Microcontroller Flight Computers For CubeSat Applications

SPACE SYSTEMS DESIGN STUDIO

Adler Smith
[ags228]

Table of Contents

1. Introduction

Since California Polytechnic State University and Stanford introduced the CubeSat standard in 1999, hundreds of these small satellites have been launched into orbit. Most of these have been produced by academia, but reduction in launch cost via these common specifications has seen extensive private interest. This explosion in the growth has been made possible by the Moore's Law advancement in consumer electronics driving costs down, while increasing the available processing power in smaller and smaller form factors. As more organizations look into developing and launching CubeSats, common templates, designs, and components have become commonplace. Numerous suppliers offer CubeSat components with prices ranging into the tens to hundreds of thousands of dollars range. While this is below the design costs of conventional satellites, Cornell University has made it an objective to lower this cost barrier even further with projects with StarShot Alpha, a 1U CubeSat with an initial $10,000 total budget, or CU-ACE, a complete attitude determination and control system for a fraction of the cost of a comparable system. Flight computers and flight software remain as a daunting challenge for nanosatellite designers, as there are little frameworks available, and costs are extreme for pre-built systems. This paper seeks to discuss the feasibility and challenges of using off the shelf micro-controllers to replace expensive flight computers for general CubeSat applications and provide a software framework to kick-start new CubeSat software development with structure guidelines and common component interfaces. The design of the flight computer for Cornell University's *Pathfinder for Autonomous Navigation* (PAN), a 3-U satellite with onboard propulsion, attitude determination and actuation, and docking mission objectives is referenced in conjunction with these guidelines to show their application to the design process.

2. Hardware Selection

During a CubeSat design cycle, the selection of the processor and flight computer general architecture should be done as soon as possible as it impacts numerous design choices later on. The processor and the board that it comes with determines the length, complexity, and mathematical precision of the software that it can execute in a reasonable time and the board determines the number of components, sensors, or other systems that can be actuated or interfaced with. The general architecture refers to the decision to use a single microcontroller or multiple ones and where certain code is executed, such as the choice of a specialized Attitude Determination and Control computer, which interfaces with a main flight computer. The restrictions on the size, mass, and power consumption of each microcontroller should be determined prior as well to assist in selection. Each design choice comes with trade-offs which must be considered before any hardware is selected.

2.1 Processor Memory

Microcontrollers traditionally have three types of memory, though there are always exceptions: Flash, Random Access Memory (RAM), and EEPROM, with a few processors supporting external memory. In most cases, the available size of Flash is the greatest, followed by RAM, and then EEPROM and external memory can vary.

2.1.1 Internal Memory

Flash memory is the read-only permanent memory that persists after a power loss and is used to store the compiled program software and the processor's bootloader. Simply put, the size of the Flash memory determines maximum size of your compiled code as it must reside entirely in Flash. It is important to note that not 100% of the listed Flash memory area is available as a small partition is used by the loader when your code is uploaded. Constants like large lookup tables can be stored in program memory with the PROGMEM modifier and related retrieval functions to save on RAM. RAM is used at runtime to store all the variables currently used in execution as well as the function stack. Any data stored here cannot be retrieved if power is lost or a reboot occurs. Loosely, RAM limits the maximum complexity of any executing code as a stack overflow from excess RAM utilization would cause a crash or memory corruption. Global variables are stored here as well. EEPROM is unique from RAM as it is both accessible at program runtime, but also persists after power loss. EEPROM is often very small with less than 1-2kB of available space and is addressed directly byte by byte. Permanent data like counters or crash logs must be stored here to be used after a reboot. On PAN, the counters for number of firings per thruster, reboots, and mission phase all are stored in EEPROM, for example. These three variants of memory differ significantly in access time. EEPROM reads and writes are slow with a single byte I/O executing on the order of 1ms, whereas RAM or Flash I/O occurs in single digit clock cycles on a reasonable processor. Without extensive modification or software workarounds, Flash is unreachable at runtime, so cannot be used for storage of non-constant values.

