Cysat Platform API and Payload Interface Documentation

Jake Drahos
drahos@iastate.edu

February 25, 2015
v0.3 (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 must use to extract data from the payload board, and how the payload thread will 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 must 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 will have a persistent state, as well as some specific configuration data that must 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 must 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.

3.15.1 Driver API

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

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

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

3.16.1 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.

3.17 USB on-the-go

Not implemented and should not be used.

3.18 CAN

Not implemented and should not be used.

3.19 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. It is based on the current understanding of the payload's requirements and capabilities, as well as the current state and capabilities of the CySat platform. 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 must be initialized into a low-power, inactive state. The payload thread and payload board must 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.

4.4 Communication Between Threads

The full interface is defined in payload.h. See the header file for details.

4.4.1 Signal Queues

Two queues exist and pass signals between the payload thread and the communications thread. The payload thread calls ReceiveCommunicationsEvent() periodically to check for CommunicationsEvents from the communications thread. The payload thread calls PostPayloadEvent() to post PayloadEvents to the Communications thread. The events are of the type PayloadEvent_t and CommunicationsEvent_t. Each event has a signal and a 32-bit integer attribute. The signal types are defined in payload.h, as well as the contextual meaning of the attribute integer.

4.4.2 Image Slots

A number of image "slots", 64KB in size, are statically allocated, along with a number of flags used to communicate state from the payload thread to the communications thread. These slots will be populated by the payload thread and read by the communications thread. Upon command from the communications thread, the payload thread must "flush" the specified slot, deleting that image from the SD card and then proceeding to load a new image into its place. If an image is over 64KB in size, it must be discarded, preferably by the payload board. The payload thread must not present an image over 64KB in size to the communications thread.

The payload thread must post the appropriate signals upon any image actions as well as keep the value of GetImagesOnSDCard() up to date as images are loaded and flushed.

4.4.3 Payload Configuration Settings

Note: The examples in this section are a placeholder to demonstrate how the settings system works. The exact settings available and meaning of different values must be finalized by project leadership

Configuration change requests are sent as CommunicationsEvents with the appropriate signal type and relevant attribute set to an enum value. The current value of a configuration setting is read by the payload thread by calling functions such as GetPayloadResolution() or GetPayloadCaptureFrequency(). These functions read the global configuration state stored by the payload and return it.

4.4.4 Global Variables and Functions

Image slots are globally accessible as an array of ImageSlot_t structs. The event queues should not be directly accessed; call the Receive and Post functions in the header file. RTOS Queues are pass-by-copy.

See payload.h for more details.

4.5 State Definitions

There are three states: uninitialized, inactive, and active.

4.5.1 Uninitialized

The payload board begins in the uninitialized state until initialization. Variables are undefined, and the payload board is in an undefined/unknown state. The uninitialized state can be the result of a complete power loss or a graceful reboot, so no assumptions as to the state of the payload board can be made.

4.5.2 Inactive State

The payload board is in the low-power mode during the inactive state. The payload thread still responds to any configuration changes and polling as appropriate. Any configuration change commands are applied to the global configuration state, and any reads of the global configuration state take any such commands into account. However, the payload board remains in the low-power mode and no SD card operations are performed until the active state is entered. The inactive state is considered the safe state for satellite reboots with regards to the payload board and SD card.

4.5.3 Active State

In the active state, the payload board is taking pictures and the payload thread is loading images into the SD card. The payload board is loaded with the current global configuration state upon activation, and any configuration change commands received when in the active state are immediately applied to both the global state and the payload board. In order to obtain a "pseudostate" where the payload thread loads images, but the payload board does not capture new images, enter the active state with the image capture frequency set to a "none" or "disabled" seting.

4.6 Normal Operation

4.6.1 Initialization

Upon initialization, the payload thread must 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 must be in an inactive state with sane default settings. As part of initialization, the payload thread must ensure that the payload board is in the low-power state.

4.6.2 Activation

Upon receiving a command to activate, the payload board must be brought out of low-power mode. As part of being brought out of low-power mode, the payload board should be loaded with the current global configuration state. The payload thread should now 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. This priority system is defined by the payload thread and the payload board, and has no influence from the rest of the satellite systems. Once an image is loaded into imageSlotA, imageSizeA must be set. Once another image becomes available, the payload thread should load it into imageSlotB, then set the size for imageSlotB. Images remain in the slot until removed by a flushImage command.

4.6.3 Deactivation

The payload board is put into low-power mode, pending SD card operation. Image loading must immediately cease, but SD card operations are allowed to continue until the filesystem is in a stable, non-corrupt state. The payload thread must not leave the filesystem in a corrupt state.

4.6.4 Configuration Changes and the Global Configuration State

The global configuration state is maintained in main memory at all times. This state is initialized to sane defaults at initialization, modified as necessary at any time, and copied to the payload board upon activation. The global configuration state is not in any specific format, rather it is a collection of variables accessed by a number of "get-" functions. It does not need to be copied in binary format to the payload board, but should accurately reflect the functionality of the payload board at all times, ie. for any given set of values, the payload board should behave in a consistent manner. When the payload thread receives a configuration change command, such as resolution or capturefrequency, it must modify the global configuration state as appropriate, and, if the payload is active, perform the necessary operations to update the payload board configuration.

4.6.5 Flushing and Replacing Images

Upon reception of a flush command, the payload thread must first set the appropriate image size variable to zero, then delete the image from the SD card. 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, replacing the current buffer contents. Once a new image is fully loaded into the slot, the appropriate imageSize variable must be set.