Software Lessons from the University of Colorado's Student Nitric Oxide Explorer

Mark Andrew Salada[*], Sean Ryan†, John Donnelly†, Gail Tate†

Johns Hopkins Applied Physics Laboratory
Johns Hopkins University
Mark.Salada@jhuapl.edu
(443) 778-7267 (Baltimore)
(240) 228-7267 (Washington)

†Laboratory for Atmospheric and Space Physics
University of Colorado at Boulder
Sean.Ryan@lasp.colorado.edu
John.Donnelly@capela.colorado.edu
Gail.Tate@lasp.colorado.edu

Abstract

The Student Nitric Oxide Explorer (SNOE: pronounced "snowy") is a student built, low cost mission intending to demonstrate meaningful science in space for a fraction the cost of conventional satellites. The program is the first of two University Space Research Association (USRA) managed Student Explorer Demonstration Initiative (STEDI) programs, funded by NASA. The student team at the Laboratory for Atmospheric and Space Physics (LASP) in Boulder, Colorado, participated in every aspect of design, construction, test, and operation of the satellite. In particular, the flight software team successfully demonstrated the student team approach with a professional mentor that the program hoped to pioneer. The small team of one graduate and three undergraduates successfully completed the task with minimal supervision. The team implemented several novel ideas that establish software as the single most important key in low-cost small satellite success. Among these novel ideas, three design features stand out as "role models" for future missions at LASP, and perhaps for any other university organization intending to produce small satellites. These are the Power-On/Reset (POR) design that uses two software images, the Command Storage Management (CSM) design that sorts its own memory, and the telemetry design that employs programmable packets. The important thing, however, is to recognize the value of using a student team with a professional mentor, where the untapped student body resources on university campuses across the nation diffuse the cost of developing space systems.

A Mission Using Software

SNOE is the first of three projects for the University Space Research Association Student Explorer Demonstration Initiative program demonstrating student involvement with small satellite production. SNOE from the University of Colorado, the Tomographic Experiment using Radiative Recombinative Ionospheric EUV and Radio Sources (TERRIERS) from Boston University, and finally the Cooperative Astrophysics and Technology Satellite (CATSAT) from the University of New Hampshire are attempting useful science in space for about $4.5M, not including launch vehicle. SNOE successfully launched in February of 1998. The TERRIERS satellite launches in December of 1998. The scientific objectives for the SNOE satellite validate a temporal relationship between nitric oxide density and solar soft x-ray radiation in the thermosphere. The additional, complimentary objective for SNOE is to prove that a complete satellite is possible with a student team for a tenth the cost of most satellites. LASP intends to demonstrate to industry that the use of students for every phase of the mission is an acceptable, even beneficial approach. The NASA 'smaller, faster, and cheaper' principle faces a strenuous test by our program. SNOE is a 254 lb., 580x550 km orbit spinning satellite that scans the limb of the Earth for nitric oxide spectral emissions with a UV-spectrometer while simultaneously measuring the soft x-ray emissions from the sun with photometers. A third

[*] Mark Salada was a Graduate Research Assistant at the Laboratory for Atmospheric and Space Physics in Boulder, Colorado during SNOE's design and construction, prior to accepting a position at the JHU Applied Physics Laboratory in Maryland.

nadir pointing instrument measures auroral emissions over the Earth's poles. It's very possible to complicate a small satellite from the start with over-ambitious and unrealistic science objectives. An important and perhaps vital reason for SNOE's success is the principal investigator's clear and simple science goals. The science team greatly reduced the amount of data handled every pass, with only two ground passes a day. For example, instead of using a LASP CODACON™ imaging device for snapshot of the continuous spectra of nitric oxide emissions, the science team reduced their data requirements to only two important spectral bands. This decision had a profound effect on the complexity of the instrument detectors and electronics, with corresponding simplifications throughout the spacecraft. Key decisions very early in the project, like the removal of the imager, paved the way to later successes.

For example, spacecraft designers placed a great deal of spacecraft functionality in software as opposed to custom hardware components. A single commercial processor coordinates all instrument science, health and safety, and command and data handling functions for the spacecraft, instead of the conventional multiple processors and hardware boards to do the same. Instead of adding a redundant processor, the means to switch memory mediums, and hence software images within the same computer accomplishes fault redundancy without the doubled hardware cost. The stored command approach relieves mission operations from tedious memory management by "sorting" the up-linked commands on-board. Actually, the software "indexes" the commands, as opposed to sorting, for increased performance. This management handles both the absolute time-tagged commands as well as the concurrent 'block' commands, which execute relative to one another in time.

