PixelSat I Software

Aadish Verma, Ashwin Naren, Vinayak Vikram
[Project Pixel Orbital, Stanford Online High School]

Contents

Introduction

Project Pixel Orbital is a student-led team at Stanford Online High School, aiming to launch a 3U CubeSat (PixelSat I) to sun-synchronous low-Earth orbit NET November 2026. In this paper, we document the PixelSat I software stack.

The PixelSat I runs on a STM32H753 ARM Cortex-M7 microcontroller. It is a 32-bit 480MHz MCU that has 1 MB of SRAM and 2 MB flash. Additionally it comes with builtin ECC and crypto acceleration (including AES and CRC).

In terms of built-in peripherals, we mainly rely on the RTC, the internal watchdog, and the second internal flash bank used for non-volatile storage.

At present, resilience is provided by the STM32's internal independent watchdog, which is configured in firmware and fed by a dedicated RTIC task. We are no longer planning a separate external watchdog IC.

Rather than a dedicated external flash chip, persistent storage currently lives in the STM32's second internal flash bank. This is used for things like image metadata, stored image data, and TLE storage.

In addition the many sensors aboard the cubesat are wired into the STM32.

The STM32 is programmed in Rust via the `stm32h7xx_hal` crate.

The main operations of the satellite are:

- ADCS
- Taking and downlinking an image
- Sending a heartbeat every minute

ADCS

The ADCS (attitude determination and control) subsystem is one of the mission-critical subsystems running on the primary onboard flight computer. The ADCS subsystem aims to estimate the satellite's current orientation and body rates (ADS, attitude determination system) and then command actuators to change that attitude as needed (ACS, attitude control system).

> **Caution**
>
> We output torques instead of dipole moments, some torques might not be able to be fully applied.

Our inputs are as follows:

- Coarse Sun Sensor (CSS): six dedicated photodiodes, one on each body face (+X, -X, +Y, -Y, +Z, -Z)
- Magnetometer (PNI RM3100-CB)
- IMU gyroscope (currently used by the estimator)
- Accelerometer data is available from the IMU, but it is not currently fused by the present flight estimator

Our outputs are:

- Estimated attitude state for the rest of the flight software
- Commanded body torque

Terminology

- MRP: Modified Rodrigues Parameters
- EKF: Extended Kalman Filter
- PID: Proportional Integral Derivative
- CSS: Coarse Sun Sensor
- DCM: Direction Cosine Matrix
- ECI: Earth-Centered Inertial

Attitude Determination System

Black-box overview

Our inputs are as stated above, and we aim to produce the following information:

- attitude (reported as MRPs)
- angular velocity
- gyro bias

The third parameter is not directly fed into the ACS law, but it is important because the angular-velocity estimate is formed from the measured gyro rate minus the estimated bias.

In the software, both the magnetometer and sun sensor are treated as vector observations: a body-frame measurement is compared against a reference vector expressed in an inertial frame. Those inertial reference vectors come from elsewhere in the overall ADCS stack, e.g. orbital knowledge plus sun and geomagnetic models.

Theory

In order to properly account for variance in sensor readings and to combine the drifting relative sensor (gyro) with absolute vector measurements (sun sensor and magnetometer),

we use an EKF. More specifically, the current implementation is a multiplicative error-state EKF: the nominal attitude is propagated non-linearly, while the filter covariance tracks small attitude and gyro-bias errors.

The Extended Kalman Filter (EKF) is the nonlinear version of the Kalman filter, which linearizes about the current estimate of the mean and covariance. Simply put, by aggregating previous sensor readings, Kalman filters aim to produce more accurate predictions than relying on a single reading alone. The sensors we work with are the following:

- Gyro: relative, high-rate, and very useful on short timescales, but drifts over time and therefore needs bias estimation
- Magnetometer: absolute vector reference, but measurements may need to be blanked or omitted while magnetorquers are actively firing
- Coarse sun sensor: lower accuracy than the gyro, but provides another absolute vector whenever the sun is visible. The current software interface assumes six dedicated face photodiode measurements (+X, -X, +Y, -Y, +Z, -Z) that are combined into a normalized sun vector in the body frame

A standard nonlinear state-estimation problem is

$$x_{k+1} = f(x_k, u_k) + w_k, \quad w_k \sim \mathcal{N}(0, Q_k)$$

$$z_k = h(x_k) + v_k, \quad v_k \sim \mathcal{N}(0, R_k)$$

where:

- x_k is the hidden state,
- u_k is the known input,
- z_k is the sensor measurement,
- $f(\cdot)$ is the nonlinear process model,
- $h(\cdot)$ is the nonlinear measurement model.

