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. Some hardware functionality is directly accessible.

The Doxygen-generated reference documentation is a companion to this section.

3.1 General Purpose IO (GPIO)

Since GPIO access is rather fundamental, no wrapping or driver will exist. Conflict resolution will be handled via a portmap/pin assignment spreadsheet.

3.1.1 Driver API

None. Use ST Peripheral Library or direct register manipulation.

3.2 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. For this reason, it is very important to use the provided driver to avoid unnecessary blocking or major loss of throughput due to context switching.

3.2.1 Driver API

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

3.2.2 Numbering Scheme

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.

3.3 Inter-Inegrated Circuit (I2C)

The I2C API is very similar to the SPI API.

3.3.1 Driver API

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

3.3.2 Numbering Scheme

The numbered I2Cx peripherals use the STM32 name, not CubeSatKit names.

3.4 USART/Serial

UART access is handled through the UART Rx and Tx threads. Queues act as buffers in a manner similar to many desktop operating systems, except that there are no per-thread buffers.

3.4.1 Driver API

Documented in the CySat_UART module of the Doxygen documentation

3.4.2 Numbering Scheme

The UARTx in the UART API follow CubeSatKit conventions. UART1 is the console UART and should only be using to print debugging information. UART2 is the radio, and should not be directly accessed. There is also a set of UART functions prefixed “vConsole”. These functions should be preferred for console debug and informational messages.

3.5 Real-Time Clock

The real-time clock is wrapped to provide a mission clock.

3.5.1 Driver API

Documented in the CySat_Clock module of the Doxygen documentation.

3.6 Backup Registers

The STM32 RTC peripheral contains a real-time clock as well as backup registers that exist in the backup power domain. The backup registers are used as nonvolatile state flags. Since the availability of backup registers is extremely limited, there is no public API. Additionally, only the initialization thread should have a persistent state, as well as some specific configuration data should persist, so for design reasons access to the backup registers is limited. Should a need for the backup registers arise, they can be allocated to the payload thread, and direct access will be used.

3.6.1 Driver API

None. Use should be limited for design reasons as well as limited resources. Direct access will be used in situations where persistence is deemed necessary and registers can be spared.

3.7 Reset and Clock Control

The RCC peripheral of the STM32 is largely responsible for power management and enabling/disabling of specific peripherals. The RCC should not be used except by drivers, or when the payload thread is granted direct access, without contention, to an entire peripheral. Any peripheral noted as "not utilized" can be monopolized by the payload thread. In this case, the payload thread is free to power the monopolized peripheral on or off as needed.

3.7.1 Enabling Peripherals

Any peripheral that is directly accessed should have its corresponding RCC Enable bit set as part of initialization. However, do not disable the RCC Enable bit as part of deinitialization. This is especially true of GPIOs. If a GPIO port will be used, enable it in the RCC as part of initialization, but never disable a GPIO port in the RCC.

3.8 DMA Controller

The DMA controller is heavily leveraged by the I2C and SPI drivers, but it may come into use for other threads. In that case, a DMA stream may be allocated to that thread.

3.8.1 Driver API

None. Direct access is used.

3.8.2 DMA Allocation

DMA Streams Allocated to Drivers	DMA2_Stream2, DMA2_Stream5, DMA1_Stream5, DMA1_Stream6

3.9 ADC

The ADC is not currently utilized by the CySat platform. It can currently be directly accessed by any thread.

3.9.1 Driver API

None. The ADC is not currently used.

3.10 DAC

The DAC is not currently utilized by the CySat platform. It can currently be directly accessed by any thread.

3.10.1 Driver API

None. The DAC is not currently used.

3.11 Timers

The hardware timers should not be used. RTOS Task Delays and timers should be used instead.

3.12 Hash Processor

The cryptographic processor is used to validate signatures for sensitive commands. Since it is a rather unique use-case peripheral, direct access will be used, but it will be protected by a RTOS mutex.

3.12.1 Driver API

Mutex protection documented in Doxygen. Direct access for use.

3.13 Cryptographic Processor

The cryptographic processor is not currently utilized.

3.13.1 Driver API

None. The cryptographic processor is not utilized.

3.14 Random Number Generator

The RNG is not currently utilized.

3.14.1 Driver API

None. The RNG is not currently utilized.

3.15 SDIO Interface

The SDIO interface is an alternative to the SPI interface for SD card access. Should SDIO be used, it will be implemented as a subset of the SPI driver.

3.16 Driver API

Not currently implemented, but would be implemented as a sub-feature of the SPI driver.

3.17 Flexible Memory Controller (FMC) and Flexible Static Memory Controller

The FSMC is used as part of the housekeeping logging thread. It should not be used by the payload thread. If the payload thread has housekeeping data that should be logged, an interface to the housekeeping thread can be developed.

3.17.1 Driver API

Should not be accessed by payload thread. If the payload thread has housekeeping data, an interface to the housekeeping thread can be created.

3.18 USB on-the-go

Not implemented and should not be used.

3.19 CAN

Not implemented and should not be used.

3.20 Ethernet

Not implemented and should not be used.

4 Platform-Payload Interface

4.1 Proposal

This section is a proposal by the Iowa State team, not a final set of requirements. This section can be expected to undergo heavy revision until this is no longer a draft or a proposal.

4.2 Payload Thread

All payload interface will be handled by a single subsystem, referred to as the "payload thread." As such, the payload-platform interface can be abstracted to an interface between the payload thread and several other threads, namely the communications thread and initialization thread.

4.3 Payload Thread Scope and Responsibilities

The payload thread will have the following responsibilities:

- Filesystem.
- SD Card access arbitration.
- Image selection.
- Payload board initialization and management.
- Presentation of images to the communications thread.

4.3.1 Filesystem

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.

4.3.2 SD Card Access Arbitration

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

4.3.3 Image Selection

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.

4.3.4 Payload board initialization, configuration, and management

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.

4.3.5 Image loading and presentation

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

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

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.

4.4 Image Presentation

4.4.1 Image Slots

Two image "slot", 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.

4.4.2 Other channels of communication

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.

4.4.3 Access to variables and naming scheme

All of the "global" variables, with the exception of the queue, will be written only by the payload thread. To enforce this, a number of getter functions will be used and the variables will be made non-global. If this is done, the getter naming scheme would be PAYLOAD_get[VariableName](). Access to the queue will be handled with PAYLOAD_QueueSend(). Access within the payload thread can be handled however is most conveniont.

4.5 Normal Operation

4.5.1 Initialization

Upon initialization, the payload thread should set all flags and configuration settings to sane defaults. No persistent state should be taken into account. Upon a power loss and reboot, the payload thread and board should be in an inactive state with sane default settings.

4.5.2 Inactive State

While inactive, the payload thread should still properly handle any configuration changing commands, but the payload board should remain in a low power state.

4.5.3 Activation

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 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.

4.6 Configuration Changes

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.

4.6.1 Flushing and Replacing Images

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.

4.6.2 Summary

The payload thread initializes into a low-power inactive state. It can be activated and deactivatied. Upon deactivation, settings are kept. Settings are reset upon activation. The payload thread maintains two images loaded into the two slots. Upon a flush command, the image is purged from the SD card, and the slot can be replaced.