2.1.2 External Memory

As it is not part of the processor itself, external memory is a vital component for any microcontroller based system attempting to use or store more than ~1MB of data. External memory units vary extensively, but the simplest and most commonly used is an microSD card accessed over an SPI interface. The code and interface to do this will be discussed later. External memory permits datalogging, which is generally impossible without it, as RAM is neither persistent nor large enough to hold a meaningful number of datapoints. As it is external, it can be sized to whatever the needs are. A single 4GB SD card is on the order of the same size of the processor itself, so the mount and wiring are not a significant concern if space is sufficient. However, I/O to an SD card is generally very slow and the Arduino SD card library is poorly designed. The included SDFat32 Library alleviates this and includes information on bring this up to a reasonable speed. Despite this, external memory is still slower than RAM and should only be used for logging data. It is important to understand the logistics of moving data from external SD cards into RAM as it must be moved in blocks to prevent a memory overflow.

2.2 Processor Precision

The bit precision of the processor determines what data types it can quickly process and to what degree they remain accurate. Most computers are 32-bit or 64-bit precision currently, while microcontrollers are made in 8-bit, 16-bit, or 32-bit. The precision determines the size of the instructions it executes and the data size that can be used in calculations quickly by design of the hardware. Some processors support larger datatypes via software, but this comes at the expense of execution speed. For most applications that do not involve extensive mathematical computation, it really is not that important which precision is selected, although high precision generally correlates with more memory, faster speed, and newer technology. For applications with extensive calculations, such as processing control laws for attitude determination or mission planning calculations like thruster firings, microcontrollers that support double precision floating point data types are required. As floating point values decrease in accuracy as their value grows, single precision floating point values generally do not have the accuracy needed for accurate positioning and velocity calculations on the scale of the Earth. Double precision values are used in the attached Piksi GPS library, as they are the standard for accurate GPS calculations. Single precision floats may be accurate enough for some calculations, but since many algorithms ported to flight computers are designed in MatLab-Simulink, which uses double precision values, running this code at the same level of precision ensures that the code functions the same on both the flight computer and in MatLab-Simulink.

2.3 Processor Clock Speed

Processor speed determines the viability of any flight computer and must be sufficient to execute the software loops in a reasonable time. Processor speed ranges significantly across boards, from the low MHz up to ~400 MHz for some. The clock speed determines the executed instructions per second, and thus, generally how fast your code can execute. Again, for applications without extensive computation, even low clock speeds are overkill, while applications with a lot of runtime floating point math may require it. The required speed is something that is somewhat hard to determine before the MatLab code is ported, but an estimate can be obtained by calculating the total number of floating point operations for the script, found in the debugging/timing help in Matlab, and dividing by the estimated floating point operations per second (FLOPS) for your processor, which can be found in a datasheet or approximated by running ~1 million floating point operations and timing it. This result gives the rough time for a single execution of that program on that microcontroller. For code executed frequently, it is important to keep this below any time steps of control laws or significant mission pertinent timescales. If an algorithm executes only a few times per orbit, then a execution time longer than a timestep can be permitted, provided it does not interfere with anything else, as on a single processor only one function can be executing at a time. However, multiple microcontrollers can increase the effective speed by dividing the work, though this has diminishing returns due to communication overhead. It can be a good design to separate control laws that run many times per second from code that executes far slower on to different boards.

2.4 General System Architecture

Selection of a microcontroller must include an inventory of what components and sensors the system must interface with and how frequently it must do so. External connections are possible with general purpose input output pins (GPIO), analog and pulse width modulation (PWM) pins, Serial Communication, the I^{2}C Bus, or the Serial Peripheral Interface (SPI) bus. On most boards, each pin can be multiple types and careful planning must be used to ensure enough of each type are available once others are taken.

2.4.1 GPIO and Analog Pins

GPIO pins are the simplest and most straightforward external interface supporting digital I/O. Voltage high or lows can be read from, or written to, each GPIO pin. The mode must be set as either input for reads or output for writes, though it is possible to read from an output pin. Each pin can supply or drain a certain amount of current to run an external device and each pin group, indicated by a letter like PORT-A, has a total current limit which cannot be exceeded. These limits are found in the microcontroller data sheet or pinout diagram. Analog pins support a greater read precision in input mode than binary high or low, rather they map an input voltage onto a range, usually 0-1023, corresponding to the fraction of the voltage high. However, the behavior is different in output mode. On most microcontrollers there is no true analog output, so analog outs are approximated with pulse width modulation. PWM outputs range from 0-255, which corresponds to a duty cycle from 0, which is full off, to 255, which is full on. This forms a square wave with average voltage equal to the output value divided by 255. Attaching a tuned RC circuit can do some signal smoothing if steadier signals are required. Some microcontrollers, such as the ATSAMD21 Cortex M0 used in Adafruit SAMD boards, have true analog outputs via a Digital to Analog Converter (DAC) on one or two pins, though this is uncommon.