The EKF works by linearizing these nonlinear functions around the current estimate.

Prediction step:

$\hat{x}_k^- = f\left(\hat{x}_{\{k-1\}}^+, u_k\right)$

$F_k = \frac{\partial f}{\partial x} \mid_{(\hat{x}_{k-1}^+, u_k)}$

$P_k^- = F_k P_{\{k-1\}}^+ F_k^t \texttt{op} + Q_k$

Update step:

Given a measurement z_k:

$y_k = z_k - h(\hat{x}_k^-)$

$H_k = \frac{\partial h}{\partial x} \mid_{(\hat{x}_k^-)}$

$S_k = H_k P_k^- H_k^t \texttt{op} + R_k$

$K_k = P_k^- H_k^t \texttt{op}\ S_k^{-1}$

$\hat{x}_k^+ = \hat{x}_k^- + K_k y_k$

$P_k^+ = (I - K_k H_k) P_k^- (I - K_k H_k)^t \texttt{op} + K_k R_k K_k^t \texttt{op}$

That last covariance update is the Joseph form, and we use it for numerical stability.

Implementation

Our current estimator uses a multiplicative error-state EKF. Internally, attitude is propagated as a unit quaternion, while the external interface reports attitude as shadow-set MRPs. The nominal state is therefore best thought of as

$$q_{\mathrm{BN}} \in H, \quad b_g \in \mathbb{R}^3, \quad \hat{\omega}^B_{\mathrm{BN}} = \omega_{\mathrm{gyro}} - b_g$$

and the covariance is carried on the 6D error state

$$\delta x = \begin{pmatrix} \delta\theta \\ \delta b_g \end{pmatrix} \in \mathbb{R}^6$$

where:

- q_{BN}: spacecraft attitude, using the passive body-from-inertial convention,
- $b_g \in \mathbb{R}^3$: gyro bias,
- $\hat{\omega}^B_{\mathrm{BN}} \in \mathbb{R}^3$: debiased body angular velocity,
- $\delta\theta \in \mathbb{R}^3$: small attitude error used by the EKF linearization,
- $\delta b_g \in \mathbb{R}^3$: gyro-bias error.

MRPs are still the main attitude representation exposed to the rest of the subsystem, because they provide a compact 3-parameter interface. The associated direction cosine matrix is

$$C_{\mathrm{BN}}(\sigma) = I + \frac{8[\sigma_x]^2 - 4(1 - \sigma^T\sigma)[\sigma_x]}{(1 + \sigma^T\sigma)^2}$$

where $[\sigma_x]$ is the skew-symmetric cross-product matrix.

Because MRPs have a singularity at 360 degrees, the reported MRPs are always mapped into the shadow set:

$$\sigma^{\mathrm{shadow}} = \begin{cases} [\sigma |\sigma|^2 \le 1] \\ \left[-\frac{\sigma}{|\sigma|^2} |\sigma|^2 > 1\right] \end{cases}$$

Process model

The gyro measurement drives the propagation step. The debiased angular velocity is

$$\hat{\omega}^B_{\mathrm{BN}} = \omega_{\mathrm{gyro}} - b_g$$

and the attitude is propagated internally with a quaternion increment corresponding to $-\hat{\omega}\Delta t$. The gyro bias is modeled as slowly varying, i.e. approximately a random walk rather than a fast-changing state.

Measurement model

The current filter uses two reference-vector updates:

$$h_{\mathrm{sun}}(q) = C_{\mathrm{BN}}(q) s_N$$

$$h_{\mathrm{mag}}(q) = C_{\mathrm{BN}}(q) m_N$$

where s_N is the inertial sun vector and m_N is the inertial magnetic-field reference vector. The coarse sun sensor does not directly output a unit vector; instead, the six face photodiode readings are differenced and normalized to produce the measured body-frame sun direction.

Similarly, the magnetometer update uses the measured body-frame magnetic vector, normalized before the update.

