Flight Software Description

The flight software for both the RT-IHU and the LIHU is an evolution of the original design for Fox-1a, with additions and removals to support new features, hardware, and capabilities.

The Operating System

The software is based on the operating system FreeRTOS, an open source real-time operating system designed for minimal overhead size and with ports to many different hardware platforms, including both the STM32L (the LIHU) and TI TM570 (RT-IHU). FreeRTOS has four basic functions:

1. Tasks and task switching
2. Queues (which also act as semaphores and mutexes)
3. Timers
4. Memory management

Of these four functions, the Fox and Golf flight software use only the first three. The OS itself allocates some memory for data structures when tasks, queues, and timers are created, but none of these structures are ever deallocated.

I/O

Note that the OS has no specific support for I/O, either generalized IO routines such as open, read, write, or even printf. In general we wrote drivers for the buses available (I2c, SPI and to some extent CAN) as well as GPIO, ADC, and DAC as routines that are called directly from "user" code--in other words with no common layer--but with as many similarities as possible.

One unwavering goal was that unlike many realtime code implementations we wrote everything as interrupt-driven so that there are no code loops waiting for an event to complete. Instead, we used the OS semaphore and mutex features to block the calling task until the I/O was complete. This allows another task to run. The LIHU I2c driver is somewhat different. It was written by someone else and while it does use semaphores, it repeatedly checks for completion in a loop, although it does use a wait call to allow other tasks to continue.

The level above the I/O drivers is called "Device Support". This is the code that understands specific devices on a bus. For example, the routine nonvolManagement understand the MRAM device, and the codes sent to it over the SPI bus, and thus does all the actual SPI driver calls.

On the LIHU, one exception to the general behavior are the D/A and A/D converters. These devices are set up to do DMA to/from the device from/to a double buffer memory continuously, interrupting when each buffer is complete. A task is then responsible for emptying or filling the newly completed buffer while the other buffer is being filled or emptied.

Semaphores, Mutexes

These are very similar in that code can "take out" (mark in-use) a semaphore if it is not in use. If a second task attempts to take it out, the task blocks until the first task frees the semaphore. In FreeRTOS, a mutex is nearly identical to a semaphore with the following exception: If a high priority task is blocked waiting for a mutex that is held by a lower priority task, the priority of the mutex holder will be raised temporarily to that of the waiter.

Generally a mutex is used to block another task from even starting to operate on a device, while a semaphore is used to wait the task that started the I/O between steps of the I/O operation.

Queues

We use queues to pass messages between tasks and between a driver and a task (for example to notify the task of an interrupt). A queue is a FIFO buffer of a defined (per queue) size. If a task tries to take a message out of an empty queue, the task blocks. (Most tasks operate in a loop which waits for a message and then performs some operations based on the message).

A bit more about delivering interrupts to a task: Most I/O is initiated by a task, and as described above, a mutex or semaphore is used to wait the task until the I/O (or series of I/Os) is complete. However some input I/Os are totally asynchronous, for example, characters typed, CAN messages received, or changes to the coordination lines from the other processor. In all these cases, when you initialize the device, you specify a task message destination and a message to be sent when the interrupt occurs. The serial I/O is a special case in that the incoming character is actually placed directly in a queue.

Tasks

A task in FreeRTOS is somewhat different from a process in Unix. In fact it is more like a thread, in that all tasks execute in the same address space and generally have no protection from each other. A task's main advantage is that it can block and restart anywhere. Thus you can write code for a task without usually considering what other tasks might be doing. On the other hand, tasks are not protected from one another. If you overflow an array or allocate too much on your stack, you might well damage another task's data.

FreeRTOS is configured so that tasks are non-preemptive. That means that one task will never actually interrupt another task. Task switching only occurs when the executing task blocks (via a mutex, semaphore, or a "wait" or "yield" call). This is important because it means that code does not need to be as careful about updating shared variables as it would have to be if the task might be interrupted in the middle of an update or some other form of critical section. The disadvantage is that tasks which do something that is time-critical (for example "Audio" or to a lesser extent "Downlink Control" and "CAN") may not get sufficient time to execute unless other tasks cooperate by placing "yield" calls in sections that might execute for a long time.

Watchdogs

A watchdog is a hardware timer which, if it is not regularly reset, will force the entire processor to restart or power cycle. We use two watchdogs: An internal and an external watchdog. The internal watchdog is part of the processor hardware and will force the processor to restart when it fires. The hardware watchdog is external to the processor and if not reset in time will actually power cycle the entire processor.

