

# ECEN 5013

# FINAL PROJECT

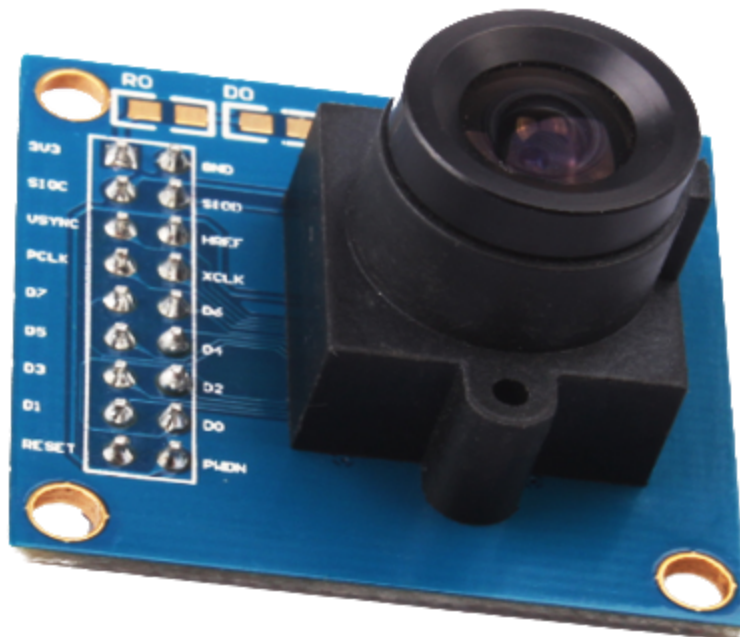
## Designing a camera module interface for the STM32F429

Release to be graded can be found at [https://github.com/benheberlein/ECEN5013\\_final\\_project](https://github.com/benheberlein/ECEN5013_final_project) with the tag *ecen5013-rel*.

Full documentation of the C code can be also be found in the Github under the *doc/* directory.

—

Ben Heberlein



# OVERVIEW

The goal of this project was to create a robust, structured, and extensible embedded software design that could control, configure, and manage functionality for several external peripherals using the STM32F429I-DISC1 discovery board from STMicroelectronics. The external peripherals used in this case were two different camera modules from OmniVision (OV7670 and OV5642) and a wifi module from Espressif (the ESP8266). In addition to external peripheral configuration, different software modules were created at various levels of abstraction to control functionality having to do with logging, profiling, on board tests, the command interface, and expandable control modules for SDRAM, camera, and wifi functionality. A detailed breakdown of specific software modules and their functionality will be provided later in this report. Several on board peripherals also had to be configured to implement the higher level software functionality. Among the STM32F429 internal peripherals used were the RCC, GPIO, USART, I2C, DCMI, and DMA. Again, there will be a detailed discussion of how these peripherals were configured later in this report. For a diagram of the entire system, look to *Figure 1* below.

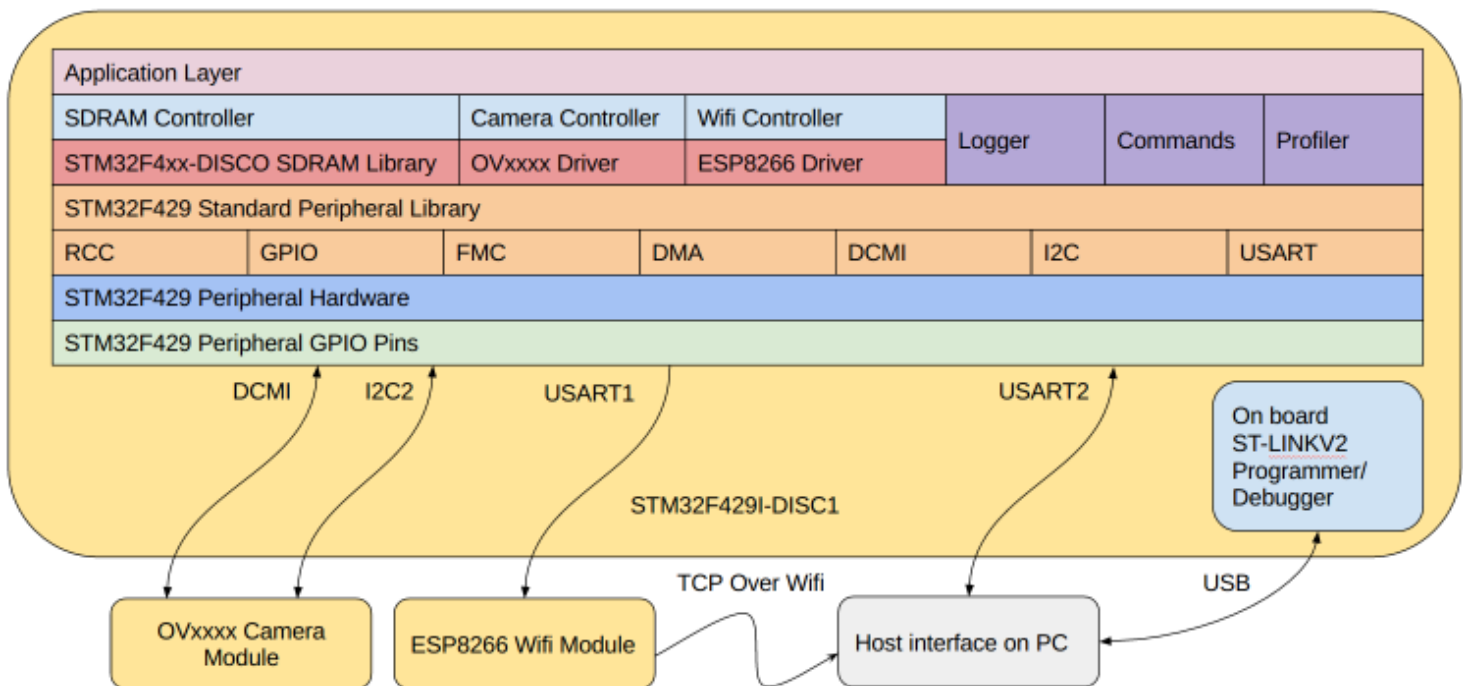


Figure 1. Top level system diagram

---

The main discussion of this report will be structured into several areas. First, we will discuss the build system, what options are present, and how the system can be configured for multiple functions. Next, the host interface is described, detailing user options as well as connection configurations. We will then spend the bulk of our description on the STM32F429 software structure, as it comprises most of our system. Finally, we will talk about how our external peripherals are configured and connected to the rest of the system.