Finally, the SNOE telemetry design embodies the full spirit of the Consultative Committee for Space and Data Systems (CCSDS) packet telemetry recommendation, but with a twist. Mission operations has the ability to literally select the contents of two packets before and after launch, and reconfiguring them as the mission matures. This feature enables debugging during integration and test, and most importantly during flight. This paper is a follow-up to a previous paper that describes the design of the flight software prior to launch, "The Student Nitric Oxide Explorer (SNOE) Telemetry and Flight Software Design."[i]

A Brief System Summary

The software designers chose ANSI C as the embedded system language, with in-line 80C186 processor assembly where appropriate. SNOE runs its own operating system, a deterministic linear scheduler with vectored module calls. Using a commercial compiler and linker, Borland C++ 4.5, developers employed the 'large' memory model to facilitate memory patching (described below). Another commercial product, Paradigm's LOCATEâ, enabled explicit control of the final embedded image. Using a 386 development computer, and a 186EB Intel Evaluation board, programmers designed, coded, and tested all software with an incremental release approach. The first release was the image in UVPROM. The final releases were all varying levels of functionality in EEPROM. There are about 10,000 lines of code in the entire design. With three undergraduates on the team, the division of labor roughly allocated one student to commanding, telemetry, and memory/POR each. Only one student, now a LASP professional, remains on the SNOE team supporting the flight software during mission operations.

Power On/Reset

The most popular buzz phrases engineers use to boost the appearance of quality in a system are 'fault tolerance,' and 'redundancy.' Considering the large amount of money spent on today's satellites, most engineers take these phrases very seriously. For the small satellite designer however, these phrases lose their luster in the face of monetary phrases like 'out of funds,' or 'broke.' Only very few systems on SNOE are redundant. There are two battery packages, more solar panels than are necessary, and fail-safe activation switches for initial turn-on. Like other small satellite budgets, the sheer cost of spaceflight computer hardware preempted SNOE from using a redundant computer (CPU). To regain some of the hardware redundancy lost by using a single on-board computer, SNOE incorporates a double boot path within the computer for power-on and reset. Specifically, two separate software programs exist in the SNOE computer, each with the capability to carry out the minimum scientific requirements for the mission. The custom hardware interface board houses the capability to decode a ground command to switch between the two images at reset. Furthermore, these separate images reside in physically different memory mediums, increasing the 'fault tolerance' by decreasing the types of failures that will completely disable the system.

The Southwest Research Institute (SwRI) SC-4A flight computer houses 64k bytes of UVPROM, and 256k bytes EEPROM. Upon delivery, the UVPROM held only the monitor code as developed by SwRI. The SNOE team replaced these chips with the minimally functional software image. This image is perhaps the most rigorously tested component in the entire spacecraft. In UVPROM, the flight software handles all commanding, telemetry, and EEPROM programming. Although difficult, the image in UVPROM alone can complete the scientific program. (The difficulty arises in an instrument pointing requirement, and the difficulty of maintaining it in view of the Earth oblateness and atmospheric deviations.)

Ideally, two separate teams develop and verify these flight software images to add to reliability. However, the SNOE program accomplished both sets of flight code with the same student team. At launch, the UVPROM image measured 58k bytes in size, and the EEPROM image was 120k. For initial turn-on and any subsequent 'cold' boot, the default software image is the UVPROM image. During the early orbit campaign, the operations team felt confident enough to 'switch' over to the EEPROM image on the fifth contact (within 10 hours of launch). Both images continue to work exactly as expected. To date, there have been no software-related computer resets.