Watchdog Operation

Each task is required to call "ReportToWatchdog" regularly, either passing the task's ID, or telling "Report" to get the task's ID from the OS. ReportToWatchdog sets a bit to indicate that this specific task has "reported in". An OS timer also calls the watchdog management code regularly to reset each watchdog. If the timer routine sees that every task has not reported in within a specified interval, it does not reset the watchdogs. The external watchdog has a longer timeout period than the internal watchdog, so if the timer is working properly, the internal watchdog resets and reboots the processor. This is intended as a hedge against software bugs and has the advantage that some information is left in memory to be telemetered to the ground. However, if the internal watchdog itself fails, the external watchdog will cycle the power. This is intended as a hedge against single event upsets, but the power cycle wipes out memory.

In addition, there are routines for special purposes that can be called to reset the watchdogs "manually" or to simulate all tasks having reported. These are used before the timer is started, and before all tasks have been started.

Finally, in a few places we want to power-cycle the processor, and we do that by manually resetting the internal watchdog but not the external.

Software Operation

Use of manufacturer's software

Both the LIHU and the RT-IHU make use of the manufacturer's basic library (LIHU) or GUI-generated code (RT-IHU). While these routines are generally less efficient and take more space than hand-coded drivers, there are two reasons for this choice. First, it allows us to ask questions of the manufacturer, often through their user community. For both IHUs, they nearly always want to hear the question in terms of their own routines. "I called routine x and this happened" rather than "I set bit y in register x and this happened", and they respond similarly ("you could have called routine Z first). Second, it is usually easier to debug for those of us who are not intimately familiar with the complexities of the processors.

On each processor type, there are some exceptions, often in cases where a change is best made at runtime and/or the manufacturer's software is too inflexible. For example, in the RT-IHU the direction of a GPIO is chosen at runtime to enable the software to know which direction a GIO is going. In addition, the flight software chooses whether a GPIO is set to open collector mode or not. This is largely to allow us to tri-state a bus line when we are not in control, but we also do it for GPIOs that are permanently open collector.

Startup

The STM32L has less builtin self-checking hardware than the TMS570, so the startup code is quite different. The STM32L has a "pre-main" routine which uses no memory and as few hardware features as possible. This code tests the hardware, including a memory check, before switching to "main" which is willing to use the full set of hardware. If pre-main detects an error, it loops waiting for the hardware watchdog to power cycle in hopes that a power cycle will drain a possible deposited charge and allow it all to work again. This sounds a little hinky, but our longest-lasting satellite so far (AO-85) never had an obvious problem. (Note: Yeah, they said the same thing about *Challenger* and *Columbia*. We should consider a design to allow a command to be uplinked that will ignore POST errors)

Startup in both cases calls a few manufacturer-supplied init routines, and then creates the first task and starts the OS scheduler.

The first task is "Console". The reason for creating only it before starting the scheduler is that there are things that need to be initialized which require us to be running in an OS task. This includes initializing drivers and fetching the saved reset number from MRAM (and updating it for the next startup).

Timekeeping

Basic timing is done by a hardware timer that calls into the operating system at a rate of 100/second. Thus 1 centisecond is the minimum time that can be specified in any routine OS-related routine. However any wait time specified is only accurate to +0/-1 centiseconds.

On all satellites prior to Golf, we had no reliable UTC time source available. On Golf, we will have a GPS receiver, but for now we can't count on it locking in immediately, and it is important to have a unique timestamp on the telemetry. What we have done on the Foxes, and we will continue to do on Golf is to timestamp telemetry in terms of "epoch" (or on Fox "number of resets") and "seconds in this epoch". We will also send the GPS time in telemetry when it is available, so that we can correlate the reset/seconds time with UTC time by post-processing.

The timestamp is generated by keeping the next epoch number in non-volatile memory (MRAM). When we command "preflight init" on the ground before loading the satellite on the rocket, this value is set to 0. Thus the first reset number when it starts up in orbit is 0. At startup, we initialize the current seconds counter to 0 and the current epoch number to the epoch number read from MRAM. We then increment the value in MRAM for the next epoch change. Because there are three processors on Golf, which might reset independently, the epoch and seconds are coordinated among the processors so it is consistent and always increasing among all three processors.