Each update produces a small attitude correction $\delta\theta$ and a bias correction δb_g. The attitude is then corrected multiplicatively, the bias is updated additively, and the covariance is reset with the usual error-state reset Jacobian.

TL;DR

$$\hat{\omega} = \omega_{\text{gyro}} - b_g$$

$$\delta x = \begin{pmatrix} \delta\theta \\ \delta b_g \end{pmatrix}$$

The reported attitude is σ_{BN}, obtained by converting q_{BN} to MRPs and then applying the shadow-set map.

$$h_{\text{sun}} = C_{\text{BN}} s_N$$

$$h_{\text{mag}} = C_{\text{BN}} m_N$$

So, compared to a textbook full-state EKF, the current implementation keeps the nominal attitude separately and only filters the small attitude-error and gyro-bias-error states.

Attitude Control System

Black-box overview

Our inputs are the estimated attitude and angular velocity, plus an optional target attitude state, and we aim to produce a commanded body torque.

At the subsystem level, that commanded torque is later converted into a magnetic dipole command for the magnetorquers. Because magnetorquers obey $\tau = m \times B$, only the torque component perpendicular to the local magnetic field is directly achievable at any instant; the allocator / actuator layer handles that mapping and saturation.

Theory

From a mission point of view, we care about three broad control behaviors:

1. Rotational velocity kill (detumble): stop large initial rotation and bring the satellite close to rest
2. Fixed attitude / Earth-pointing hold: maintain a specified reference attitude
3. Ground-station tracking: update the target attitude over time so the spacecraft points toward a ground station during a pass

The current controller crate directly implements two control laws:

- if no target is set, it performs rate damping (detumble)
- if a target is set, it performs attitude tracking about that supplied target state

So fixed-pointing and ground-station-tracking are not separate low-level laws inside this crate; they are different ways of generating the target attitude trajectory that the controller follows.

At the base we use a PID-style control law on attitude and angular-velocity error:

$\text{pid}(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de}{dt}$

where: $e(t) = \text{expected} - \text{actual}$ at time t.

In the current implementation, the tracking law is more specifically

$$\tau_c = \mathrm{sat}\left(-K_p\sigma_{\mathrm{BR}} - K_i\int \sigma_{\mathrm{BR}}dt - K_d\omega_{\mathrm{BR}}^B\right)$$

with

$$\omega_{\mathrm{BR}}^B = \omega_{\mathrm{BN}}^B - C_{\mathrm{BR}}\omega_{\mathrm{RN}}^R$$

If no target is active, this reduces to a pure derivative-style damping law:

$$\tau_c = \mathrm{sat}\left(-K_d\omega_{\mathrm{BN}}^B\right)$$

The integral term is clamped to avoid windup, and the commanded torque norm is also saturated. The target-state interface already contains a desired angular acceleration term for future feedforward support, but the current controller sets the feedforward torque contribution to zero. Tuning for the K constants is expected to be done via simulation and hardware testing.

Simulation pipeline

Note: the simulation pipeline is mostly for ADCS; it serves as an interface to test routines before the actual launch. It is written for Basilisk, an astrodynamics simulator. Simulator nodes publish satellite updates and accept control commands through the Zenoh-based sim network.

Outgoing state updates mostly contain satellite attitude/orbit data (for testing suites, etc) as well as raw virtual sensor data that will be processed in the routines. Incoming commands at the moment are `applyTorque` and `applyMagneticDipole`.

Comms

Introduction

PixelSat operates on UHF frequencies at the 435 MHz band. For modulation, Pixelsat primarily uses the LoRa modulation scheme because of its extremely high link budget and its resistance to Doppler shift. Pixelsat sends down a LoRa heartbeat packet once every 60 seconds; see downlink packet type TODO. The vast majority of communications that occur will be directly between our ground station and the satellite; the ground station will begin LoRa communications with the satellite after detecting a heartbeat. The TinyGS ground station network will be used for larger transmissions – such as image downlinks – and the heartbeat. However, to ensure a TinyGS ground station is overhead, the satellite needs its orbital elements (two-line elements, or TLEs) from ground. It will have rough TLEs from prior information about the launch, but any sensitive TinyGS transfer (i.e., images) will require precise TLEs from ground. For TinyGS heartbeats, the receipt of the heartbeat is assumed to be not guaranteed, so precise TLEs are not needed.