2.4.2 Serial Communication

The data transmission interfaces, Serial, I^{2}C, and SPI will be discussed in detail later, but in the earlier design stages only the number of them or their presence is need to pick hardware. Each digital device that the flight computer must interface with will have a single communications protocol that it is designed to use, and thus the microcontroller must support it to work with that device. Serial communication is an asynchronous 2 wire interface, meaning data transmissions are sent and there is no guarantee that the receiver was listening at all or any form of error checking, unless added via software. One wire is used for transmission, the other for receiving and there is no way to detect that they are connected to a working device unless data is sent back on the other wire. Microcontrollers have a fixed size serial buffer, which contains any unread data. If this is exceeded, behavior is indermerinant with incoming data lost or overwriting old data in the buffer. The size of this buffer usually is 32 or 64 bytes, so receiving data over this size must be done in segments or synchronized with the sender. Multiple devices can be connected to same serial transmission or receiving lines, but only one can send data at once or data may be corrupted; multiple receivers can be done without issue. Generally, Serial is used for single device to device connections, however.

2.4.3 I^{2}C Bus

I^{2}C is a synchronous 2 wire master bus protocol, where a single device acts as a bus controller or master, sending data to or requesting data from any other slave devices on the bus. One wire is used as a clock signal to synchronize the bus and other to send data. Each device on the bus is addressed with the master as address 0. Only the master device can initiate data transfer, either by sending data to a slave device or requesting a number of bytes to be returned from a slave device. The slave devices run an interrupt routine on receiving data and another when data is requested, so data is processed the instant it is received and data sent the instant it is requested. Returning less than the number requested is permitted, but extra bytes will be dropped. There is a max byte transmission limit but it is component specific with ranging values and minor support for increasing it on some microcontrollers. Up to 127 slave devices can be connected on a single bus, with some microcontrollers supporting more. It is very important to consider that many devices, such as Pololu Inertial Measurement Units, have a preprogrammed I^2C address. This means that two of these devices cannot be attached to the same bus due to the address conflict. Very rarely, two separate devices may have the same I^2C address and may can be checked on Adafruit's I^2C Address List. Some devices include a pin toggle that can toggle the I^2C address between two values, so two can be put on a single bus. Larger microcontrollers often have multiple I^2C buses to mitigate this issue. Microcontrollers can choose their own address if they join an existing I^2C network as a slave device.

2.4.4 Serial Peripheral Interface

SPI is used less frequently than the other two options, but as it is used exclusively on SD cards, it is relevant to our applications. SPI is a 4+ wire synchronous protocol that supports multiple slave devices via a slave select line. Each slave device requires a line to it, when set to voltage high, that device is listening for transmission. Like I^2C the only the master device can initiate communication, however the return number of bytes is flexible, making it good for unknown sizes of returned data. Many microcontrollers have a SPI interface and hardware mount for an SD card on the board itself, which aids in using it for external memory as no extra wiring is needed. Multiple SD cards can be added on the same bus, only requiring an additional slave select pin, provided they have mounts or connections to the 3 other common SPI wires.

3. Software Framework Overview

Flight software is extremely varied across satellite programs, with numerous different approaches to ensure mission critical success. Some systems have redundant cores, memory error checking, and backup compiled software, among other safety checks. Without extensive effort, none of these are possible on microcontroller architectures. Thus, the software must be designed to handle unexpected crashes gracefully and ensure safe continued operation in the event of an external component failure. Despite the differences from conventional satellite software, the software possible on microcontrollers maintains many similarities to what would be expected. While there are several more complex architectures which will be discussed briefly later, like running a Real-Time Operating System (RTOS) on a microcontroller, this paper focuses on the easiest and most common microcontroller format, a single thread loop. At the highest level, this format is usually programmed in C or a C derivative language, like C++ and contains the equivalent of two functions; a setup routine called on initial boot up and a main loop called after running the rest of the software via subfunctions. If using the Arduino IDE or its compiler, this is explicitly the setup() and loop() functions.

3.1 Setup Routine

The setup routine has numerous vital responsibilities as it must handle system startup in normal conditions at the beginning of the mission as well as restarts after losses of power and software crashes. In the software design process, the setup routine will change extensively but work should begin in completing the standard setup routine before any error handling or reset prevention is attempted.