The basic timekeeping code is in the module called MET.c. It is not a task, but it called by an OS timer once per second. This routine can also generate multiples of 1 second. For example, every 4 calls, it releases a semaphore to allow the Telemetry Collection task to run (see below).

Specific Tasks

There are eight tasks on each processor:

1. Diagnostic (aka Debug) or Console
2. Coordination
3. CAN Support
4. Telemetry Collection
5. Downlink Control
6. Audio or Telemetry Radio
7. Command
8. Experiment (the Vanderbilt experiment)

The main part of each task is in a separate module with that task's name. It should be noted though that a number of routines within that module can be called by other tasks, and thus these routines might actually be executing on behalf of other tasks.

Here are more details about each task:

Console: This task is not used in orbit and could actually not be run in orbit except for the fact that it is better use the same configuration in orbit that we use on the ground. Its purpose is to allow the developer or tester to issue commands to the satellite over a serial line. These commands have many purposes. Some (for example "get realtime telemetry") simulate an in-orbit operation and allow the tester to see the telemetry that is collected without using a radio. Alternatively some simulate a transmitted command so that the tester can check most commands without using a radio. Others (for example "preflight init") are used to set the state of the satellite for launch. And still others are just to allow a developer to see some of the inside state for debugging (for example, "show lihu state" or "enable comprint"). These commands change rapidly as they are needed during development. Help lists the commands along with a short descriptive text string.

The console task is the only task that does not (and is not required to) report to the watchdog. No harm done if it fails in orbit.

The console task works by having set of enums and a structure array where each element consists of a command text string, a help text string, and an enum. When a command is typed, the structure array array is searched for a match between what is typed and the command string field, and then the enum from the structure is used in a 'switch' statement. The enums are also used as the case labels. The command help lists all the commands in the structure along with their help strings.

Coordination On Golf-Tee there are two IHUs: The RT-IHU (which itself contains two processors) and the Legacy IHU (LIHU), which is very similar to that used in the Fox satellites as well as the LTM-1 module (which flew in UW's HuskySat-1, will be in UMaine's MESat, and can be available to other partners). The LIHU is space-proven and is intended to fly largely as a backup to the

newly designed RT-IHU. Only one of these processors is "in control" at a given time, and yet they must coordinate their time and states. The coordination task is in charge of that coordination. See the document "IHU Master Coordination" in the Golf-T SVN under Design Docs.

The coordination task runs basically on a 4x4 state table with columns indicating its current coordination state, and the rows indicating the state of the other IHU. The task, like most, consists of a loop that executes once and then waits to be awakened, in this case by a message. Messages that it sees are sent either by a timer (telling it to send a status message to its opposite number), a change in state of the opposite number which causes the task to check the state table for what to do, or the CAN task which notifies coordination that a coordination CAN message has arrived. The coordination task is also responsible for ensuring that some operations commanded by umbilical are actually performed on all IHUs, for example "set inorbit flag" and "preflight init".

If the in-command IHU fails, its state changes in the other IHU and the non-failing IHU can take over within a few seconds. If a ground command is sent for one IHU to take control, it can request the other IHU to drop control and do so, all via this task.

CAN Support: The actions of most of the I/O devices that we have drivers for are initiated by the controlling IHU itself. CAN is unique in that message can arrive asynchronously. We deal with the asynchronous nature by way of a special task that receives interrupts when a CAN message arrives, and which fetches the message from the hardware, looks at its ID to determine what type of message it is, and dispatches the message to queues in other tasks depending on the type.

The LIHU does not have a CAN driver because the CAN device is a chip separate from the LIHU which is accessed via the SPI bus. Thus it has a "device support" module that uses the SPI driver, and a GPIO that generates an interrupt and notifies CAN Support when something changes. On the other hand, the RT-IHU does have an integrated CAN device and an actual driver (canDriver.c). In both cases, though, CANSupport gets an intertask message when there is a new CAN message to read, it reads the message, and dispatches based on the type of CAN message, as determined by some bits in the ID.