The OV2640 onboard camera module supports several "subsampling modes" which affect both its frame rate (which we, of course, do not care about) and resolution. Images will most likely be taken at the CIF resolution of 408x304, but, if all goes well, further image attempts at the SVGA resolution of 818x610 may be attempted.

The transceiver module the satellite is using is the E22-400T30D module from Ebyte, which is based on an SEMTECH SX-1262 series LoRa module. It supports both LoRa and GMSK modulation, with a maximum transmission power of 30 dBm and supporting a carrier frequency of 410 to 493 MHz; the transceiver is connected to the STM32 over UART. Ashwin is working on a Rust driver for the transceiver. Unfortunately, while the underlying SX1262 chip does support both LoRa and GMSK, the E22-400T30D module does not support GMSK mode, so we use LoRa mode for all communications.

Transceiver

The transceiver driver will probably be in the `crates/comms` crate. It is responsible for managing the physical interface to the E22-400T30D module, including mode control, configuration, AUX handshaking, and for framing, sending, and receiving packets as described in the comms overview.

The driver should be a single `Transceiver` struct that owns the UART peripheral and the three GPIOs it needs:

```
pub struct Transceiver<'d> {
    uart: Uart<'d, Async>,
    m0: Output<'d>,
    m1: Output<'d>,
    aux: Input<'d>,
}
```

We want to interrupt when AUX is pulled, write RX to a buffer, and then read the buffer on next mainloop iteration.

The constructor should take the raw peripherals, run the full configuration sequence, and only return once the module is in normal mode and ready to use. A successfully constructed `Transceiver` should always be in a known state; the caller never needs to call a separate `init`.

Mode switching, AUX

The module's operating mode is set by the logic levels on M0 and M1. We only use two modes; normal (M1=0, M0=0) for all actual comms, and configuration (M1=1, M0=0) at startup. Deep sleep is handled at the system level by cutting power to the module via the MOSFET on the $5V$ rail since restoring from deep sleep still requires a mode switch and AUX wait and offers no meaningful advantage over a power cycle.

After any change to M0 or M1, the module takes 9-11 ms to transition, during which it holds AUX low. The driver must poll AUX and not interact with the module until it goes high.

Similarly, before any write to the UART in normal mode, the driver must confirm AUX is high. AUX going low mid-operation indicates the module is processing a prior transmission or has just received a packet; writing into this window runs the risk of completely silent data loss.

Configuration

On init, the driver should switch to configuration mode and issue the following AT commands over UART at the fixed configuration baud rate of 9600 8N1:

- `AT+ADDR=0xDE69` (2-byte)
- `AT+NETID=0`
- `AT+CHANNEL=25`
- `AT+UFREQ=-125000`
- `AT+RATE=2` (2.4 kbps air rate, also just the default so technically no need to set this)
- `AT+POWER=0` (30 dBm)
- `AT+TRANS=1` (fixed-point mode)
- `AT+PACKET=0` (240-byte packet size, just the default again)
- `AT+UAUX=1` (set AUX to stay low for full RF airtime)
- `AT+DRSSI=?` (see Appendix 1)

Each command is written to UART and the driver must wait for the module to respond with `=OK` before issuing the next. A 500 ms timeout on each response is sufficient. If any command times out or receives a non-OK response, configuration fails. In deployment, we should most likely just reset the transceiver if this happens via the MOSFET.

After configuration, the driver switches back to normal mode and the UART baud rate is no longer constrained to 9600, though since we also configure `AT+UART` to 9600, it stays the same.

Sending

In fixed-point mode, the module does not prepend the 3-byte routing header itself; the host is expected to prepend it. So for every outbound transmission, the driver writes 243 bytes to UART: `[ADDH, ADDL, CH]` followed by the 240-byte packet. The module detects a frame break (a gap of approximately 3 byte-times, ~3 ms at 9600 baud) after the last byte and begins the RF transmission.

After writing, AUX will stay low from when the UART data begins to get loaded into the internal buffer through the entire RF airtime, at the cost of blocking the send path for the entire on-air duration.

Receiving

LoRa is half-duplex; the SX1262 cannot simultaneously transmit and receive. If a packet arrives over the air while the module is transmitting, it is silently lost at the RF level without any buffering and no indication to OBC that anything was missed. As far as I can tell, this is unavoidable regardless of what we do in the driver.