3.1.1 Standard Setup from Deployment

With regards to its first task, the design procedure for the setup routine is relatively straight forward. Gather of list of all the relevant components which require some form of initializations as well as the functions to do so, and ensure that they and all their dependencies, i.e the IMU may require to I^2C bus to be set at a particular Baud Rate, are included in the setup routine. It is important to include saved flags for the success of any initialization that has the possibility of failure. Conventionally, this is stored in a structure referred to as the Hardware Availability Table (HAVT). This provides a preemptive safeguard against later software crashes, as attempting to access an uninitialized or malfunctioning component can easily result in unexpected behavior, with a best case of erroneous data and a worse case of a segfault or null pointer exception resulting in a hard reset. Any time a component in the HAVT is used, a check to the table must be present to prevent this. Generally, determining if a component is malfunctioning but still responsive is hard to determine in software so the HAVT flag should be able to be toggled from the ground. A component not responding or returning clearly corrupted data is a valid case for the software to set this flag automatically. If a failure to initialize a component can result in a crash, handling it is more complex and explained in the Hard Reset section. The final part of the normal conditions start sequence is entering the correct initial state of operation. While the state machine model for CubeSats will be discussed in the loop section, most specifications of the CubeSat standard require a 30-45 minute dormant coast before any transmissions or hardware deployments like antennas occur. So on the first boot up, triggered by the deployment switches detecting payload release from Nanoracks or a P-POD system, the dormant phase of the loop should be entered after the setup routine finishes. Lastly, it is critical to set a permanent memory flag in EEPROM incrementing the number of reboots as this will be used for future fault recovery to detect that the mission has already begun.

3.1.2 Setup from Soft Reset Error Condition

Unexpected error or fault conditions resulting in a reboot can arise at any time during satellite operations, though they fall into two categories; soft reset triggers or hard reset triggers. Soft resets, in this context, are conditions that will result in a reboot that can generally be predicted such as a progressive memory leak or upcoming possible power loss. The conditions that result in these are generally slow so steps can be taken to preserve mission critical parameters in EEPROM prior to the reset or steps can be taken to prevent the reset entirely. This section focuses only on recovering from such resets once they occur, as mitigating them is done in the main loop. For successful error handling, a EEPROM flag should be set prior to the reset so that the cause can be known during the subsequent reboot. At the start of the setup routine, a switch statement should switch on this value, and enter a handling routine before the rest of the standard setup is run. This will generally involve reducing or increasing frequencies of component use as well as setting the satellite state to whatever it was before the crash, to avoid an accidental dormant cruise. It is good practice to alert the ground system operators of the nature of the crash and the success of the reboot. In most soft reset cases, the reboot can be prevented beforehand but it is important to be able to handle them if they are inevitable, as in many cases like memory leak or problems in the on board timer as reboot is required to fix them.

3.1.3 Setup from Hard Reset Error Condition

Hard resets are far more problematic, both to handle and prevent. On microcontrollers, a hard reset refers to a unexpected software crash, which can be caused by an exception or a watchdog timer (WDT) forcing a reset due to an infinite loop or software hang. Software exception handling like try-catch statements or similar are very memory intensive, as they usually copy the entire function stack, and thus are difficult to implement or completely unsupported, which is the usual case. WDTs are critical to a microcontroller as a consequence of the previous statements. These timers can be internal to the chip or external, but at least one should be present in any system. Most GomSpace power boards include one with a configurable timeout by design. A WDT constantly counts down and if it is not reset or “kicked” by the flight computer and expires, the chip hard resets or the power briefly cut off causing a hard reset. For a rule of thumb, a WDT should be at least twice as long as the time it takes to run any code that could run in between kicks, to ensure that it only ever is used when it should be, but not too long that significant amounts of time are wasted waiting for the WDT to reset.

As the cause is of a hard reset cannot easily be predicted, storing as much data about what code is currently executing right before the crash in permanent memory is helpful in future prevention. EEPROM I/O is generally slow so it may not be feasible to do this around every function, but storing a flag indicating the high level subroutines, i.e. calculating ADCS reaction wheel speed or checking the IMU, that will execute on the next loop as bits in an ~2 byte value can be done. Since EEPROM wears out after around 100,000 writes, multiple locations and a flag on which is valid should be used if mission duration is long enough to warrant it. If a crash is caused by a subroutine that is not critical to operation or communication, not in the downlink/uplink functions or power management, that subroutine should be disabled until a downlink with the current status is sent to prevent a constant loop of crashing and resetting. After the downlink, either enabling the routine or waiting for an uplinked command is permitted depending on the confidence in the error handling routine. These flags should included in the soft reset switch statement so they can be handled on startup as well.