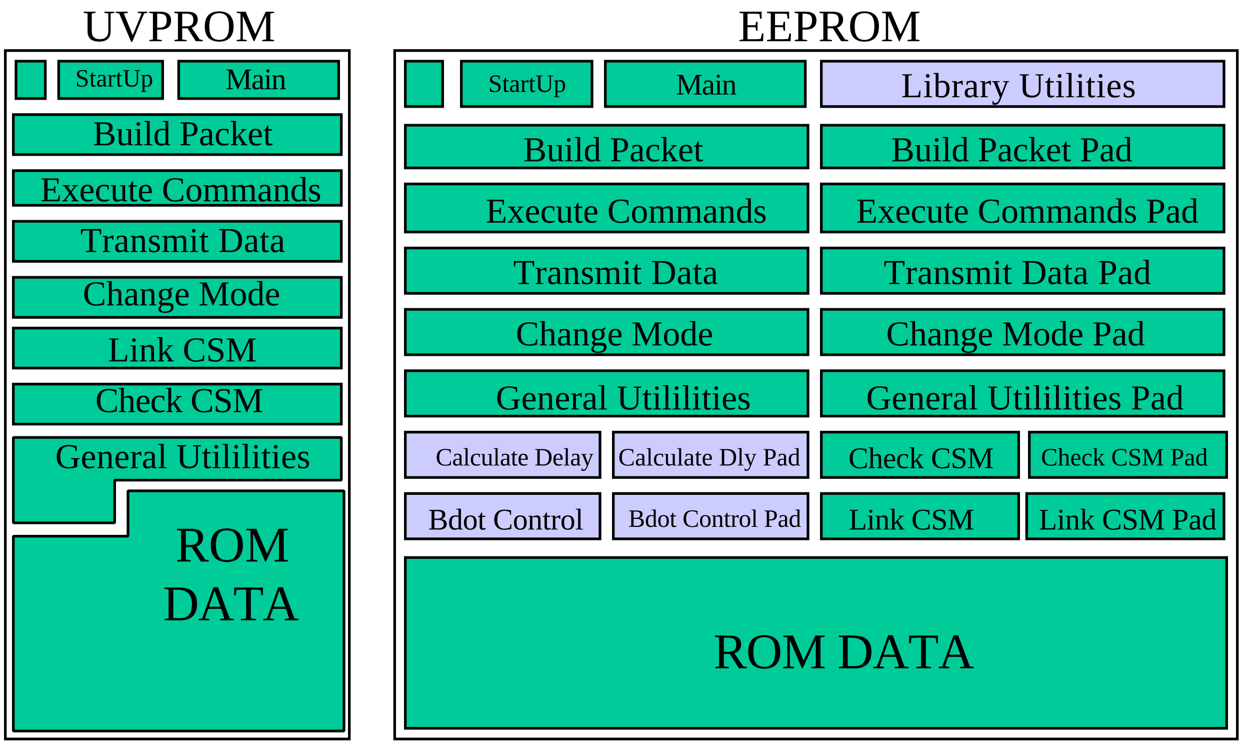

Figure 1. The UVPROM image and the EEPROM image each contain the minimal set of flight software functionality for mission success. The EEPROM image contains additional modules for closed loop attitude control and instrument timing corrections (highlighted blocks). Note that every EEPROM module that the software team expects to mature on orbit has a pad immediately following the original. The ground can load a new version, possibly patching a new module up to twice the size of the original without affecting the rest of the image.

The SNOE power on and reset approach demonstrates, by including the means to switch memory mediums within the same computer to execute different images of flight software, cost effective small satellite redundancy and fault tolerance without the doubled computer hardware cost of two CPUs.
SNOE uses the spacious EEPROM in a more sophisticated way. In addition to the minimal set of telemetry, commanding, and stored commanding functionality, EEPROM includes additional instrument pointing adjustment and attitude recovery modules. Not only is the functionality more sophisticated, but the physical placement of all modules takes advantage of the increased space. The arrangement physically separates all modules from each other (with a 'pad') to allow for 'patching' in flight. This way the ground can add new software functionality, replacing individual modules without re-loading the entire software image (See Figure 1). Most importantly, the EEPROM image corrects the annoying software problems identified in the rapidly developed UVPROM flight image. Of course, none of the problems encountered in the UV image were mission critical.

Stored Command Management

All satellites require, in some facility, a means to execute time-tagged commands to support non real-time activities. In this regard, SNOE is not unique. The command storage management designers chose a capacity of 128 stored commands. These 128 commands execute at an absolute time, relative to UT, and are aptly named absolute time-tagged (ATT) commands. Added to this capacity are three 'block' commands (A, B, and C), where each block holds 16 commands. The commands in each block execute relative in time to the start of the block. The starting UT for a block is a separate command, 'start block A,' presumably located in the ATT command buffer. All three blocks can execute concurrently with the ATT commands (See Figure 2).