## BUILD SYSTEM

The build system for this system uses the program *make* to cross compile an executable for the STM32F429. Our make system contains three files, the actual *Makefile* in project root directory, a list of sources in *config/sources.mk*, and a list of compiler tools in *config/tools.mk*. These two files are configured to support multiple build options, and handle the selection and implementation of compiler directives during our build. The make system passes these to the compiler during the build process, allowing us to configure our source code to turn on and off bits of code. This allows a configurable, size-efficient build process that can support a variety of peripherals. See *Figure 2* for a view of the project folder structure. This will be helpful in understanding the build system.

```
root/  
  bin/  
  build/  
  config/  
  data/  
  doc/  
  host/  
  inc/  
    drivers/  
    project/  
    startup/  
    test/  
  periph/  
    esp8266/  
  src/  
    drivers/  
    project/  
    startup/  
    test/
```

*Figure 2. Project folder structure*

---

## STRUCTURE

The build system takes source from the */src* directory and include files in the *inc/* directory and builds them into object files, assembly output, or preprocessed output in the */build* folder. If a full binary build is being performed, the executable is placed in the *bin/* directory. This is where the host interface will look for files when starting a debug session. Configuration files like additional makefiles, linker files, etc. will be found in the *config/* folder.

Depending on different build options, sources will be gathered from the subdirectories in the */src* folder. Because of the way the build system is implemented, we rarely will build everything in the */src* folder. Instead, the make system build a list of sources for only the files we need, saving significantly on code space in the end. For example, the *src/drivers/* folder contains files for the entire STM32F429 Standard Peripheral Library (SPL), but we will only ever build four or five of these files into our build system. In the future, if we need to implement a new code module that relies on the SPI interface, we can modify our *sources.mk* file to include the SPL SPI driver as well. The main idea is to separate the sources from the build logic so that the entire process is more organized and well understood.

The build system runs on top of the GNU ARM Embedded Toolchain<sup>1</sup>. This needs to be downloaded and installed before the build system will work. In the *config/tools.mk*, we define the cross-compiler tools we use. That way, if we need to modify our compiler toolchain to support other architectures or functionality, we can quickly change our set of tools.

## Options

The build system has several options that change what files and compiler directives are included. *Table 1* is a list of the build system options, along with their default configurations. The options can be passed in by executing '*make OPTION=VALUE*' at build time. These correspond pretty directly to software modules and compiler directives. For example, Setting '*LOG=INFO*' builds in the '*\_\_LOG*' and '*\_\_LOG\_INFO*' directives.

*Table 1. Build options*

Name	Options	Default	Function
<i>DEBUG</i>	<i>TRUE, FALSE</i>	<i>TRUE</i>	Enables debug mode
<i>LOG</i>	<i>INFO, WARN, ERR, NONE</i>	<i>INFO</i>	Enables logger and sets verbosity
<i>CMD</i>	<i>TRUE, FALSE</i>	<i>TRUE</i>	Enables host commands (see note below)
<i>PROF</i>	<i>TRUE, FALSE</i>	<i>TRUE</i>	Enables profiler
<i>TEST</i>	<i>TRUE, FALSE</i>	<i>TRUE</i>	Enables test framework
<i>CAMERA</i>	<i>OV5642, OV7670</i>	<i>OV5642</i>	Selects camera
<i>WIFI</i>	<i>ESP8266, NONE</i>	<i>NONE</i>	Enables wifi functionality and selects board
<i>BOARD</i>	<i>STM32F429I_DISCOVERY</i>	<i>STM32F429I_DISCOVERY</i>	Selects board

It should be noted that disabling command mode only disables commands from the host interface. This translates to ignoring the configuration of the USART1 Rx on the STM32F429. The command module should still be built into the software, as other software modules can queue tasks within the command module. The command module is described in more detail later on in the report.

## Targets

The build system supports several build targets as well, allowing the user to incrementally build and view things like preprocessor output. See *Table 2*.

*Table 2. Makefile targets*

Target	Description
<i>build</i>	Builds the full cross compiled executable in the <i>bin/</i> folder. This is the default.
<i>compile</i>	Compiles object files and puts them in the <i>build/</i> folder.
<i>preprocess</i>	Preprocesses files and puts the output in the <i>build/</i> folder.
<i>assemble</i>	Creates assembly files from source files, outputs to the <i>build/</i> folder.
<i>clean</i>	Removes all files from the <i>bin/</i> and <i>build/</i> folders.
<i>burn</i>	Builds an executable and flashes it to the board using <i>st-flash</i> .

---

# HOST INTERFACE

In the *host/* directory, there is a Python file called *host.py*. This program implements the entire PC side of the system, allowing the user to read status logs and data, debug, and command the microcontroller all from one spot. The host interface may require superuser permissions depending on how the serial ports are set up on the computer. The host interface has only been tested on Linux Mint 17.3 Cinnamon and Linux Mint 18.1 Cinnamon 64 bit environments, but should be able to work on other systems with only slight modification.

The host interface does several things. First, it prompts the user to select the serial port that will be used for debugging through UART. It then opens the port and configures it for the correct baud rate. Second, the program initializes a connection with the ST-LINK-V2 on the discovery board. In order to do this, a set of ST-LINK tools needs to be built and installed on the host machine<sup>2</sup>. These can be obtained from the link in the references. These tools allow the user to program and debug through the ST-LINK-V2 interface on the discovery board. The host interface then creates several threads to handle logging, commanding, network, and ST-LINK events independently.

The image shows a terminal window with a dark background and yellow/green text. At the top, it says 'Welcome to the Solens Debug Interface' followed by a dashed line. Below this is a large block of text consisting of many lines of 'D' characters and some numbers, arranged in a somewhat structured but mostly random pattern. At the bottom, after another dashed line, it says 'Please select a COM port.' followed by three numbered options: '0: /dev/ttyACM0', '1: /dev/ttyUSB0', and '2: /dev/ttyprintk'.

Figure 3. View of the host interface

## PROGRAMMING AND DEBUGGING

The debugger can be started from the host interface prompt by typing '*debug start*'. This command will look in the *bin/* folder for a valid *.elf* file to load onto the microcontroller. It opens up a GNU ARM Embedded Toolchain *gdb* instance, loads the code into the microcontroller flash, and sets a breakpoint at main. From there, the user can use standard *gdb* commands to step through and run code.

---

The ST-LINK tools described earlier could be used to directly flash the microcontroller without opening a debug session. Further along in the project, commands for flashing the board and running in release mode will be built in as well.