The types of CAN messages (which are dispatched to different places within the software) are as follows:

- Telemetry. This type of message is sent to the Telemetry Collection Task
- Coordination. This type of message is sent to the Coordination task.
- UplinkCommandxxx. This type of message is sent to the Command task. (xxx is the command namespace; Ops, Telem, and Exp1 for example)

Telemetry Collection: This task's code consists of a loop that runs every 4 seconds by way of a semaphore that is released by a timer. It behaves differently depending on whether this IHU is in control. We will discuss the two modes separately, but first let us point out that there are really three processors--the LIHU, the active RT-IHU (which can access the bus) and the standby RT-IHU (which can not access the main bus--it has only a private CAN connection to the active RT-IHU). When we say "in control" or "not in control" in this description, the standby RT-IHU always counts as "not in control" and behaves essentially the same as the other "not-in control" processor.

The 'in control' IHU behaves similar to all the other Fox and LTM satellites in that it collects all the housekeeping data that is available locally over the I2c buses, SPI buses, or the CAN bus. In addition, on Golf-Tee, it collects data that is only available from the "not in control" processors by way of CAN messages that all IHUs send to each other. Such data would include the LIHU processor temperature (only available on the LIHU), the RT-IHU board temperatures (only available from the RT-IHU whose temperature is being measured).

Messages that arrive over CAN all arrive asynchronously from each other and from the Telemetry Collection task. Thus the Telemetry Collection task module contains a method for double buffering arriving CAN messages. Any time the CAN task finds an incoming message marked as "Telemetry" in the 'type' bits of the ID, it calls "incomingCANTelemetry()" in the telemetry module, which then distributes it to other routines to be buffered and assembled into a collection of telemetry that is either from the CIU, or from the opposite IHU with data only known to that CPU.

An entire set of health data is thus collected in another buffer which will be available to the DownlinkControl task for encoding and transmission. In addition, data that has min and max values are compared with the current min and max if necessary, stored as the new min or max.

Every 60 seconds the IHU saves a copy of the current telemetry into MRAM as Whole Orbit Data (WOD) along with the time that it was collected and the mode that the satellite is in (Health, Science, Safe, etc). It also sends a WOD record to the not-in-control IHUs. (If the LIHU is in control, the active RT-IHU passes the WOD data from the LIHU to the standby RT-IHU since the standby can not access the main CAN bus).

The behavior is somewhat different if the IHU is not in control. It also executes every 4 seconds, but only collects the unique data that only it can access. It then sends that data as a telemetry packet on the CAN bus to the other IHUs. Every minute it receives from the active IHU the WOD record. It stores this record in MRAM, but also uses this record to do its own min/max comparisons. In other words, the controlling IHU does a min/max compare every 4 seconds, while the non-controlling IHUs only compares every minute. The min/max values are likely not identical then, but are close. This avoids an entire data record being sent every 4 seconds OR the complexity of having the active IHU send only the min/max values that have changed.

Downlink Control: This task is one of the most complex in the system. (*Later BF note--the telemetry collection task and coordination task are also in contention for this award*) Its basic job is to manage the downlink both by determining when and what telemetry to send, as well as managing the transmitter's on/off state, whether it is acting as a transponder, whether it is sending telemetry and its RF power level. It uses as input for this management the current satellite mode (safe, health, science) as well as messages about timers, commands, and downlink operations.

Downlink Control is based on a state table with 9 rows corresponding to 9 incoming messages, and 7 rows corresponding to 7 internal states. The internal states are

- Health mode. Continually sending telemetry. Transponder on if it has been enabled.
- Safe mode no carrier. Safe mode during the time when the transmitter is turned off
- Safe Beacon. We are in the process of sending two frames of telemetry in safe mode
- AutoSafe. Nearly identical to safe mode except that we can exit automatically when the condition that put us here is lifted
- Autosafe Beacon. see above
- Transmit Inhibit. This is for the RT-IHU only. A transmit inhibit command has been received, and only a transmit enable will allow transmission again. (On the LIHU this is all in hardware)
- Science mode. Continually sending telemetry (but maybe different frame types from health mode). Transponder on if it is enabled.

The incoming messages are

- Idle Timeout
- Beacon Timeout
- Telem Done
- Enter Safe
- Enter health
- Enter Autosafe
- Inhibit Tx
- Enable Tx
- Enter Science

The table contains the next state to switch to enter when the specified message comes in. To change states, we call "State Change To" which not only changes the current state but performs whatever actions are needed to enter the new state, for example, entering "Safe Mode No Carrier" turns off the transmitter and starts a timer.