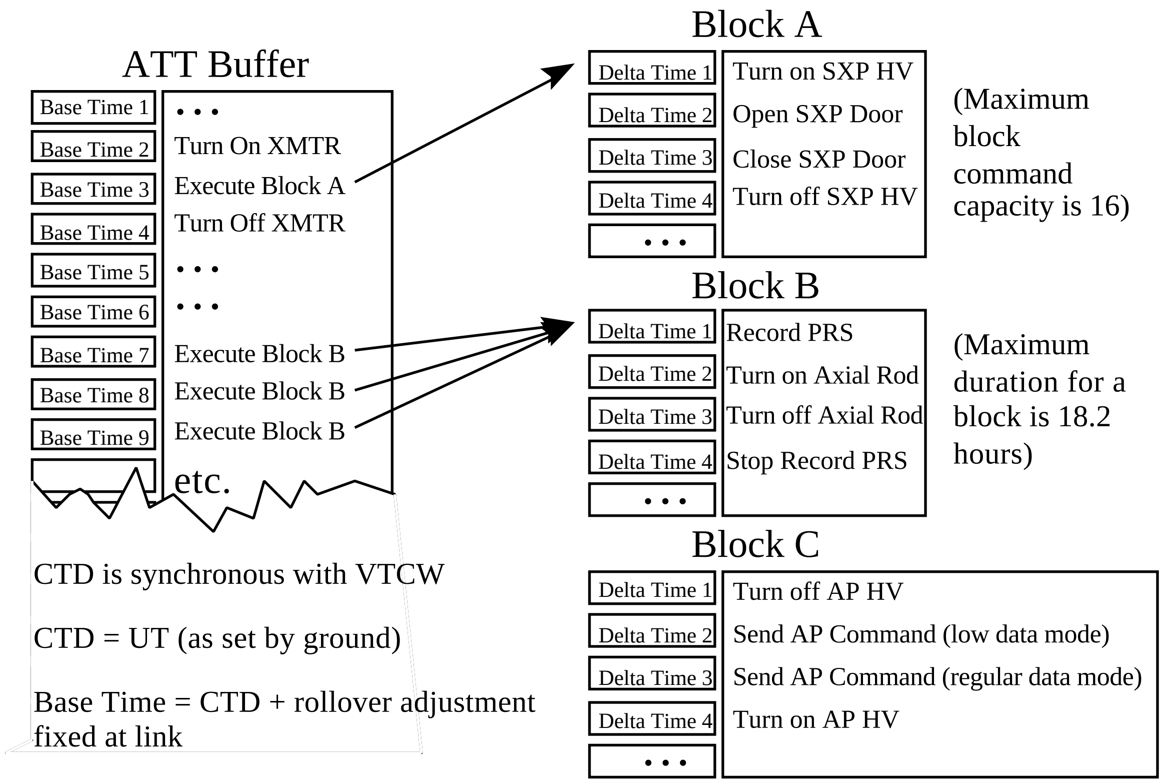

Figure 2. The command storage management buffers sort and execute from a base time calculated on-board at time of link. The actual base time tags are not visible to the ground. The ground sends all stored commands with CTD tags and then commands the link. The system manages its own memory spaces, continually cycling empty slots for new commands. The designers intended the block commands to act as short scripts to reduce the volume of commands in the ATT buffer.

It is also possible to 'link' the blocks by placing a 'start block' command in one of the other blocks. In theory, this provides an extensive relative command capability. However, the primary intent for block commands is to reduce repeated sets of commands in the ATT. For example, to conserve the sensitivity of the Auroral Photometer detectors, the satellite operations team may choose to cycle the high voltage on the instrument during the daylight. The turn-off and turn-on sequence is a perfect candidate for the block buffers since it contains about a half dozen commands to execute in sequence (See Block 'C' in Figure 2). With approximately fifteen orbits a day, the sum total commands would consume a majority of the ATT command buffer. Utilizing the block commands, the ATT needs only a total of fifteen 'start block' commands.

Two key decisions drove the CSM design. The first decision was to constraint check the commands at execution time, rather than up front during the sorting. In retrospect, this razor cut proved to be the one most in Occham's spirit. Piping all stored commands through the regular real-time command processing engine at time of execution greatly simplified the system. There was therefore no need to develop and test a separate command acceptance package for time-tagged commands. Once past the hardware checked error codes, and then past the CCSDS command checks, the uplinked stored commands reside on board untouched until their time-tags expire. Only then does the software check for constraints such as duplicate commands, contradictory commands, and bad command fields. The down side is that stored commands uplinked to the spacecraft during a pass may only later manifest themselves as bad, or poorly constructed. For event critical stored command actions, this can be fatal to the mission objective. However, due to the principal investigator's mission planning, SNOE's science schedule is periodic, and unrecoverable event-critical stored command events on subsequent orbits are rare.