## HOST COMMANDS

The host interface comes with a series of commands that the user can input to control both the ST-LINK/debug session, as well as the microcontroller. These commands do not necessarily map to single command packets as described in the STM32F429 section, although they certainly do in some cases. Other commands however, solely control configuration on the PC side, and will not send commands to the microcontroller. *Table 3* contains all the commands available in the host interface.

*Table 3. Host interface commands*

Command	Description	Sends command packet to MCU?
<i>help</i>	Displays the help text.	No
<i>stlink restart</i>	Restarts the ST-LINK connection	No
<i>debug start</i>	Starts a debug session.	No
<i>debug stop</i>	Stops a debug session.	No
<i>debug restart</i>	Restarts a debug session.	No
<i>log init</i>	Initializes the logger module.	Yes
<i>cmd init</i>	Initializes the command module.	Yes
<i>sdram init</i>	Initializes the sdram.	Yes
<i>cam init</i>	Initializes the camera module.	Yes
<i>cam config</i>	Configures the registers in the camera module.	Yes
<i>cam capture</i>	Captures an image and stores it in SDRAM.	Yes
<i>cam transfer</i>	Transfers an image via UART or wifi, depending on build	Yes

In the *host.py* file, there is a separate thread for user input. This thread spins on user input from the terminal and adds the command to a queue if user input is found. The main program then checks this queue for data periodically, and, when there is data, executes the command or forwards it to the microcontroller in a log packet (or prints an error if the

---

command does not exist). See *Table TODO* in the STM32F429 Discovery Board section for the structure of a log packet.

## LOG HANDLER

A separate thread is created in the host interface to handle log events. If the executable on the microcontroller is built with logging enabled, log packets can come from the microcontroller at any time. For details about the structure or implementation of log packets, see the logger section in the STM32F429 Discovery Board section.

The log handler is similar to the command handler. The thread spins on the serial port (configured at the start of the program), and waits for data to come in. If data comes in from the UART line, it is put into a lightweight byte FIFO<sup>5</sup> for processing by main. The main loop then processes the data, unpacking the log and displaying the associated message and data. The host interface can detect based on the range of the status code if the log contains an info, warning, or error message, and displays the message with a distinct color for each log type. Again, the logging section in the microcontroller discussion will have more information.

## TCP SOCKET

In order to handle network events such as an image coming over wifi, the host interface contains a small TCP socket that acts as a server to the ESP8266 module. The server works almost identically to the logger, sharing the same packet structure and thread procedure. If data is received on the TCP socket, it is put into the bytes FIFO along with other logging messages, and processed accordingly. This allows us to send structured information over wifi in the same way as we did for the logger. While they share the same information packet structure in this system implementation, it is likely that the wifi packet structure will be changed in future implementations to support specific parameters such as timestamps, image sizes, etc. For a detailed view of the wifi packet structure, see the description of the wifi software in the next section.

Additionally, it is likely that the server will not reside on the host computer, but rather on a dedicated server through Amazon Web Services. By mirroring the log packet in this implementation, it will be easier to parse data on an external server in the next revision.



---

# STM32F429 DISCOVERY BOARD

The true majority of software for this project resides on the actual STM32F429 microcontroller on the discovery board. It is here that we created a hierarchical, structured codebase with a distinct error handling system and clean separation of functionality.

## STRUCTURE

As mentioned in the build system section before (See *Figure 2*), our code is organized into several subdirectories in our root directory. The *inc/* folder contains *.h* files for the declaration of all functions within the code modules. Each module will have its own distinct declaration file, where a user can find documentation and descriptions functions. The declaration files also contain definitions of structures for things like status codes and log, command, or wifi packet definitions. In the *src/* folder, there are the implementation files. These files implement functions described in the declaration files. Again, there is a single source file for each module, with the exception of the test module, which contains a distinct source file for each module under test.

Within both the *src/* and */inc* directories are several subdirectories. First, there is the *driver/* directory, which contains files provided by STMicroelectronics. Within this directory is the entire Standard Peripheral Library v1.7.1, as well as specific drivers like the SDRAM and LCD controllers which are unique to the STM32F429I-DISC1. This is where any additional STMicroelectronics libraries would go if the user was developing under a different board configuration.

There is also the subdirectory *startup/*, which contains configuration code for the ARM Cortex M4, the startup routine assembly file, and register definitions for the STM32F429. The system file *src/startup/system\_stm32f4xx.c*, which sets up clocks, voltage regulators, and processor buffers has been modified slightly from its original version to customize system clock speed. The system expects a 25 MHz crystal oscillator, but the discovery board contains a 8 MHz crystal oscillator. This leads to a strange system clock frequency that does not work well with the USART modules. In order to adjust for this the PLL was modified to create a 96 MHz system clock and the parameter `HSE_VALUE` was changed from 25000000 to 8000000 in the *Makefile*.

---

The main folder where we can find project source code is in the *src/project/* and *inc/project/* folders. These contain the definitions and implementations of both private and public functions for all the modules written in this project. While all function are technically public in the C programming language, the structure of the code is such that certain functions are not intended to be called from other modules. These still have normal declarations and documentation in the associated header files however, allowing a user to have full control over the module's internal functionality. This allows us to have all of the documentation in one spot, and helps the user to debug more effectively.

The functions and structures within a module follow a distinct naming convention. All functions in the *project/* folder are first designated by their module name, then an underscore, then a function name. The function name is capitalized based on whether the function is private or public (a convention somewhat inspired by the Go programming language<sup>4</sup>). For example, the *cmd* module has a function *cmd\_QueueGetStatus()*, which is public, but a function *cmd\_uartInit()* which is private and should not be directly called by the user. It is important to remember that these are only conventions to create an easy to use, readable, and structured codebase. The conventions don't actually mean anything to the C compiler or the freedom of the user to call any function from anywhere.

Finally, there are folders *src/test/* and *inc/test* which contain both the test function declarations and implementations. There is a separate test file for each module. Documentation for all *project/* files was generated by Doxygen and can be found in the PDF file in the *doc/* folder.

## LIBRARY DRIVERS

In this project, the on board peripherals are mostly accessed through the SPL. While there are some instances where register definitions from the *stm32f4xx.h* file are used directly, in general the peripherals are accessed through the library. This is an intentional decision that helps keep code more portable and organized. The SPL supports several different chips, and is well tested and maintained by STMicroelectronics. It is about as close to direct register access as one can get (unlike the higher level STMCube software package<sup>5</sup>), but organizes code in a readable and structured way. It should be emphasized that this is just a basic wrapper around peripheral register functionality. Most functions are a one-to-one mapping to desired register configurations, and in the few situations where we needed a more specific register manipulation, we can still access the register easily and change our configuration (this was needed for a few things in this design).