Downlink performs several other related tasks also. There is one additional message called "DwnlnkCollectTelemetry" which is sent by Audio (see below) when a buffer is freed up. When Downlink receives this message, it calls GetTelemetryData, which first determines what the next telemetry frame type should be based on the mode. The routine CreateFrame builds the frame by calling CreateHeader followed by CreatePayload for each payload type in the frame. Next GetTelemetryData calls routines to encode the telemetry with Reed-Solomon error correction and 8b10b line encoding. It then calls the telemetry buffer management routines to indicate that there is another frame ready.

Downlink Control also contains the routines for determining if we should enter autosafe, whether the transponder is enabled, and similar functions.

Audio or Telemetry Radio: This task is somewhat different on the two processors because of the different radios in use for telemetry and commanding. On the LIHU, it is divided into two parts: *Telemetry* is responsible for turning the telemetry bits given to it by the DownlinkControl task into digitally signal processed output samples in the D/A buffer to be sent as analog voltages to the PSK modulator on the RxTx card. The *Software Commands* section is responsible for collecting samples from the ICR command receiver, extracting a clock from them, forming the bits into bytes, and sending them to the command task.

In both cases the loop in the Audio/Radio task is based on when it is time to give the radio more data. In the LIHU, Audio manages a double buffer that the D/A converter scans and converts continually. The D/A converter sends the output to the hardware telemetry modulator. Thus, the Audio routine is responsible for placing samples in the buffer fast enough to fill one buffer before the previous one empties. Here is the outline of the *Telemetry* section for the LIHU:

Audio fetches an entire FEC and 8B10B-processed frame from Downlink Control. Each frame consists of approximately 600 "bytes". Each of these bytes consists of 10 bits. Each bit is converted to 8 samples, and the samples are placed in a buffer of 48 samples. So the loop goes: Fill the buffer with samples. If the buffer is full wait for the next interrupt. If we run out of samples, get the next bit and convert to samples. If we run out of bits, get the next byte. If we run out of bytes send a message to Downlink Control telling it the current frame buffer is free and get the next frame. Repeat all. There are a few complications in that the first "byte" of each fram is actually a 31-bit sync word, the bit-to-sample conversion is actually done by a digital low pass filter and there is some special code to deal with the safe mode case where we keep track of the last of the two frames sent and when it has been actually transmitted, we send a message (TelemDone) to the Downlink Control task so it can turn off the transmitter.

The LIHU *Software Command* section outline is as follows. Hardware demodulates and decodes the uplink into ones and zeros, but without any clock. An A/D converter samples the ones and zeros faster than the bit rate and does clock extraction by noticing when

the bit parity changes. The approximate number of A/D samples per command bit is known based on the bit rate of the uplink. A changing sample is assumed to be the start of a bit and the sample to be used for the command (you could say the clock strobe) is taken from the middle of a bit time and added to the end of a software shift register. We look for a synchronization sequence marking the start of the command, and when we see it, we start collecting bytes by shifting 8 bits at a time into the shift register. When the correct number of bytes have been collected, we send that buffer via a message to the command task.

For the RTIHU, the Radio routine is similar to the LIHU *Telemetry* section, but it responds to interrupts from the DCT and fills the DCT FIFO with the data. It must turn the 10-bit bytes into 8-bit bytes for the DCT, but it does not need to do the digital signal processing.

The DCT command section is TBD, but it will probably report directly to the command task without using the audio/transmitter task.

Command: The command task main job is of course to receive, decode, and act on ground commands. However, we use it for a few other timing-related things as well. For example, it keeps track of how long it has been since no command was received after powering up the first time, and it can go to a default mode if it never receives anything.

Without detailing the exact command features, (of which more are specified in the Command and Control project of RedMine), we can specify the general path of this routine. As described in "Audio", when a software command is detected, the bytes are buffered by Audio and sent as a message to "Command". "Command" undoes the line encoding, corrects bits using Forward Error Correction, checks that the command is intended for this spacecraft, and authenticates that the command came from an AMSAT Command Station. At this point we have a trustworthy command consisting of a namespace number and a number of bytes specific to that namespace. For AMSAT's name spaces (Ops, Telemetry) the bytes are divided into a command and 4 arguments. Each command with its optional arguments is acted on. It may be that it will be sent as a message to another task (for example a command to enter safe mode is sent as a message to Downlink Control, while a command to reset the Min and Max values is executed as part of the command task.)