The second key CSM design decision came more out of need than out of clever designing. One of the most difficult issues for stored commanding is marking time with finite-size counters. Any sixteen or 32-bit counter has the unsettling feature of rollover. In the flight software itself, this does not pose much of a problem. Most programs easily manage large counters. However, the limiting width in a command time-tag is not on board, but actually in the command itself. For example, the width of the time tag for relative (block) commands is 16 bits. At the one Hertz command poll rate, this limits the delta time to just over 18 hours. SNOE mission operations would like (understandably) a 24 hour delta time ability that better matches the ground passes schedule. Due to the width limited duration, maintaining a continuous time reference for commanding is a challenge. The on-board commanding reference for the ground is the fully adjustable Command Time-of-Day (CTD) counter that is synchronous to the hardware Vehicle Time-Code Word (VTCW). The CTD is a 22-bit, one Hertz counter, which the ground uses to synchronize the VTCW with UT. Of course this means that, at any POR, the ground could set the CTD close to its modulo, and cause a discontinuous hiccup in the command time reference before the maximum duration. To avoid the hiccup, SNOE stored commanding maintains a virtual time for commands in the ATT command buffer that is not visible to the ground. This time, the base time, is the actual reference by which commands will sort and execute. The algorithm correlates the base time to the command time-of-day at the same time it sorts, or 'links,' commands. The base commanding reference takes full advantage of the width of all counters, avoiding the discontinuous rollover of the CTD.

The flight software maintains the same virtual, 'base,' time for the ATT command buffer and block reference. Because of this, the CSM cannot link (sort) commands while a block, or blocks execute. The time reference for commands within the block would be out of alignment with those that have already executed at time of sort. Mission operations registered this limitation as a complaint well before launch. The preferred arrangement would allow for a re-link of ATT commands while one, or any of the blocks still has pending commands. To accommodate linking commands while a block executes, the flight code would have to maintain a separate base time for each of the blocks, in addition to the base time for the ATT command buffer. The complexity involved dissuaded the flight software team from implementing separate base time for each stored command buffer. However, the natural enhancement to the design would provide for multiple base time references.

Lessons from a Past Mission

The SNOE stored command design pays its respect to the lessons learned on the previously successful LASP and University of Colorado endeavor, the Solar Mesospheric Explorer (SME). NASA launched SME in 1981 with a two year expected lifetime, and continued to operate until 1987, teaching students for over six years. As the first student run satellite ever operated entirely from a university, SME taught the novice and expert alike that in-flight design features, good or bad, magnify ten-fold on the ground. A key lesson from those days is that the complexity of memory management for stored commanding consumes a great deal of ground resources. The most tedious ground activity was tracking the physical memory locations, and status of uplinked commands.

One of the main goals in the SNOE CSM design is to remove the visibility of physical memory addressing for command storage and execution. The flight software team accomplished this design goal using an indexing sort algorithm for all CSM buffers. The algorithm comes from Numerical Recipes QuickSort indexing routine[ii]. The algorithm required two additional indexing arrays for the sort, and the team introduces another index array for management of free slots.

The basic approach is to sort the commands while continually cycling the physical memory addresses once commands expire and execute. The CSM statically declares all stored command memory, and each command is the same size. The flight software prevents memory loads from stored commanding. As mentioned before, the commands received on-board pass hardware and preliminary protocol checks before the CSM has access to them. At link the flight code only examines the time tag, and the field that delineates between ATT and block command types. Performing no other checks at that time, the software creates the base time references, and sorts. This is the only functionality at link time. To complete the stored commanding system, recall the initial razor cut in the design. Each second, another software module simply queries the top-most available command in each of the stored command buffers. The module then places expired commands in the regular pipeline for command processing. The module continues to pull commands out of the buffers until it finds a time tag that has not expired. Command processing discards duplicate commands only at this time.

The software team based their decision to use the QuickSort algorithm on providing the ability to continually link new commands into the ATT command buffer while not to requiring the ground to uplink commands already sorted in time. The fastest sorting algorithm possible seemed the appropriate choice. In retrospect, even though the QuickSort algorithm is the fastest sorter available for arrays as large as the ATT index array, is has a worst case efficiency of N^2. Mission operations routinely links commands in time order, even though they are not required to do so. This exercises the worst case condition for QuickSort almost every link. Heap Sort[iii] may be a better algorithm choice to perform the sorting, with appropriate modifications for indexing. In any case, the QuickSort approach works perfectly. The time slice allocated for the link is 200 milliseconds, and the worst case link measured before flight was 91 milliseconds, real time.