---

## AVAILABLE MODULES

In the *src/project/* and *inc/project* folders, there are files for all the software modules in the project. The file *inc/project/mod.h* has module type *mod\_t* that contains all the available modules. This file should be updated along with *inc/project/err.h* when new software blocks are added. *Table 4* shows a complete list of software modules with their unique IDs.

The modules written for this project are structured into two different hierarchical levels, the *Function Layer* and the *Peripheral Layer*. See *Figure 4* on the next page. You'll notice that the logger, profiler, and command module span multiple layers. These modules can be thought of as in the *Function Layer*, but with a direct link to the driver layer for initialization. These logger, profiler, and command module can be called from application code directly just like the SDRAM, wifi, and camera controllers. The only difference is that there will not be an

underlying peripheral layer for these modules. For the purposes of our discussion, they will be described in the *Function Layer* section.

*Table 4. All software modules*

Module	ID
<i>LOG</i>	0
<i>CMD</i>	1
<i>STDLIB</i>	2
<i>SDRAM</i>	3
<i>OV5642</i>	4
<i>OV7670</i>	5
<i>PROF</i>	6
<i>TEST</i>	7
<i>CAM</i>	8
<i>ESP8266</i>	9
<i>WIFI</i>	10

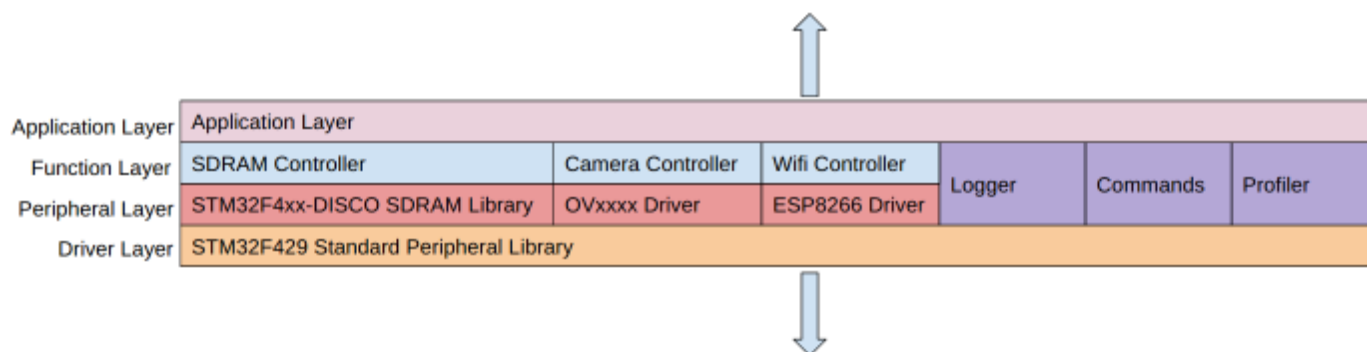


Figure 4. Structure of project modules in relation to other software.  
The modules described in this section are in the Function Layer or the Peripheral Layer.

## Peripheral Layer

The peripheral layer contains the drivers for the external peripherals. In this case, the drivers are for the SDRAM chip, the OV5642 or OV7670 camera modules, and the ESP8266 wifi module. The software in this layer is intended to be easily swapped out for other software depending on the selection of external peripherals. The inclusion of two separate camera modules is intended to illustrate this. We can easily swap out the OV5642 driver (default camera) for the OV7670 driver through the build system. The camera controller in the *Function Layer* will then call different functions based on the compiler directive ‘`_OV7670`’ supplied by the build system. The interface that the application layer sees, however, will remain unaffected, and application code can continue running as if the swap never occurred. The SDRAM and wifi controllers both work in a similar way, depending on the values supplied to the ‘`BOARD`’ and ‘`WIFI`’ build options. Refer back to *Table 1* for a look at the build system options.

### ESP8622

The ESP8266 has a relatively straightforward configuration. It runs a Lua based firmware called NodeMCU<sup>6</sup> that basically emulates the structure of the popular JavaScript-based Node.js in an embedded system environment. The actual code running on the ESP8622 consists of three separate Lua files. The files just initialize the system, connect to a network over wifi via an SSID and password, opens a socket to connect to the host computer, and initializes UART pins. The system then waits for UART data and forwards it over TCP to the host computer, where it is captured by the host interface program. The actual Lua files can be found in the *periph/esp8266/* directory. They are named ‘*init.lua*’, ‘*app.lua*’, and ‘*config.lua*.’

OV7670

The register file needs to be organized and explored more carefully - Most of the supplied register configurations did not appear to work for this setup. Eventually, after pouring through a very scattered datasheet, a working configuration was created by hand, and captured in the register map as *ov7670\_QVGA\_YUV\_regs*. The registers are not very well understood at this point, and an overhaul of the register file is in order before a quality picture can be taken.

The rest of the OV7670 implementation was straightforward, however. The camera works with several different on board peripherals in the STM32F429. In no particular order, these systems are the GPIO, RCC, I2C, Digital Camera Interface (DCMI), and DMA controller. The camera is configured using an I2C module (I2C2 in this case), writing specific one byte registers in a one byte address space. Technically, the OV7670 uses a communication protocol very similar to I2C, which OmniVision calls the Serial Camera Control Bus (SCCB)<sup>7</sup>. However, this is basically the same as I2C, with restrictions on how to send register addresses for larger address spaces. For the purposes of communication with the OV7670, we can write functions that call several instances of I2C functions.