The consequence for protocol design is that the ground station must not transmit a command during the window when the satellite is transmitting. Since the satellite's heartbeat and response transmissions are initiated by STM32, the timing is predictable: ground should listen first, send a command only after the satellite's transmission has completed, then wait for a response. On the receive side, when a packet arrives the module pulls AUX low for approximately 3 ms, outputs the 240 bytes on TX, then returns AUX high. After we receive the packet (TODO: figure out impl, interrupts or no?), we validate the magic (destiny hehe) at the start of the buffer and do whatever with the packet.

Packet alignment

There is one major failure mode though; if OBC restarts mid-reception or the UART queue contains stale bytes from a previous session, the read boundaries will be misaligned and all following packets will fail the magic check. If our driver gets misaligned, it will stay that way indefinitely.

We recover by scanning byte-by-byte through the UART stream using a sliding 8-byte window until the magic is found, then reading the remaining bytes to complete the packet. Once resynchronized, the driver will return to the fast fixed-length path. This resync will be triggered whenever the magic check fails on a normal read.

Error handling

```rust
// TODO: this is bs
pub enum TransceiverError {
    UartWrite,
    UartRead,
    AtTimeout,
    AtNack,
    Busy,
}
```

`AtTimeout` and `AtNack` are only possible during configuration. `Busy` is returned if `send` is called while AUX is low, which really should never happen but should probably be guarded against regardless. `UartWrite` and `UartRead` are simply hardware-level UART faults.

Framing

All packets contain the same following basic structure. All packets are 240 bytes (the max transmittable length of the transceiver).

1. 8-byte magic string. This is a fixed-length ASCII string used to identify the packet. The magic string used on downlink transmissions will be `PIXELSAT`. The magic string used on uplink transmissions is `DESTINY_`.
2. 4-byte CRC32 checksum. This is used to verify the integrity of the packet. This is computed by running the CRC32 algorithm on the contents of the packet (i.e., the 228 bytes, not including the transceiver framing, magic string, or checksum). Forward error correction such as Reed-Solomon error-correcting codes is currently not planned.

   The remaining 228 bytes of the packet are considered the "contents", which is what the checksum is run on. For *all uplink packets*, the contents are encrypted (the checksum is ran before, not after, encryption). Encryption uses AES-GCM-128 with a 16-byte passphrase shared by ground and the satellite. Of course, the passphrase is not published.
3. A maximum of 228 remaining bytes, we do not transmit a full packet if we do not have to, in order to reduce strain on transciever. If this is an uplink packet, an extra 12 bytes are used by the nonce.

Transceiver framing & configuration

The transceiver module also adds framing of its own to the packet, as we use it in point-to-point mode:

> During fixed-point transmission, the module identifies the first three bytes of serial data as: address high byte + address low byte + channel, and uses this as the wireless transmission target.

In other words, the framing of the Ebyte module consists of 3 bytes prepended to the packet content containing a 2-byte address and channel. The channel determines the frequency as follows:

$$\text{frequency} = 410.125\ \text{MHz} + \text{channel} \times 1\ \text{MHz} + \text{adjustment}$$

To attain the 435.000 MHz frequency needed for our transmission, we thus set the channel to 25 and add an adjustment of -0.125 MHz or -125 kHz using the `AT+UFREQ` AT (attention) command.

Importantly, on outbound transmissions, the transceiver *does not* add this framing for us; we are expected to prepend the 3 bytes ourselves. However, we must still inform the transceiver of the module address (e.g., our own identity) using the `AT+ADDR` AT command. In TX, the module address of the receiver (e.g., the address of the ground station) is inferred from the header we include in the packet. In RX, a packet is dropped if the address in its 3-byte header does not match the module address set by `AT+ADDR`. It is thus imperative that the address used by a transmitter in outbound transmissions matches the address set on the transceiver by the receiver.

After the header is prepended, all OTA packets are 243 bytes. However, if the address and channel match on the receiver, the header is stripped out before the packet is passed onto the STM32. In summary, a transmitter must include the 3-byte header with a channel and address matching the receiver's settings, but the receiver need not account for the header in its packet parsing. The addresses for the ground station and satellite are currently TBD.