There have been two patches loaded to the spacecraft since launch, both just one word in size. Each fixed a bug in the CSM functionality. Although not the most complicated set of code on-board, the CSM proved to be the most arduous to test thoroughly. Furthermore, mission operations, at time of writing, has not taken advantage of the block commanding. There are several reasons for their exclusion at this time, one being the near flawless performance of the ATT commands. Mission operations intends to utilize the block command functionality as time permits.

Programmable Packets

Telemetry design for SNOE started as a Time Division Multiplexing (TDM) system with bandwidth allocations nearly identical to SME. The designers assumed a static, deterministic telemetry allocation between engineering and science data, based on a finite number of telemetry modes. With packetized telemetry streams, a static declaration of bandwidth is not necessarily appropriate. SNOE software designers recognized the need for a flexible system and argued for a 'programmable' element to the telemetry very early in the program. The result is a pair of programmable packets in which ground operators interactively choose telemetry items, fill available 'slots' in generic packets, and send them to the ground at an adjustable rate. Note that these programmable packets are in addition to a set of fixed engineering packets, as described in reference [i]. Admittedly, this has an effect on the bandwidth that only probability and statistics analyze well. The added flexibility however completely outweighs any unknowns induced by an undeterministic system.

The Programmable Engineering Packet

The first step in enabling a programmable stream is organizing all on-board telemetry. The SNOE flight software places all telemetry items in a global structure, called the Master Telemetry Index (MTI). The MTI is actually an array of the following structure:

```
typedef struct {
      item_type type;
      union {
            volatile dword I;
            word huge *p;
      } value;
      word size;
      word PE_size;
      word DB_Address;
      byte Line;
      byte MUX;
      word Mask;
} Telemetry_Point_Type;
```

With all items intended to be visible to the ground located in such an array, the array index itself becomes the unique telemetry point ID. This organization greatly simplifies the engine that generates all packets. The engine merely loops through an array of MTI indices until it finds a unique identifier, End-of-Packet (EOP), progressively adding items to the data field of the source packet. There is an array of MTI indices for each packet on board, each carefully terminated with an EOP. This generic approach to packet construction is scaleable. Only the total number of elements in the MTI array limits the total number of telemetry items selectable. Note that there are two fields for the size of the telemetry point, "size," and "PE_size." The .size field is the actual size, in bits, of the telemetry point. The .PE_size is the item .size, rounded to the width of the most appropriate slot in the PE packet.

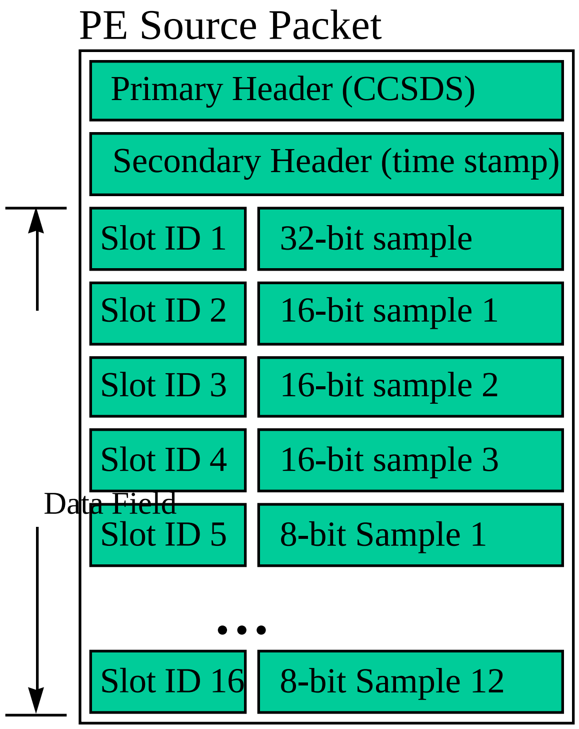

Figure 3. The Programmable Engineering packet data field contains alternating ID slots and samples. The IDs correspond to MTI indices, uniquely identifying each telemetry item. The OASIS-CC ground software uses the ID to update the correct database item in real time.