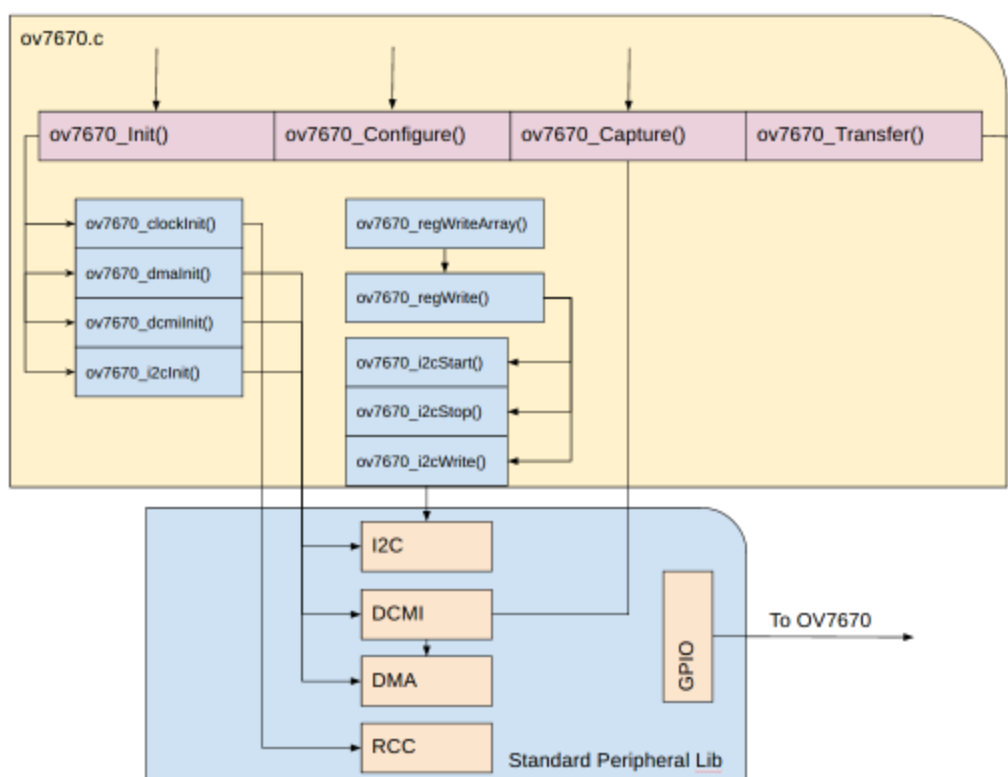


Figure 7. Simplified diagram of OV7670 driver

When the time comes to actually grab image data, we use a combination of DCMI and DMA to capture and transfer the image. The camera module, once configured properly, continually takes pictures as fast as it can and sends them out on its data lines (along with a clock *PIXCLK*, a vertical synchronization signal *VSYNC*, and a horizontal synchronization function *HSYNC*). It's the duty of the microcontroller to wait for a *VSYNC* frame to start, then start a DMA transfer using the data coming out of the camera module. To take an image, we set

---

a *CAPTURE* bit in the DCMI registers, causing the DCMI peripheral to capture on *VSYNC*, then automatically starts a DMA transfer (we have to configure the DMA to accept the correct DCMI channel). The image is then transferred through DMA to the memory location specified in the DMA setup - In this case, to the beginning of our SDRAM section.

Finally, to transfer an image, we simply invoke the logger to log our image data with the special status code *OV7670\_INFO\_IMAGE*. This status code is interpreted by the host interface as an image, meaning the host interface will save the image to the *data/* folder and display the image to the user in the debug interface. It should be noted that application level code should not call the *ov7670\_Transfer()* function. Instead, application code should call the *cam\_Transfer()* function which will also transfer an image out of SDRAM, but has the capabilities of transferring over wifi as well (depending on build configuration).

## OV5642

The configuration for the OV5642 is almost identical to the configuration for the OV7670, as they are from the same family of camera modules, with only a few key differences. In this section we will focus on the differences as the DCMI, DMA, RCC, and GPIO setup is nearly identical. The purpose of building a driver for two cameras is mostly just to show how code in the *Peripheral Layer* can be easily modified or replaced without affecting higher level code in the *Function Layer*.



Figure 8. The OV5642

The main difference between the OV5642 and the OV7670 is the address space within the module. They both share the same SCCB interface, but the OV5642 has a two byte address space instead of just a single byte. It is a 5MP camera, and because of its additional size it has additional complexity as well.

The implementation of the OV5642 camera also has a register map associated with it - but there is even less documentation on it than the OV7670. The register map defined in *inc/project/ov5642\_regs.h* was created by modifying a register map found in the open source Arducam<sup>8</sup> codebase. Again, these register configurations did not appear to work for our setup



---

right off the bat, but should be sufficient if explored thoroughly enough. For the OV5642, the driver can configure the module through I2C, but for now is incomplete in the fact that it can't configure the camera with a working register configuration. Extra time will be needed to explore the datasheet of the OV5642 and figure out which registers need to be tweaked in order to get a working image. Again, the inclusion of this peripheral code was more to demonstrate the hierarchical structure of this codebase, and the reusability of interfaces across multiple external peripherals and build configurations.

## Function Layer

The *Function Layer* consists of six different modules, each with functions that should interact with application level code or general higher level constructs. These modules were designed with the intention of being reusable by application code regardless of the external peripherals that implement their functionality. The user should be able to capture an image for whatever camera he/she is using by simply calling `cam_Capture()` - Given the system was built with the right options in place. This structure of code is modifiable, maintainable, structured, and easy to build upon.

### Camera

The camera software module, or '*cam*' as it is called in the code, implements camera functionality. The camera software can initialize and configure a camera and then capture and transfer an image from the camera to the host interface (either using wifi or the hardwired UART link). The camera software consists of functions that wrap around lower level camera driver functions depending on the preprocessor macros the system was built with. The camera also checks for lower level errors in the drivers for the OV5642 or OV7670 and uses those to report its own errors to the application level code. We will discuss the error management system in more detail in a few pages. This code is fairly straightforward once compiled, simply wrapping around lower level functionality.



---

## Wifi

The wifi software module works in a similar way to the camera software module described above. The wifi module also simply wraps around the ESP8266 driver (or another wifi module) and implements an initialization function and a send function. The wifi module also defines a wifi packet - which in this case is the same structure as the log packet. The wifi packet is defined in *Table 5* below. This structure is declared with ‘`__attribute__((packed))`’.

*Table 5. Wifi packet structure `wifi_packet_t`*

Field	Data type	Description
<code>wifi_packet_mod</code>	<code>uint8_t</code>	Module ID
<code>wifi_packet_status</code>	<code>uint8_t</code>	Module status from <code>err.h</code>
<code>wifi_packet_msgLen</code>	<code>uint8_t</code>	Length of message in bytes
<code>wifi_packet_msg</code>	<code>uint8_t *</code>	Buffer containing ‘\0’ terminated ASCII message
<code>wifi_packet_dataLen</code>	<code>uint32_t</code>	Length of data in bytes
<code>wifi_packet_data</code>	<code>uint8_t *</code>	Buffer containing data

Wifi packets of this structure will be serialized and sent to the host interface over the associated wifi module on a call to `wifi_Send()`.

