Cysat Platform API Documentation

Jake Drahos
drahos@iastate.edu

January 5, 2015
v0.1 (DRAFT)

Contents

1 Introduction

This document serves two purposes. The first is to detail the functionality provided by the CySat platform and describe the APIs to access that functionality.

The second purpose of this document is to fully explain the layer of connection between the payload handling thread and the rest of the CySat platform, namely the communications/downlink thread. This document will act as a guide for payload integration, detailing the interfaces that the payload thread should use to extract data from the payload board, and how the payload thread should communicate this data to the communications thread.

This document is currently a working draft. Proposals will be taken into account and adjustments made. Also note that, until this document leaves draft status, the API documentation is not intended to be an exhaustive or canonical reference. Rather, it is intended to give an idea of how the APIs work. As development proceeds, new APIs can and will be added to expose additional functionality, but these APIs will operate in much the same way as those documented here.

2 Pin and Peripheral Mapping

SECTION UNDER CONSTRUCTION - PENDING FINAL STM32 MCU MODULE DESIGN

This section details which ports and pins map to which CubeSatKit IO lines, as well as which STM32 peripherals are used for the CubeSatKit standard interfaces, such as System I2C, System SPI, and UART1/2.

3 Hardware API

There is not a full true hardware abstraction layer, but hardware should be accessed through these API functions if available, rather than through manual direct register manipulation or through the ST Peripheral Library. This API contains RTOS wrapping and wrapping to go from CubeSatKit naming conventions to the actual peripherals.

If a hardware function does not have APIs documented here, it is likely accessible directly. If in doubt, check the code. GPIOs will always be directly accessible, but check to see if the pins are in use.

3.1 Serial Peripheral Interface (SPI)

The SPI peripherals are wrapped in a DMA and RTOS layer to allow the current task to block while transferring, yet other tasks to run in the meantime. Mutexes are used to avoid peripheral conflicts. It is conceivable to take the mutex for a SPI peripheral and then directly access it, but that is not recommended.

The SPI API functions refer to SPIx, where x is a number. These numbers correlate to the STM32 peripheral names, not the CubeSatKit bus names. See the above section "Pin and Peripheral Mapping" to associate STM32 peripherals with CubeSatKit bus interfaces.

See the CySat_SPI module in the Doxygen-generated documentation for detailed information.

3.2 Inter-Inegrated Circuit (I2C)

The I2C API is very similar to the SPI API, however it features only one buffer and separate functions for reads and writes.

The numbered I2Cx peripherals are based on the STM32 name, not CubeSatKit names.

See the CySat_I2C module in the Doxygen-generated documentation for detailed information.

3.3 UART/Serial

The UARTx in the UART API follow CubeSatKit conventions. UART1 is the console UART, and UART2 is the Radio UART. The functions are all rather self-explanatory. vConsole(Put/Print/Printf) should be used for console printfs, reserving explicit UART1 and UART2 access for situations where the distinction is significant.

See the CySat_UART module of the Doxygen-generated documentation for detailed information.

3.4 Mission Clock

getMissionTime(): returns a time_t of the mission time (seconds since first power-on).

3.5 Nonvolatile Storage (State Flags)

Nonvolatile Storage is available through the RTC's backup registers. Ideally, the payload thread should be stateless or read any state information from the SD card. Additionally, the backup registers are extremely limited, as only 20 4-byte registers are available. For these reasons, no API is provided at this time. However, should this functionality become necessary, functions will be added. These functions will be named [name]BackupRegisterRead() and [name]BackupRegisterWrite(), where [name] is based on the subsystem that will use the backup register. Backup registers will be added by name, rather than address or offset. This will discourage usage creep, which is a major issue due to the extremely limited availability of the backup registers.

Access functions will be implemented on an as-needed basis and documented in the Doxygen-generated documentation.

4 Platform-Payload Interface

This section is a proposal by the Iowa State team, not a final set of requirements.

All payload interface will be handled by a single subsystem, referred to as the "payload thread." As such, this section largely describes the interface between the payload thread and the rest of the platform threads.

This section will undergo heavy revision during the draft/feedback process.

4.1 Payload Thread Scope

The payload thread will have the following responsibilities:

- Filesystem.
- SD Card access arbitration.
- Image selection.
- Payload board initialization and management.