The PE packet is a standard CCSDS source packet, with a data field of alternating slot IDs and slots (See Figure 3). There are a fixed number of slots, all with predefined bit widths. These slots are actually telemetry points themselves, enabling the same engine that generates the other packets to also generate the programmable engineering packet. There are a total of sixteen slots in the packet. One 32-bit slot, three 16-bit slots, and twelve

8-bit slots comprise the data field. A slot ID field immediately precedes each slot. When filling the slots, the command constraint checker compares the requested item's .PE_size field with the selected slot size, and accepts the selection only with a match. As mentioned before, .PE_size fields are rounded-up values for the width of telemetry points. Each .size field rounds 'up' to one of the three allowed sizes in the PE packet. For items greater than 32 bits in size, the engine telemeters only the most significant 32 bits. For telemetry items less than 8 bits, such as a status bit, the engine zeros the remaining seven bits in the field. Prior to launch, mission operations selected default items for each slot, enabling the immediate transmission of the packet. Although clearly not the most efficient way to deliver telemetry, the flexible nature empowers mission operations in a manner not previously available on missions like SME. Once mission operations completes the packet contents, additional commands select the packet transmission period to anything between and including 2 seconds to 60 seconds, on the second. PE packet provides a quick look, or comprehensive minute-by-minute dwell over a long interval of time on any combination of items in the MTI.

One of the limitations of spacecraft telemetry systems is the corresponding ground station capability to parse the packets in real time. One of the SNOE challenges in the ground was handling the PE packet once it arrived at the Program Operations Control Center (POCC) in Boulder. Using a built-in feature, the mission operations team uses a LASP product, the Operation and Science Instrument Support Command and Control (OASIS-CC) software to automatically parse the packet, and update the appropriate telemetry item in the real-time database. OASIS-CC triggers off of the arrival of the packet based on the source packet application id in the primary header. Mission operators essentially never see telemetry from the packet itself. They only see the selected items updating at the newly commanded rate. This seamless integration of the programmable engineering packet proves most valuable during a real-time pass, where time is short. SNOE mission operations uses the programmable packet primarily during a ground pass to maximize efficiency.

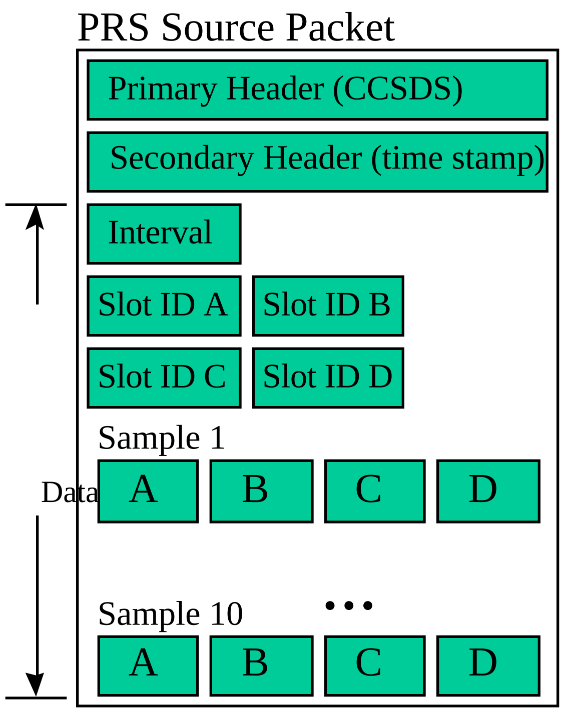

Figure 4. The Programmable Rapid Sample packet is essentially identical to the PE packet in function, but it handles a much higher sample rate for fewer items. Ground users reconstruct a sample time with the time tag in the secondary header, and adding the appropriate multiple of the interval time. The SNOE team uses the PRS packet mainly for detailed transient behavior analysis prior to and in flight.

The Programmable Rapid Sample Packet

In the event that the two-second transmit period for the PE packet is insufficient, the Programmable Rapid Sample (PRS) packet handles telemetry items at a much higher sampling frequency. With a selectable sampling frequency range of 5 Hertz, 2.5 Hertz, 1.25 Hertz, and 0.75 Hertz, the PRS packet provides a mechanism for detailed engineering debugging in flight. The packet design is considerably different than the PE packet. First of all, the increased sample frequency limits the number of selectable items to four. Furthermore, only 8-bit slots are available. With the decreased number of items, the packet data field is more compact. Deciding to place ten samples of the four items in a single packet ties the packet construction and transmission to a factor of ten times the sample period (See Figure 4). For example, a sample frequency of 5 Hertz yields a transmit period of two seconds. This way, the ground indirectly controls the transmit period, and the consequent bandwidth or storage impact of the packet.