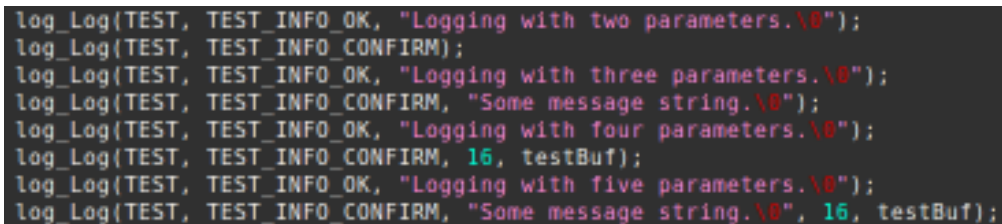
## SDRAM

The SDRAM, again, follows a similar pattern to the wifi and camera software modules. The SDRAM controller is a simple wrapper around the only SDRAM driver we have available, supplied by STMicroelectronics to drive the SDRAM for the discovery board. However, the structure will still be valuable to us because we can so easily configure another SDRAM driver beneath it. In the future, we are going to need to write our own SDRAM driver when we move away from the STM32F429I-DISC1 and onto our own PCB with a different SDRAM chip. Once that happens, we can simply write a new driver that implements initialization, read, and write functions, modify our build system to include it as an option, and set up new functionality in `sdram.c` based on the new compiler directives. While it may only wrap a single function now, it will preserve the interface to the application layer while maintaining configurability.

The logger was the first thing written, as it is critical to the debug process. The logger allows a frustrated user to add logging messages everywhere in his/her code and get feedback on the program as it progresses. The logger allows the user to see when and where invalid states happen through an extensive error and warning system. Best of all, the logger can be turned off when building the executable, without having to go back through the code and get rid of all of the log statements.

The log is implemented in a very user friendly way as well. There is a single function, `log_Log()` that can take any number of parameters between two and five, and logs them in a packet to the debug console. The `log_Log` function is a macro that expands to different logging functions based on the number of parameters included in the function call. By creating the main logging function in this way, the user only has to use a single function name, hiding the implementation of the logging system. The last thing a stressed out user needs to worry about is even more compiler errors based on the wrong log function.

The `log_Log()` function takes in a module ID of type `mod_t`, a general status code of type `gen_status_t`, an optional null terminated message, and an optional data packet with associated data length. Some example calls to the logger are shown in *Figure 9*. From these, we can see the general structure and configurability of a `log_Log()` call.



```
log_Log(TEST, TEST_INFO_OK, "Logging with two parameters.\0");
log_Log(TEST, TEST_INFO_CONFIRM);
log_Log(TEST, TEST_INFO_OK, "Logging with three parameters.\0");
log_Log(TEST, TEST_INFO_CONFIRM, "Some message string.\0");
log_Log(TEST, TEST_INFO_OK, "Logging with four parameters.\0");
log_Log(TEST, TEST_INFO_CONFIRM, 16, testBuf);
log_Log(TEST, TEST_INFO_OK, "Logging with five parameters.\0");
log_Log(TEST, TEST_INFO_CONFIRM, "Some message string.\0", 16, testBuf);
```

*Figure 9. Typical calls to the logger*

The logger calls a private function, `log_send()`, when it is ready to send the data. This function constructs a log packet, serializes it, and sends it out to the host computer over a UART link. The packet definition is the same as that of the wifi module (intentionally, to make receiving wifi packets easier), but for the sake of completeness is given in *Table 6*.

---

Table 6. Log packet structure *log\_packet\_t*

Field	Data type	Description
<i>log_packet_mod</i>	<i>uint8_t</i>	Module ID
<i>log_packet_status</i>	<i>uint8_t</i>	Module status from <i>err.h</i>
<i>log_packet_msgLen</i>	<i>uint8_t</i>	Length of message in bytes
<i>log_packet_msg</i>	<i>uint8_t</i> *	Buffer containing '\0' terminated ASCII message
<i>log_packet_dataLen</i>	<i>uint32_t</i>	Length of data in bytes
<i>log_packet_data</i>	<i>uint8_t</i> *	Buffer containing data

It is through the logging interface (or wifi) that camera images get delivered. If a packet is sent with *log\_packet\_mod* equal to *CAM*, and *log\_packet\_status* equal to *CAM\_INFO\_IMAGE*, the host interface will interpret the data in this message as a raw image, saving and displaying the image.

#### Command module

The command module, or '*cmd*' as it is found in the code, handles commands both from the inside of the microcontroller and from the outside of the microcontroller. Its functionality is twofold - A user can call certain commands from the debug interface based on PC input, and the system can queue commands itself without user intervention. Commands from the host are sent to the STM32F429 via UART, buffered in a command queue, and eventually executed by the main command loop. Commands created within the STM32F429 will be directly put into the queue without user intervention. The idea here is that even without direct commands from the host interface, the system will be able to schedule events in an orderly manner based on interrupt service routines or other system events, and then just run the main loop to process them in an orderly fashion. *Figure 10* shows a good view of the command system through a system block diagram.

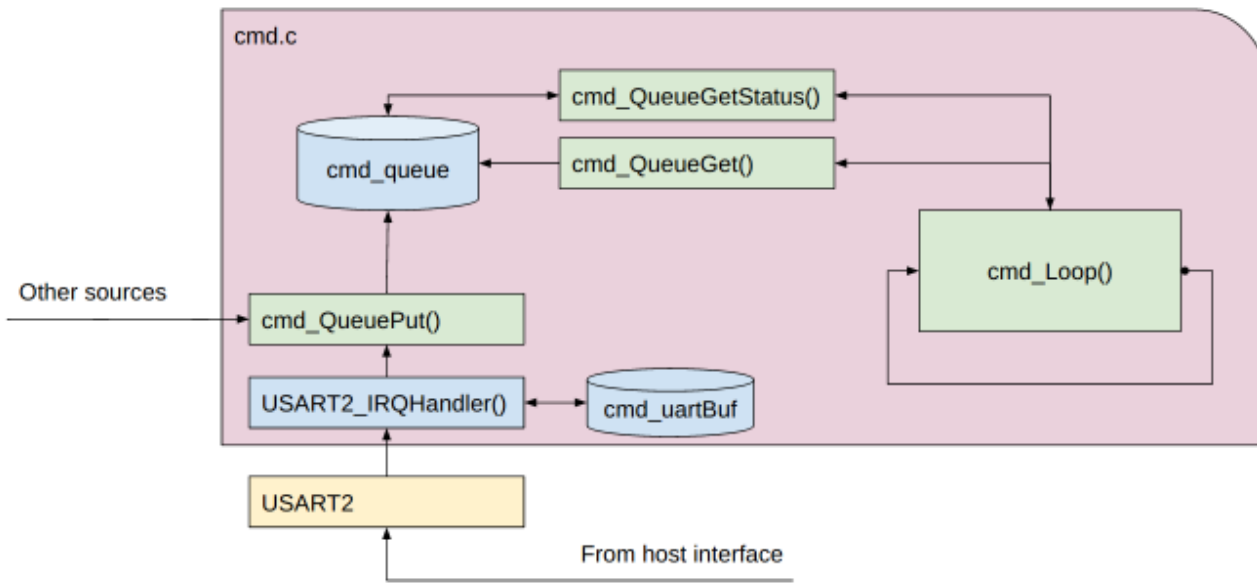


Figure 10. Command module structure

The command packet structure is similar to that of the log packets and wifi packets, but leaves less room for data and no room for messages. *Table 7* shows the command packet structure, and *Table 8* shows the command queue structure. For a full view of the command system implementation, go to the source code or Doxygen docs in the *doc/* directory.