Another relevant case of WDT or exception resets is crashes within the setup routine. As this function is run on every startup, a persistent crash causing function here will prevent the mission from progressing, killing the satellite. With this in mind, the simplest components or data structures should be initialized first and the complex ones must have a EEPROM start and end flag around their init functions which can be detected and skipped if it has failed multiple times. For example, a device on I^2C can hang in a low power mode if certain bits are dropped due to the lower than expected voltage. Thus if the WDT resets the chip, the next time the setup routine runs it can be detected that this device failed to init and it is then left uninitialized with a flag in the HAVT stored so this device will not be used until it can be successfully initialized later, either on command from the ground or once battery voltage is higher. Exception handling is a difficult task, especially on hardware not designed for it, but without it a mission may become much shorter than expected.

3.2 Main Loop

CubeSat flight computers are generally much simpler than larger satellites and operate on a state machine model. This structure is formed on the concept that the spacecraft has distinct modes of operation with each mode having certain routines or functions active based on the mission requirements. For CubeSats some modes are required per the CubeSat specifications and others depend on the mission Concept of Operations (CONOPS).

CONOPS is a term common to the satellite industry, which correlates well to the state machine model, as it lays out the sequence of mission critical events and their prerequisites in a standard format. The required states for a CubeSat mission are an initial dormant cruise and a post mission kill-switch state. The kill switch state permanently disables all formed of satellite actuation and communication, which is required by law. Commonly included modes are an Initialization and Checkout state, Normal Operations, Eclipse Operations, Low Power State, Post Mission Operations, and Fault State as well as any mission specific modes. For example on PAN, mission specific modes are Near-Field Rendezvous Mode, Docked Mode, and Large Slew Maneuvers. Each of these modes has specific sensor and component requirements as well as frequencies their data is needed.

3.2.1 Dormant Cruise, Initialization and Checkout

At the start of a Cubesat mission, the satellite is released from the launcher and the deployment switches are depressed powering the spacecraft for the first time. After the setup routine is run, Dormant Cruise is entered. During this state no actuation or radio transmission may occur. Actuation includes any reaction wheels as well as any electrically deployed antenna or solar panels, mechanically deployed systems are permitted with approval. This state is required to last for 30-45 minutes depending on the specification used. From a software perspective, this state needs only to consist of a timer and nothing else, though it is generally good to include some tracking of battery power or incoming solar panel current, for the Initialization state. CubeSats will often launch with discharged batteries so numerous orbits may be spent waiting for a sufficient amount of charge to be accumulated.

When the timer has expired, the Initialization and Checkout state occurs. During this state, every subsystem is checked for correct functionality if possible. Any basic startup procedures or deployments will occur and be verified. Data should be gathered from sensors and a status downlink should occur, so the ground can observe as well. It is good practice to start a timer when this state is entered, so the satellite can be forced into normal operations after a significant time has past to ensure that mission will still progress if an unexpected and unhandled error occurs.

3.2.2 Normal Operations

For a standard mission, the satellite will execute code in this state for the majority of the mission. Depending on the nature of the mission, different subroutines will be present but a common structure is usually used. Design decisions and selected hardware determines if a polling method or interrupt based method is used for interacting with the other components. Usually polling methods are used where a component is interfaced with and used at some regular interval based on the system timer, as it aids in understanding when events occur, especially with regards to handling error conditions. If components require it or wait times before processing component data is not permitted, interrupts may be used where on a trigger from an external source, code execution is passed to an interrupt routine and the data is processed immediately from whatever device caused the interrupt. Regardless of the interface method used, normal loops generally follows the following format: kick WDTs, check system scheduler, gather sensor data, calculate mission relevant parameters, actuate spacecraft systems, and determine the next mode of operation.

A single loop cycle in Normal operations should begin with kicking any WDTs present in the system to ensure continued operation. Immediately following, the system scheduler should be checked if present. The scheduler permits tracking and execution of future planned events such as thruster firings or payload deployments. A scheduler is only required if the satellite generates future events rather just waiting for a threshold to pass or a timer to expire. On PAN, this scheduler handles propulsion system firings to ensure that they are executed at exactly the correct time. The scheduler is checked at the start of every loop and should enter the planned event's subroutines a second or so before the event is planned to occur to mitigate the effect of any unexpected delay from the previous loop cycle.