Just as with the PE packet, the PRS packet contains the item IDs for the selected telemetry points. In this case the format groups the four IDs at the front of the packet instead of alternating between samples as in the PE packet. The real-time parsing of the PRS packet makes less sense. Only the last value in each packet will be visible to operators. The packet mainly provides a post-pass detailed analysis capability to small sets of items. Candidates for the PRS packet include solar array voltages when the satellite crosses the terminator, or instrument voltages and currents during a 'switch-on.' (All analog items on SNOE are a filtered 8-bit A/D conversion through the SC-4A.)

PRS packet construction is a slight departure from the generic packet generator since it has the merit of being the only packet not completely constructed at once. The same MTI IDs apply to the PRS packet, and the selectability appears the same to the ground.

A natural expansion of the SNOE telemetry design is to make all telemetry packets programmable in the spirit of the PE and PRS packets. More sophisticated query ability with the MTI could develop into a fully populated real-time database in space. The concepts for such an advanced flight system are not new, but perhaps industrial confidence in the approach lags behind the capability. To that, the SNOE team encourages industry to turn to the university student body for ideas and people.

Other Lessons from the SNOE Software

A detailed description of what the team did well on the software is not necessarily the only lessons learned during the development. There are, of course, a few regrets. A great deal of time and work may have been saved had the team allocated more effort to the POR, or 'boot,' design earlier in the program. Initially, the emphasis for the software team focused on commands and telemetry, relegating the boot design to an "after the hardware is finished" position. This caused more late hours, loud conversations, and basic turmoil than any other aspect of the flight software. The SNOE team hereby encourages small satellite software developers to stress the hardware developers as early as possible concerning necessary boot sequences, safing, and start-up modes. When in doubt, keep it simple.

Another "gotcha" that developers regret is absence of a basic memory scrubbing utility that might identify Single Event Upsets (SEU), and correct them. The team considered the utility, at the time, an unnecessary task. In retrospect, being able to identify when SEUs occur during an orbit might lead to safer operations. The utility's absence does not endanger the mission in any sense, but environmental studies are now impossible without uploading a module.

Finally, the most painful lesson is not a technical lesson. When choosing a software development team, using students that don't have software backgrounds introduces a difficult quality assurance burden. We felt that hiring students with electrical engineering, or computer architecture knowledge might aid with embedded system programming. However, computer science students, although not well versed in satellite design, handle complicated programming tasks most readily.

Conclusion

This paper highlights just three elements of the flight software: the dual-image power-on/reset, the self sorting stored commands, and programmable packets. Briefly, the major lessons from the experience are:

- Use two software images instead of two computers.
- Sort stored commands on board.
- Accommodate *separate* absolute and relative non real-time commanding capabilities.
- Organize on-board telemetry to allow for programmable packets.
- Use programmable packets!
- POR requires a major effort
- Include a memory scrubbing utility

- Computer science students make the best programmers

Other features of the flight software, such as on-the-fly reprogramming, the 'large' memory model organization in EEPROM, and the automatic instrument delay correction are worth mentioning. Furthermore, the mission operations capabilities that the SNOE team demonstrates daily warrant commendation. For more detail on the SNOE satellite, visit the SNOE homepage at http://lasp.colorado.edu/snoe. Also, satellite engineering telemetry is available, completely summarized and plotted, *automatically* updated within thirty minutes of a pass at: lasp.colorado.edu/snoe/data.html.

The single most important lesson from the SNOE project is this: using students to make flight software for spacecraft *works*. During the Solar Mesospheric Explorer days in the early 1980s, the LASP management established itself as the experts in running student employed teams. LASP re-establishes itself as a leader in university-industry cooperation with the SNOE team success. The flight software exceeds the minimal mission requirements and proceeds to raise the level of functionality expected from small satellite software systems. We expect that, at the very least, on-board memory organization of stored commands, and some sense of programmable telemetry will become common flight software features, if they are not already. Industry should be encouraged to, and feel compelled to use students in every aspect of spacecraft construction. It is an inexpensive resource industry cannot afford to ignore, especially since today's graduate and undergraduate students are more computer proficient than ever.

---

[i] Salada, M. A., and Davis, R. L., "Student Nitric Oxide Explorer (SNOE) Telemetry and Flight Software Design" Supplemental Proceedings of the 9th Annual USU Small Satellite Conference, Logan, UT, 1995

[ii] Press, W. H., Numerical Recipes in C: The Art of Scientific Computing Second Edition; Cambridge University Press, New York, 1992; pp.338 - 341

[iii] Ibid. pp. 336 - 338