Keep this in mind as you try new things.
- Also be sure to keep an eye on the overall size. This program has to fit alongside the main firmware on the board, and we only have 128 kB of flash to work with.
- The Bootloader prototype has been uploaded to our shared Firmware Remote Update folder, along with this document.
- I've also uploaded a folder named Hex Padder. This contains a utility to "pad out" a given binary file to a set amount of pages. I.E. if your firmware is 44.5 pages, this will add a bunch of 0s to the end of the last page so that the firmware becomes 45 pages. Makes flashing much easier, since it goes a page at a time.