Table 7. Command packet structure *cmd\_packet\_t*

Field	Data type	Description
<i>cmd_module</i>	<i>mod_t</i>	Module ID
<i>cmd_func</i>	<i>gen_func_t</i>	Function to execute on specified module
<i>cmd_dataLen</i>	<i>uint16_t</i>	Length of data in bytes
<i>cmd_data</i>	<i>uint8_t</i> *	Buffer containing data

---

Table 8. Command queue structure *cmd\_queue\_t*

Field	Data type	Description
<i>cmd_queue_buf</i>	<i>cmd_cmd_t</i> **	Base location of buffer
<i>cmd_queue_head</i>	<i>cmd_cmd_t</i> **	Location of buffer head
<i>cmd_queue_tail</i>	<i>cmd_cmd_t</i> **	Location of buffer tail
<i>cmd_queue_capacity</i>	<i>uint16_t</i>	Queue total capacity
<i>cmd_queue_size</i>	<i>uint16_t</i>	Queue current size
<i>cmd_queue_status</i>	<i>cmd_queue_state_t</i>	Current status of queue

Future implementations of the command system will likely be modified from this structure slightly. Particularly, it would be nice to have some sort of timing guarantee on tasks with a deadline, or at least a priority to commands so that more urgent commands can get processing time when they need it. It is likely that future command queue implementations will include the ability to push to both ends of the buffer, or a multi level priority scheme that can balance execution times more effectively.

## Profiler

The system also has a built in code profiler (*prof*) which, like the log module, uses a preprocessor macro to implement its functionality. The preprocessor macro basically wraps any chunk of code within its function call with calls to start and stop a STM32F429 timer. The profiler then uses the logging system to report back the results of the profile call with a status code *PROF\_INFO\_RESULTS*. Because of the way the profiler is implemented, the code can be littered with extra profile commands without issue. If you want to turn them off, simply disable the profiler or debug mode in the build system and the system ignores the calls entirely, simply calling the function or performing the statements within the profiler macro. Some typical calls to the profiler are shown in *Figure 11*.

---

```
// For a single function or statement
prof_Profile(xxx_functionToProfile(), "Benchmark identifier\0");

// Several statements or functions
prof_Profile(
    xxx_functionToProfile1();
    xxx_functionToProfile2();
    uint32_t someVariable = 100;
    // ...etc
    , "Benchmark identifier\0"
);
```

*Figure 11. Typical calls to the profiler*

The profiler has a resolution of 1 us, with a maximum time of 100 seconds (we are using a peripheral timer that can capture 32 bits). If the underlying timer needs to be changed to a 16 bit timer, we will have to keep track of the number of timer overflows with an interrupt service routine.

#### Test

The test framework is the final module, and it's a bit of a special one. As stated earlier, the test functions have their own folders in *inc/test* and *src/test*, because they a lot of test functionality. The test framework is technically a software module as defined in *mod.h*, but will be described in a separate section after the error system.

## STATUS AND ERRORS

The system has a robust status and error management system. In the file *inc/project/err.h*, there are status codes defined for each module at three different levels - *INFO*, *WARN*, and *ERR*. When a new module is created or new functionality is added to an old module, the status types in the *err.h* file can be modified accordingly.

Within a module, all functions return a status code of the type associated with that module. That way, when you call a function from a specific module, you will always know what type the return type will be - Speeding up programming time and frustration. If an error does occur, you can handle it easily in the calling function, by logging the error to the console, trying

---

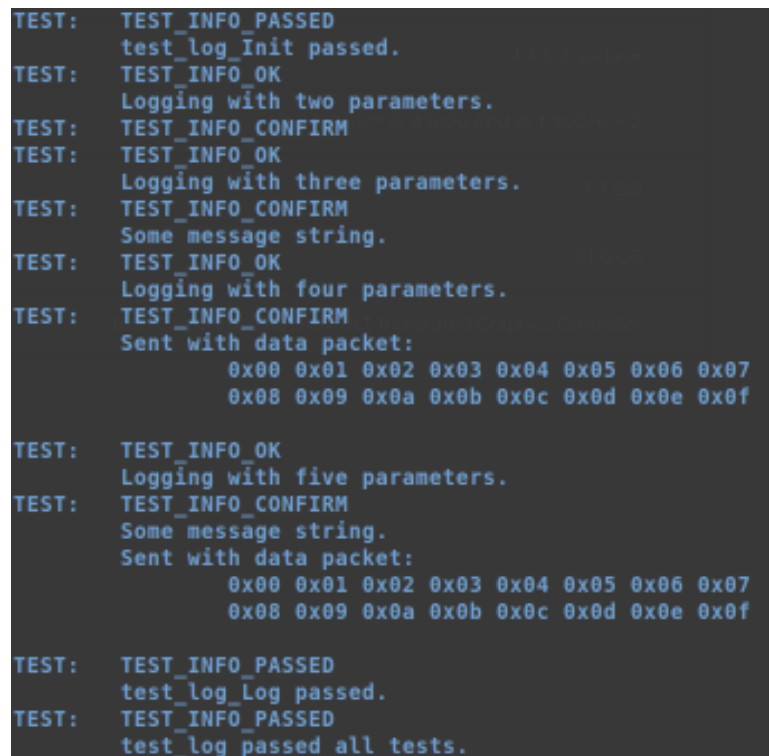
again with different parameters, or failing silently if the error is not important in that case. If you look *src/project/main.c*, you will find good examples of how to handle status codes at the top level.