Regulation

The FCC imposes regulations on the operation of amateur radio stations through Title 47, Chapter 1, Subchapter D, Part 97. PixelSat itself also qualifies as an amateur radio station, which is defined as

> An amateur station located more than 50 km above the Earth's surface, [§97.3(41)]

which of course is fulfilled by an orbital satellite. Similarly, any ground station used to communicate with PixelSat qualifies as a telecommand station, defined as

> An amateur station that transmits communications to initiate, modify or terminate functions of a space station. [§97.3(45)]

Radio stations are in general prohibited from transmitting obscured messages (e.g., using encryption or similar):

> messages encoded for the purpose of obscuring their meaning [are prohibited], [§97.113(a)(4)]

but telecommand stations are exempt:

> A telecommand station may transmit special codes intended to obscure the meaning of telecommand messages to the station in space operation. [§97.211(b)]

which allows us to encrypt all uplink packets.

Regardless, it is sensible to publish our framing protocol so that ground station operators can interpret PixelSat's downlinked packets, and to avoid any skirmishes with the law.

Further incentive to publicize our framing is the TinyGS ethos; their fourth key principle is

> 4. Fairness. Every station operator in the community has exclusive control over their station and has the right to know what communications are being held through their station.

and they repeat in several places that all transmissions received by TinyGS should be in plain, decodable formats:

> Identifiable packets from a known satellite whose structure and content has not been published. For short, we call these Obscure packets. Because its information is hidden even if it is not encrypted.

> However the community groundstations must not be used for profit. Obscured payloads are a threat for this. Its real content is hidden so there is no way to verify they meet key principles 1, 3 and 4, and they add no value to the community as its content has no readable information. Therefore the payload of non-public payloads will not be saved on the public TinyGS database.

> "I am a satellite operator. Can I use TinyGS stations to get packets from my satellite?"
>
> Yes, as long as the content of the payloads is public. This is the way to meet principle 4 so station operators know what type of communication is being held though their stations and study and learn from it.

This suggests that, as long as we publish reasonably detailed documentation about our framing (i.e., this document), TinyGS will allow reception and storage of PixelSat transmissions.

Packet types

As general convention, all numbers used to refer to a chunk index or number of chunks in an image are 2 bytes; images are chunked into 222-byte parts.

Name	Type	Payload length	Response	Description	Payload description
Ping	0	0	Pong (0)	Tests connectivity.	—
GetImageInfo	1	0	ImageInfo (2)	Requests information about the stored image.	—
GetImageChunk	2	4	ImageChunk (3)	Requests a chunk of the stored image.	The chunk index and number of chunks to send (both u16). For example, if the payload is 0x00010002, chunks 1..<3 will be sent.
Reset	3	0	—	Resets the OBC.	—
ExecuteTorqueProfile	4	TBD (will be ≤224)	AdcsInfo (4)	Executes a torque profile on the ADCS.	Payload shape TBD, but will essentially be a vector of tuples of (torque vector, duration).
GetAdcsInfo	5	0	AdcsInfo (4)	Requests info about the ADS and ACS.	—
CaptureImage	6	TBD (will be ≤224)	ImageInfo (2)	Requests the OBC to capture an image.	Any associated flags for the image capture request. TBD but will likely include what resolution setting to use and whether to compress the color profile.
SetDownlinkSchedule	7	TBD (will be ≤224)	TBD	Sets points in time to downlink various chunks. PixelSat will wait until the specified time before sending the packet. It will attempt to adhere to attitudes if possible.	Ordered list of (time, packet, attitude).
SetAdcsOmegaKill	8	TBD (will be ≤224)	AdcsInfo (4)	Sets satellite to kill rotational velocity.	—
SetAdcsAttitude	9	TBD (will be ≤224)	AdcsInfo (4)	Sets satellite target attitude.	Attitude in MRP
SetAdcsTarget	10	TBD (will be ≤224)	AdcsInfo (4)	Sets a target ground station to track	Targets lat/long
SetAdcsAuto	11	TBD (will be ≤224)	AdcsInfo (4)	Lets satellite decide what to do (rotational kill unless transmitting image)	—
SetTimestamp	12	TBD (will be ≤224)	Heartbeat (6)	Set the timestamp	u64 for milliseconds