After this is completed, sensor data should be collected. Depending on what data is being read, error correction algorithms may be required to remove system noise or drift. Analog values often have significant electromagnetic interference noise from nearby wires or components, especially magnetic torque rods or motors. Simple averaging or more complex methods like Kalman Filters may be required depending on the necessary degree of precision and amount of noise at the expense of computation time. Values from IMUs are often subject to drift which can be mitigated well with a Kalman Filter.

Once all the required data is gathered and stored in an accessible way, mission relevant parameters can be calculated. These values determine if certain subroutines should execute this cycle, as the cycle time is often far shorter than the frequency certain things must be done. This can be as simple as checking a timer to see if a period of sensor data should be moved to external memory or as complex as calculating if the satellite's orbit has as been matched to a target orbit. Any decision making process that does not require leaving Normal Operations should be calculated here. As missions vary extensively, the code here is very mission specific. After these values are determined, the satellite systems should be actuated accordingly. Attitude determination and control timesteps executed based on control laws, radio communication initiated, and thruster firings among other things should occur if needed.

The final part of a cycle is determining if the spacecraft mode should change. The mode can change based on various thresholds to enter a fault state or to enter any of the mission specific modes as well as low power or eclipse operation. Entering fault state is an important consideration for any mission as each subsystem has numerous conditions which may be considered reason to enter a fault state. Generally, any unexpected behavior that if it persists could damage the mission should be considered a valid reason to enter the fault state. This could be satellite under or overheating, an abnormally high spin rate, or persistent lack of solar charging current or numerous other things which depend on the subsystems present. All of the possible ways that something could go significantly wrong, without causing a software crash, should be detected here and then handled in the fault state.

Entering other modes is self-explanatory and is done so based on the requirements for each mission specific mode. Generally, these modes do all the same things as Normal Operation though with different frequencies or perhaps with sensor data logging. For example, in Eclipse Operations downlinks would occur less frequently if the power budget is tight, the same with a low power state.

3.2.3 Fault State

Handling faults is one of the most complex tasks for spacecraft software design as it involves both foresight and flexibility. Prediction of the possible way of errors can arise and be handled is critical to a successful mission as well as supporting recovery methods from unknown faults. At a high level, the fault state is analogous to a Safe Hold Mode where the satellite does as little as possible while awaiting a communication from the ground on how to resolve the issue. Any unnecessary subsystems to this end should not be used unless commanded from the ground for testing purposes. The satellite must remain capable of downlinking any error information that is has and be able to receive and process and uplinked commands to resolve the error for as great a portion of the time it is in fault state as possible. Sufficient power and antenna direction should be maintained if possible. As a spacecraft may tumble erratically if attitude control is lost, the spacecraft must have an average net gain of electrical power in this mode which determines the frequency that downlinks or uplinks can occur. A long duration timer should be included to force the satellite out of fault state as a last resort if uplinks are impacted by the cause of the fault.

Design of a fault tolerant system is exceedingly difficult. During the satellite design cycle, a live list of all the ways that systems can malfunction, how to detect them, and how to attempt a fix for each subsystem should be maintained and updated to assist in creating the fault state code. The code should simply consist of managing the communication subsystem and any critical hardware to keep the satellite alive. The comms subsystem code should handle the uplinked commands with the routines to attempt to resolve the fault as well as downlinking the spacecraft status. After the fault is verified to have been resolved, as separate command is sent to return the spacecraft to the correct mode.

3.2.4 Post-Mission Operations and Kill Switch

After a satellite has completed its primary mission, post-mission operations may begin. Software here is flexible provided it does not interfere with anything else. Once post mission operations are entered, either by threshold or command depending on the mission, the satellite should not be able to re-enter Normal Operations unless explicitly commanded.

In all spacecraft modes, the mandated kill switch must be supported. This is a way to permanently and irreversibly disable the spacecraft preventing it from actuating or communicating in anyway, essentially turning it into a piece of space junk on a ballistic trajectory. While this is required, it should be somewhat difficult to do. Good practice involves sending multiple commands where if any are not present or sent quick enough, the kill switch will not be activated. Due to the nature of satellite launches being public, care should be taken so that the exact method of triggering the switch is not available online, such as in a Public Github Repository. Some comms system hardware is only accessible by the design team which mitigates this concern but in general the ability to end a satellite mission should be kept secure, especially for normal non-Iridium radio systems.