Table 1: Globally accessible Payload-Platform Interface variables

Variable	Brief description
uint8_t[] imageSlotA	First 64K image buffer
uint8_t[] imageSlotB	Second 64K image buffer
uint32_t imageSizeA	Image A size in bytes. 0 for not loaded
uint32_t imageSizeB	Image B size in bytes. 0 for not loaded
uint32_t imagesOnSDCard	Number of images on SD card. Periodically updated
payloadStatus_t payloadStatus	Enum for current payload status (active/inactive)
resolution_t currentResolution	Enum for resolution.
captureFrequency_t currentCaptureFrequency	Enum for capture frequency.
xQueueHandle payloadCommandQueue	Queue of commands sent to payload thread
uint32_t errorFlags	Bit mask of error flags

- Presentation of images to the communications thread.

The filesystem will be entirely the domain of the payload thread. It will not be used for any other purpose. As such, the filesystem implementation is completely internal to the payload thread. However, the payload thread must use the platform APIs (SPI or SDIO) for SD card access to ensure that other threads will not be blocked during large reads or writes and to avoid conflicts with other external hardware that may be on the same bus.

SD card access arbitration will be done by the payload thread using direct access to the GPIOs. The system I2C bus can also be used for arbitration.

Image selection will be handled by the payload thread. This process must be non-volatile, ie. if there is a power loss, the same images must be re-loaded upon restart. Ideally this will be handled by the flash filesystem, however access can be granted to the backup registers if this would simplify the process.

The payload thread will initialize the payload board as part of its initialization. Initialization will be done upon startup, however the payload thread and payload board should be initialized into a low-power, inactive state. The payload thread and payload board should be able to be activated and deactivated many times throughout the mission timeline.

The largest responsibility of the payload thread will be to load images from the SD card ard present them to the communications thread for downlinking. This is described in detail below.

4.2 Image Presentation

Two image ”slots”, tentatively 64KB in size, would be statically allocated, along with a number of flags used to communicate state from the payload thread to the communications thread. These slots, imageSlotA and imageSlotB, would be populated by the payload thread and read by the communications thread. Upon command from the communications thread, the payload thread would ”flush” the specified slot, deleting that image from the SD card and then proceeding to load a new image into its place.

As stated above, a number of flags will communicate state from the payload thread to the communications thread. The payload thread will receive from a queue to receive commands from the communications thread. These commands would include the ”flush” command above, as well as activate/deactivate, purge, and miscellaneous configuration information that should be forwarded to the payload board or otherwise handled appropriately.

All of the above variables, with the exception of the queue, will be written only by the payload thread. To enforce this, a number of getters could be used and the variables could be made non-global. If this is done, the getter naming scheme would be getVariableName, where VariableName is the name given above with the first letter capitalized. Getters for the image slots are obviously impractical.

4.3 Flow

Upon initialization, the payload thread should set all flags and configuration settings to sane defaults.

Upon initialization or deactivation, the payload thread should ensure that the payload board is in a low-power mode. While inactive, the payload thread should still properly handle any configuration changing commands.

Upon receiving a command to activate, the payload board should be powered on (or brought out of low-power mode) and the payload thread should begin loading images into the slots as soon as they become available. Images should have some sort of intrinsic priority (such as timestamp) and the highest priority image should be loaded as soon as a slot is available. Once an

Table 2: Non-exhaustive list of commands

Command	Brief description
activate	Turn on payload board and begin populating image slots if possible
deactivate	Turn off payload board and immediately cease population of image slots
purge	Delete all images from SD card. Ignored while active
flushSlotA	Flush slot A and delete from SD card
flushSlotB	Flush slot B and delete from SD card
setResolutionLow	Set resolution
setResolutionMed	Set resolution
setResolutionHigh	Set resolution
setCaptureFrequencyHighest	Set frequency at which images are taken
setCaptureFrequencyHigh	Set frequency at which images are taken
setCaptureFrequencyMedium	Set frequency at which images are taken
setCaptureFrequencyLow	Set frequency at which images are taken
setCaptureFrequencyLowest	Set frequency at which images are taken
setCaptureFrequencyNone	Stop capturing images. Put payload board into low-power mode, but do not deactivate.

image is loaded into imageSlotA, imageSizeA should be set. Once another image becomes available, the payload thread should load it into imageSlotB, then set the size for imageSlotB.

When the payload thread receives a configuration change command, such as resolution or capturefrequency, it should set the appropriate global flags, and forward the command to the payload board if appropriate. Depending on the implementation of ”low-power mode” for the payload board, this might not be possible while inactive. Should that be the case, any configuration information must be copied to the payload board upon activation.

Upon reception of a flush command, the payload thread should delete the image from the SD card, then set the image size to 0. At this point the slot is considered empty. The highest priority image (that is not already loaded into the other slot) should be loaded into this slot once available.