Table 1: Uplink requests

Name	Type	Payload length	Request	Description	Payload description
Pong	0	0	Ping (0)	Response to a Ping command.	—
Error	1	1	—	Indicates an error occurred during communication.	The error code. A rough error list is in Appendix I, but the exact mapping of error code to meaning is not yet finalized.
ImageInfo	2	TBD (will be ≤224)	GetImageInfo (1)	Info about the stored image, including but not limited to 1) whether there is an image in onboard nonvolatile storage or if an image capture is in progress, 2) the number of chunks of a stored image, 3) the resolution of the stored image.	See Appendix II
ImageChunk	3	224	GetImageChunk (2)	Response to a GetImageChunk command.	2-byte chunk index followed by 222-byte chunk. Zero-padded if the chunk is smaller than 222 bytes (e.g., the last chunk).
AdcsInfo	4	TBD (will be ≤224)	GetAdcsInfo (5)	Info about the ADS and ACS, including but not limited to sensor statuses, current estimated orientation, current angular velocity, and raw sensor readings. Also returned after a torque profile is executed.	Payload shape TBD.
Heartbeat	6	113		Heartbeat packet sent by the satellite every 60 seconds, including general info about comms, ADCS, OBC, and other sensors.	See Appendix II.

Table 2: Downlink responses

Appendix I: rough error list

Comms

- MalformedPacket
- CrcFailed
- ScheduledTransmissionReadFailure
- TransceiverTimeout

ADCS

- ImuFailure
- ImuNonfatalFailure
- SunSensorFailure
- MagnetometerNoisy
- MagnetometerPwmFailure
- IgrfReadFailure
- AcsPidFailure
- AngularVelocityAbnormallyHigh

General

- PostSystemReset
- RtcFailure
- CameraFailure
- NoTleStored
- LowBattery

Appendix II: unabridged packet shapes

```
#[repr(u8)]
pub enum AdcsMode {
    VelocityKill,
    AttitudeTarget,
    TrackingTarget,
}

#[repr(bitpacked)]
pub struct AdcsTelemetry {
    /// Milliseconds since boot when this was generated.
    pub uptime: u32,
    /// active ADCS mode
    pub mode: AdcsMode,
    /// Estimator health / convergence bitmask
    pub estimator_status: u16,
    /// Controller / actuator status bitmask
    pub controller_status: u16,
    /// Raw gyro measurement in body coordinates [rad/s].
    pub gyro_body: [f32; 3],
    /// Raw magnetometer measurement in body coordinates [tesla].
    pub mag_body: [f32; 3],
    /// Raw accelerometer measurement in body coordinates [m/s^2].
    // TODO: ADCS might or might not be using accel to be fused with gyro ...
TBD if this is even useful
    pub accel_body: [f32; 3],
    /// Raw coarse sun-sensor values from the six dedicated face photodiodes
in face order +X, -X, +Y, -Y, +Z, -Z.
    pub css: [f32; 6],
```

```
    /// Estimated body attitude as MRPs.
    pub attitude_mrp: [f32; 3],
    /// Estimated body angular velocity [rad/s].
    pub angular_velocity_body: [f32; 3],
    /// Estimated gyro bias [rad/s].
    pub gyro_bias_body: [f32; 3],
    /// Current target attitude as MRPs, or zeros if no target is active.
    pub target_attitude_mrp: [f32; 3],
    // TODO: Tracking GS option (maybe use union)
    /// Tracking error in MRPs, or zeros in non-tracking modes.
    pub attitude_error_mrp: [f32; 3],
    /// Controller output in body coordinates before actuator allocation.
    pub commanded_torque_body: [f32; 3],
}

#[repr(bitpacked)]
struct ImageInfo {
    /// Whether image has been taken yet
    pub exists: bool, // 1 byte repr
    /// Total number of chunks in image
    pub chunks: u32,
    /// Size of image in bytes (consider that last chunk might not be full
length ...
    pub image_size: u32,
    /// X resolution
    pub res_x: u16,
    /// Y resolution
    pub res_y: u16
    // TODO: anything else we need
}

#[repr(bitpacked)]
struct Heartbeat {
    pub r: [f32; 3],
    pub v: [f32; 3],
    pub mag: [f32; 3],
    pub omega: [f32; 3],
    pub accel: [f32; 3],
    pub css: [f32; 6],
    pub att: [f32; 3],
    pub rad: f32,
    pub temp: f32,
    pub uptime: u32, // milliseconds
    pub timestamp: u64, // milliseconds; 0 for no RTC state?
    pub cam: bool,
}
```