The status codes are also used in the logger and the wifi module. This allows the user to log with more structure and more consistency with the code - A message from the logger saying *PROF\_WARN\_ALINIT* means that the profiler couldn't be initialized as it was already initialized. By matching logging messages with error codes, we create consistency across the project and an easy environment to debug in.

## TEST FRAMEWORK

Finally, the test framework allows the user to set up and run basic unit tests before and during their application. The test framework is managed in two places. First, there is *inc/test/test.h*, which holds declarations for all test functions (organized by modules), and preprocessor macros that form the body of the test framework. This test framework is based on a tiny code snippet called MinUnit<sup>9</sup> - and adapted to work well with the logging system.

The macros in the header file define an assert function, which evaluates a condition and returns a message if the condition fails, as well as a test function, which takes the name of a test to run and runs it. If the test passes, the test system logs *TEST\_INFO\_PASSED* with the test pass message, and if the test fails, logs *TEST\_WARN\_FAILED* with a specific failure message. Note that in this case a warning is used, since the test module is not actually experiencing an error, but is warning the user that the test has failed. This could mean that other modules will give the user errors, but the test system simply presents a warning. For a view of the test system in action, see *Figure 12*.



```
TEST: TEST_INFO_PASSED
      test_log_Init passed.
TEST: TEST_INFO_OK
      Logging with two parameters.
TEST: TEST_INFO_CONFIRM
TEST: TEST_INFO_OK
      Logging with three parameters.
TEST: TEST_INFO_CONFIRM
      Some message string.
TEST: TEST_INFO_OK
      Logging with four parameters.
TEST: TEST_INFO_CONFIRM
      Sent with data packet:
           0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
           0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f

TEST: TEST_INFO_OK
      Logging with five parameters.
TEST: TEST_INFO_CONFIRM
      Some message string.
      Sent with data packet:
           0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
           0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f

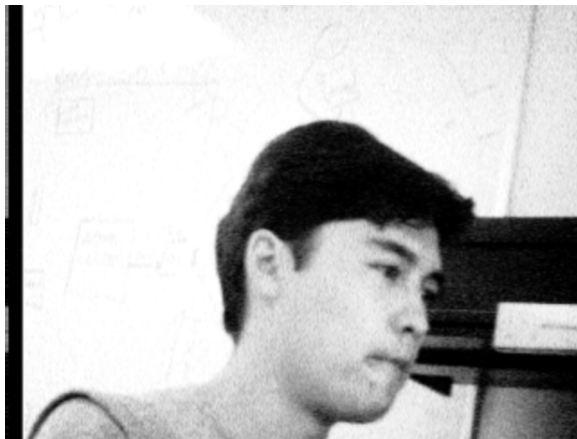
TEST: TEST_INFO_PASSED
      test_log_Log passed.
TEST: TEST_INFO_PASSED
      test_log passed all tests.
```

*Figure 12. Running the log test function*

---

## DISCUSSION

The ultimate goals of this project were to create an interface for both a camera module and a wifi module that was robust, hierarchical, organized, and modular. In these goals the project was a success. The OV7670 camera is able to be initialized, configured, and can capture and transfer an image to the computer. The next few figures show some successful examples.



*Figure 13. Capstone partner working hard*



*Figure 14. Friends mugging for the camera*

```
cam config
CAM: CAM_INFO_OK
    Configured camera module.
cam capture
CAM: CAM_INFO_OK
    Captured an image into SDRAM.
cam transfer
CAM: CAM_INFO_OK
    Beginning image transfer.
CAM: CAM_INFO_IMAGE
    Saved image to data folder.
    Processing image for display.
    Displaying image.
```

*Figure 15. A successful image transaction on the host interface*



---

The few caveats to this project come from the wifi module. The most successful image transfers occur when the image is transferred via the UART host interface. The wifi module seems to not be able to match the performance of the direct UART link, which makes sense. The data path from the wifi module to the computer is orders of magnitude more complex, and the wifi module may not be able to supply a enough transmission power to make the transfer effective.

In order to get the wifi module to transmit reliably, the UART speed on the connection needs to be slowed down to the 9600 Baud range. This is in stark contrast to the direct link to the host, which can be clocked at speeds up to 921600 Baud on the right computer (using a USB hub slows the system down to about 115200 Baud for reliable transfers). If this wifi module is to be used, work will have to be done to get the speed up and reliable.

Additionally, there were issues with the integrity of the data over wifi. Sometimes, data would be missing on the host interface side, lost somewhere in the TCP transfer. With limited networking experience, it is hard to say where or why that happened, but it may be beneficial to explore more of a transactional scheme for sending data, where we can be sure a data packet is successfully sent and acknowledged before sending an additional packet.

A few known bugs exist in the software as well. The log handler on the host interface side sometimes times out when receiving data. It is an issue with how the reset logic is calculated, and the immediate fix is not clear. The issue isn't very pressing, the worst case situation is a missed log message, and generally the logger just displays a reset warning. Given the chance, this will be fixed. Additionally, some logic should be implemented to handle the power to the camera and wifi modules. If the camera module gets a clock signal before it receives power, for example, it will not configure correctly. This led to some serious frustration at what appeared to be a sporadic bug. Finally, the gdb connection is occasionally unreliable and is only fixed by a full power cycle on the board. The debug/program part of the host interface could definitely use some polishing.

Overall, however, the project was a success. Many modules were created in a highly modular and hierarchical design, with robust logging, commanding, and error handling functionality. Going forward with a strong software foundation will help enormously in future revisions of the design, and through all the trials of the code creation and debug cycle.

Please explore the source code on Github and the documentation in the *doc/* folder.

---

## REFERENCES

1. GNU ARM Embedded Toolchain  
<https://launchpad.net/gcc-arm-embedded>
2. Open source version of STMicroelectronics ST-LINK tools  
<https://github.com/texane/stlink>
3. Lightweight byte FIFO for Python  
<https://github.com/hbock/byte-fifo>
4. The Go Blog - Package names  
<https://blog.golang.org/package-names>
5. STMCube  
<http://www.st.com/en/embedded-software/stm32cube-embedded-software.html>
6. NodeMCU  
[http://www.nodemcu.com/index\\_en.html](http://www.nodemcu.com/index_en.html)
7. Serial Camera Control Bus (SCCB) Function Specification  
[http://www.ovt.com/download\\_document.php?type=document&DID=63](http://www.ovt.com/download_document.php?type=document&DID=63)
8. Arducam website and codebase  
<http://www.arducam.com/>  
<https://github.com/ArduCAM>
9. MinUnit -- a minimal unit testing framework for C  
<http://www.jera.com/techinfo/jtns/jtn002.html>