Post Release (from Launch Vehicle) and Deployable Release

Post release timing and subsequent release of the "deployables" (antennas and solar panels) are two of the most critical jobs of the flight software. On the original Fox satellites, this work was all done in the pre-main section using as few pieces of the IHU as possible. Given our 5-satellite experience, the LIHU is far less of a concern that we had originally. In addition, the deployables on Golf (as well as HuskySat-1) are released by bus commands (I2c on both satellites, with CAN added for redundancy on Golf). This means we need more features of the operating system to reduce the need to write (and test!) special code to access these buses without the OS. Thus we go further into the boot cycle on Golf, LTM-1, and HuskySat before opening the deployables. The description is slightly different for the LIHU and the RT-IHU:

Wait and deploy on the LIHU -- On the Golf LIHU, we start the OS (including its internal timer task) and the console task first. Before console starts its main work as described above, we will call it the init task. Init does additional initialization as well as the post deploy wait and deployable release (if necessary).

Init first initializes the SPI ports to access the MRAM, Modulator, Gyro, and CAN, as well as some early GPIOs. If the umbilical or charger is attached, we do not do the post-release wait nor any of the deployable releases, and just proceed to start all of the required tasks and then execute the console task.

One requirement of Golf-Tee is that as much as possible, tasks that must be done by the RT-IHU in later missions be done by the RT-IHU on Golf-Tee with the LIHU as a backup. This includes the antenna deployment. In order to accomplish this with a minimum of resources in use, we do not start the coordination task during the startup phases, and thus there is a bit of "manual" work that the init task does to determine if it should release the antennas. Here is the algorithm:

- On the first startup in space, manually set the coordination lines to say "alive but not in control"
- Wait the required post launch delay time (around 50 minutes) plus a few additional minutes beyond what the RT-IHU will wait.
- On coming out of that wait time, check to see if the RT-IHU is still in control. If so, assume it has done its job and go on to start the rest of the tasks and operate as listed above.
- If the RT-IHU is not in control for some reason, we take control ourselves, and check the status of the deployables. If none of the required deployables are released, we execute the normal release sequence: For each deployable, we power the release mechanism and watch the status until the release is complete (or there is a timeout).
- If the sense status shows any deployables released, we don't know if there is a sensor failure or if the RT-IHU did only some of them. In any case, don't trust the sensor--just burn the deploy mechanism using time rather than trusting the sensors.

Notice that if the RT-IHU fails during the post-LV wait, the LIHU does not take over immediately because the whole coordination task is not yet running. It only notices the failure after it exits its own post-LV wait.

Timing The Wait

On both IHUs (RT and L) the timing is broken into serveral pieces. In the outer loop, "AlertFlashingLoop" is called for 1 minute at a time. When it returns, we subtract 1 from the total amount of time we are waiting, and write that to MRAM. If we should reset, we will start with that new value to avoid waiting forever on repeated resets (as long as they are 1 minute apart). AlertFlashingLoop waits by

using the routine WaitSystemWithWatchdog, which in turn breaks the wait request into 1 second intervals to call 'vTaskDelay' and then reset the watchdogs. (We may need a shorter time for the external watchdog).

Dealing with two-buses for releasing

Golf-TEE has two different ways of releasing the deployables and of reading the status of the deployables. Both use the same burn wire and the same sensor mechanism but there are two separate buses and GPIO-like devices to sense and release the deployables. One bus is an I2c bus. It connects to (among other things) a PCA9539 16-bit port expander on the CIU, i.e. a GPIO run by the I2c bus. 8 ports are initialized to read and 8 to write. The read ports read the sensors (ground=stowed, pulled up=released). The other bus is the system CAN bus, which is read by a PSOC processor on the CIU. The PSOC is programmed to convert CAN commands to read 8 pins configured as GPIO input ports and 8 configured as GPIO output ports. The outputs of both devices are "OR"ed to a transistor driver which enables the burn wire.

The software for releasing deployables goes through several levels, but the essentially parameters are:

- Which device(s) to release (a byte with one bit representing each device)
- Which bus to use
- How long to wait for the burn to complete
- Should we burn only until the sensor says the device has been released, or override and burn the full time

If BusI2c or BusCAN is specified, only that bus is used for burning and sensing. If the sensor appears to fail then the full time will be burned. If BusAuto is specified, both buses are simultaneously commanded to burn and both bus sensor are checked for completion. If override is not specified, both sensors must agree that the deployable is released before the burn is stopped.

When the burnwire is turned off, if the I2c or CAN command fails, the buses are reset, which should turn off the burn on either device.