4. Miscellaneous Software Considerations

Microcontrollers have numerous shortcomings and quirks relative to conventional flight computers and the software design must consider them. The following is a short list of issues and good design practices made apparent during the design of the StarShot Alpha and PAN flight computers, though many more may arise in other applications.

4.1 Timers

Microcontrollers have designated timing registers which are used to trigger functions or subroutines on specific intervals. Greatly simplifying the direct use of registers, the Arduino IDE has a function millis(), which returns the number of milliseconds that since the board first received power. While very useful, the on board timers are sensitive to thermal disturbances, speeding up or slowing down a measurable fraction over several hours. Since satellites are subject to thermal cycling via entering and exiting eclipse, the accuracy degrades over several days. Thus if accurate timing is needed, such as timed thruster firings or image captures, a reference point must be set every few hours to mitigate drift. If a GPS board is present, the GPS time can be used or the time can be included in a uplink.

Another concern with timers is integer overflow. Most timers are contained in unsigned long values which for millis has a max value corresponding to ~49 days. If a mission is going to exceed this duration then a more robust timing method must be used to prevent unexpected behavior planning events across this point. Each of the other timers, like micros(), has a similar overflow point which should be understood before it is used.

4.2 Large Array or Data Storage

Many complex control laws or positioning algorithms use large lookup tables or calculation matrices as a consequence of being designed in and autocoded from MatLab-Simulink. When run on a desktop computer, there is little penalty for using large arrays, but on a microcontroller these structures can be a measurable fraction of the total RAM, pushing the system towards a stack overflow. Lookup tables in RAM are easy to avoid since they can be loaded into Flash memory with the PROGMEM variable modifier on most AVR chips or saved in external memory on an SD card and read from there. Arrays that are calculated at runtime and needed in frequent calculations are a concern however. Tests must be done to ensure that a safety margin for RAM consumption is present so stack overflows cannot occur. If arrays larger than available RAM are required, then external memory and custom versions of the functions operating on the array must be included in the design, which comes at the expense of calculation time. Image files are a simple example of a large array which will not fit in RAM on most microcontrollers and must be processed in segments.

4.3 Execution Timestep

Attitude control laws and other algorithms are generally designed and tested in MatLab-Simulink which is a far superior computation environment than can be supported on any microcontroller. Thus it is to be expected that the execution of a single loop will take orders of magnitude longer running on a flight computer. Depending on the complexity, simulations often use a flexible timestep for accurate calculations around rapid input or output variable changes, which is not easy to imitate on less powerful hardware and attempting to do so is a significant performance hit. Simulations should be created with fixed timesteps to ease porting to flight hardware via the C Autocoder included in Matlab. If the loop is able to complete on the microcontroller before a timestep has passed, then it should execute identically to the Matlab version, and is easier to debug later during integrated testing. The Simulink Autocoder supports numerous host architectures specific to chips, but Arduino is the simplest and most human readable.

4.4 Version Control

In industry, Software version control is almost universally achieved with Git via GitHub repositories. Git is a simple free command line application which facilitates multiple people working concurrently on the same code. Git supports individual in-progress local versions of code as well as centralized cloud storage, each with its own version control tree. The code can be branched or merged back together at anytime with the software gracefully handling merge conflicts, where two people edit the same functions, which prevents many issues caused by less robust version control. While very helpful, git has issues with the Arduino IDE. Whenever an updated version of code is pulled from the cloud to a local repository, the Arduino IDE **must be closed** to avoid accidental overwriting of the code when the IDE then saves, as git will not track it correctly. This problem in the Arduino IDE has caused numerous losses of time and effort, so care must be taken not to do the same. Matlab and Simulink both have extensive Git and Github support integrated with the desktop application for ease of use. All Cornell Students have free 3 year pro memberships which permits private repositories and ownership is easy to transfer to maintain them.

4.5 Bidirectional Thresholds

When implementing thresholds for safe operation like minimum battery voltage or temperature ranges, single value thresholds often can cause problems. A threshold of 7.4 volts determining the edge of low power operation can cause the satellite to bounce between modes, as minor noise around that value triggers a state change. In a worse case, the act of changing states pushes the system in the other direction across the threshold, resulting in an unwanted loop. This can waste valuable time and computational power. To prevent this, a upper and lower threshold should be implemented depending the direction the value is moving. For example, a threshold of 7.2 volts can be the trigger from Normal

Operations to Low Power mode but the transition from Low Power to Normal Operations requires 7.6 volts of charge.