Appendix III

> After activation, when the module receives wireless data, it outputs via the serial port TXD followed by an RSSI strength byte. The current packet RSSI is: dBm = -(256 - RSSI)

Is it worth setting RSSI? @aadish, @ashwin

Sources

Transceiver Section

https://www.cdebyte.com/pdf-down.aspx?id=4217
https://cdn.sparkfun.com/assets/6/b/5/1/4/SX1262_datasheet.pdf
https://avbentem.github.io/airtime-calculator/

Everything Else

https://github.com/tinygs/tinyGS/wiki/Manifest-Work-in-progress
https://www.cdebyte.com/products/E22-400T30D
https://www.ecfr.gov/current/title-47/chapter-I/subchapter-D/part-97/
https://www.uctronics.com/download/cam_module/OV2640DS.pdf

Architecture

We use the Real-Time Interrupt-driven Concurrency (RTIC) framework to schedule and handle all our functionality on a single CPU. RTIC is much more lightweight than a full RTOS but provides guarantees like race-free resource access and deadlock-free execution via static analysis.

At its core, RTIC involves an app, which is an executable system model for single-core applications, declaring a set of local and shared resources operated on by a set of init, idle, hardware and software tasks.

RTIC is using a hardware interrupt controller (ARM NVIC on cortex-m) to schedule and start execution of tasks. All tasks run as interrupt handlers. Tasks bound to an explicit interrupt are called hardware tasks since they start execution in reaction to a hardware event. A software task is not explicitly bound to a specific interrupt vector, but rather bound to a "dispatcher" interrupt vector running at the intended priority of the software task.

One thing to consider is that RTIC thinks of tasks as a short bursts, but we often have long loops. This incongruency is solved by async rust. Yield points provide task segments that RTIC uses to split loops into smaller tasks.

On PixelSat we shall have the following tasks:

- EByte E22 interrupt handler (priority 7)
- IMU interrupt handler (priority 6)
- Mag interrupt handler (priority 6)
- Photodiode / CSS sampling task or interrupt (hardware integration still TBD, priority 6 probably)
- Camera interrupt handler (priority 5)
- Rad sensor interrupt handler (priority 4)
- Temp sensor interrupt handler (priority 4)
- ADCS (priority 3)
- Handling comms message (priority 2)
- Heartbeat task (priority 1)
- idle (priority 0)

The general pattern is as follows:

- Really important interrupt servicing
- Important interrupt servicing
- Less important interrupt servicing
- ADCS
- non time critical operations

Interrupt servicing simply records whatever data was provided, it doesn't perform any long running tasks

Software Codebase

Status

ADCS

Writing drivers for various sensors. Core logic complete

Comms

Mid-planning and doing hardware assembly.

Contributing

- Submit PRs
- Do not lump too many changes into the same PR.
- Do not force push to main
- Maintain the monorepo for the esp (no creating random repos).
- Maintain working code on main

Code Quality Guidelines

- Ensure code is formatted
- Ensure clippy passes
- As an extension, ensure CI/tests pass as well (this covers formatting and clippy)
- No recursion (unless attempting explicit tailcall optimization)
- No heap allocation after init (unless absolutely necessary)
- Add testing when possible
- Remain platform agnostic when possible
- Avoid adding dependencies when possible

AI Policy

Contributions to any codebase related to this project must not include content generated by large language models or other probabilistic tools, including, but not limited to, Copilot or ChatGPT. Code generated by tab complete tools, such as Copilot Next-Edit Suggestions (NES) or Zed's Edit Predictions, are acceptable, but if you commit code generated by such tools, you are expected to be able to defend it.

This policy covers code, documentation, pull requests, issues, comments, and any other contributions to the PPO project. For now, we're taking a cautious approach to these tools due to their effects — both unknown and observed — on project health and maintenance burden. This field is evolving quickly, so we are open to revising this policy at a later date, given proposals for particular tools that mitigate these effects.

We reserve the right to close any PRs that include AI-generated content.

We reserve the right to update this policy at any time.

Rationale

PPO intends to put a satellite in space; as such, we are incredibly careful when auditing and considering new code. While we acknowledge AI is a powerful tool, we cannot risk an entire mission (costing in the hundreds of thousands of dollars) on probabilistic models.