4.6 Threshold/Data Modification via Uplink

One of the most important parts of attempting to resolve in-flight errors is the ability to modify a satellites decision making process. If it less than expected solar charging current is observed, then power intensive systems should operate less frequently. Thus any timers or threshold values that determine when things run need to be modifiable from the ground. This can also be extended to overrides or sensor data spoofing. For example, if a CubeSat's thrusters are not permitted to fire unless a certain tank pressure is reached, but the pressure sensor is malfunctioning, then the sensor must be able to be overridden or the data from it spoofed to above the threshold. Being able to control any subsystem or subroutine from the ground, if required, is a very desirable trait for flight software.

5. Hardware Drivers

The software design process for CubeSats traditionally has a bidirectional approach, both top down and bottom up. Work on the high level layout can be done at the same time as the hardware and component drivers, the code that interfaces with sensors and actuators. Included with this work are drivers with easy "plug and play" functionality for the Piksi GPS board, QLocate Radio, and the GomSpace EPS Board. These components or similar ones are common on CubeSats and using these tested drivers can accelerate the design process so work can focus on more high level mission specific tasks, rather than redesigning existing software. Each of these drivers is given as a C++ library and header file and was written using the Arduino IDE commonly used for microcontrollers.

5.1 Piksi GPS

The Piksi GPS board built by SwiftNav provides numerous valuable functions for CubeSat missions. Full position and velocity solutions are provided at 50Hz and relative positioning between two satellites with Piksi boards via carrier phase RTK are supported. The included driver returns the GPS time as well as position data in the latitude, longitude, altitude or north, east, down reference frames as well as other relevant information. Use of the driver is very simple, only a UART serial connection is needed to the Piksi board. Simply calling the constructor function with the used serial port as an argument at setup and then running updateGPS(...) at the solution rate, which can be modified, is all that is required. The full specification of these functions are available in the Piksi.h and Piksi.cpp files.

5.2 QLocate Radio

The QLocate Radio from Quake is an Iridium network Short Burst Data protocol radio. On downlinks from the satellite, up to 320 byte blocks of data are sent to the device from the flight computer, a communication session with the Iridium grid is initiated, and the data is sent over the internet a computer via a web portal. Uplinks follow the reverse process, though it is important to note that the communication session must still be initiated by the satellite and only up to 270 bytes are permitted. Multiple uplinks may be sent to the Iridium network where they wait in a queue to be fetched by the satellite. Messages sent cost credits, which can be refilled from the web portal. The included library supports all the necessary functions as well as additional callbacks and debugging functions for ease of integration into CubeSat architectures. For an in depth explanation of how to integrate this library, see the function specifications within the QLocate.h and QLocate.cpp files. This code should work with any serial Iridium 9602 based radio, such as the RockBlock Radio board, though due to its small size, the QLocate is usually prefered.

5.3 GomSpace NanoPower EPS

The GomSpace NanoPower EPS is a prebuilt integrated electrical power system designed specifically for CubeSats. It supports numerous functions required for satellites with solar power management, thermal sensors, configurable power outputs among many others. While the board is initially configured to work as a master device on the I^2C bus, intended for a multimaster system, the included library sets it as a slave device as it better integrates with an existing I^2C bus layout for less experienced designers and does not require all devices to support multimaster I^2C. An important functionality of this board is it contains a WDT which can be configured or kicked via I2C. It is set by default to 12 seconds. Fetching all system relevant values, kicking the WDT, and configuring the outputs are all supported, see the attached library for function specifications.

6. Summary

This work is intended to provide a brief introduction to the various design choices that arise for the new software engineer when designing a CubeSat or nanosatellite flight computer. The general structure and guidelines above are not exclusive nor applicable to every possible mission, but they provide a solid foundation for any new team to develop their own unique system without the hurdles of experiencing the same mistakes as the engineers before them. Cubesat missions have continued to get increasingly more and more complex over the years, which in turn demands more from the software designed to actuate them. As the complexity of the software increases, the importance of templates and design advice grows as well. Sharing of design advice and experience will continue to drive down the barrier to entry for spaceflight for new organizations across academia and private industry. I hope that this design document helps many in realizing the tremendous accomplishment of launching their own satellites into space.