A Packet Based, Data Driven Telemetry System for Autonomous Experimental Sub-Orbital Spacecraft

J. R. Kalibjian
Lawrence Livermore National Laboratory

Keywords

autonomous sub-orbital spacecraft, data driven telemetry system

Abstract

A data driven telemetry system is described that responds to the rapid nature in which experimental satellite telemetry content is changed during the development process. It also meets the needs of a diverse experiment in which the many phases of a mission may contain radically different types of telemetry data. The system emphasizes mechanisms for achieving high redundancy of critical data. A practical example of such an implementation, Brilliant Pebbles Flight Experiment Three (FE-3), is cited.

Introduction

The design of a good telemetry system for an autonomous spacecraft must insure the ease with which new data types may be added to or removed from the telemetry stream and the ease with which that new data may be decoded. To achieve this a packet/priority based flight telemetry system is recommended where data may be submitted for telemetry in raw form or inside structures known as circles (circles hold a short history of a data value; thereby, providing data redundancy for that item). The telemetry decoder should only be able to recognize a data packet and depend on an input file to bind data structures to the data being carried inside the packets. It should have no knowledge a-priori about the data being telemetered. After binding type information to the data, the decoder can output the data in a standard format. An interactive data analysis tool (which can parse the output of the decoder) can be employed to display the data in a graphical form, or perform further reduction on the data.

Flight Telemetry Services

The flight telemetry system must provide the flight software tasks many services. They are: *TELEMETRY_START* - Initialize telemetry system; *TELEMETRY_SEND* - Inject data items into telemetry stream. Items that must be specified in this request are

1) Data identifier (unique integer), 2) Data size (number of words in telemetry request), 3) Location of where data to be telemetered can be found, 4) Frequency at which data is to be injected into telemetry stream, 5) Total number of times data is to be injected into the telemetry stream over the specified period in 4. (for many data items this number will be "forever"), and 6) The priority of the request. Observe that when a telemetry request is generated, it may implicitly induce data redundancy into the telemetry stream by specifying a frequency above the update rate of a data value. For instance, if a data value was updated at 20Hz; but the telemetry frequency was specified at 40Hz, a 2X data redundancy would be achieved; *TELEMETRY_UP_ REQUEST_PRIORITY* - Up the priority of a previous telemetry request. The new priority as well as the location of where data to be telemetered is located must be included in this request; *TELEMETRY_ DELETE_REQUEST* - Delete a previous telemetry request. The location of where data to be telemetered was located must be specified in the request; *TELEMETRY_INQUIRE_ STATUS* - A mechanism for determining whether the telemetry system (hardware and software) is encountering difficulty in operation.

Telemetry Circles

Data redundancy may also be introduced by telemetering a short history of an item as opposed to one instance of it. This short history structure is known as a circle. The circle contains a header and a number of records which contain the most current sample value and prior values. The circle records may be one computer word or many. See Figure 1. The telemetry system provides two utilities for circle utilization:

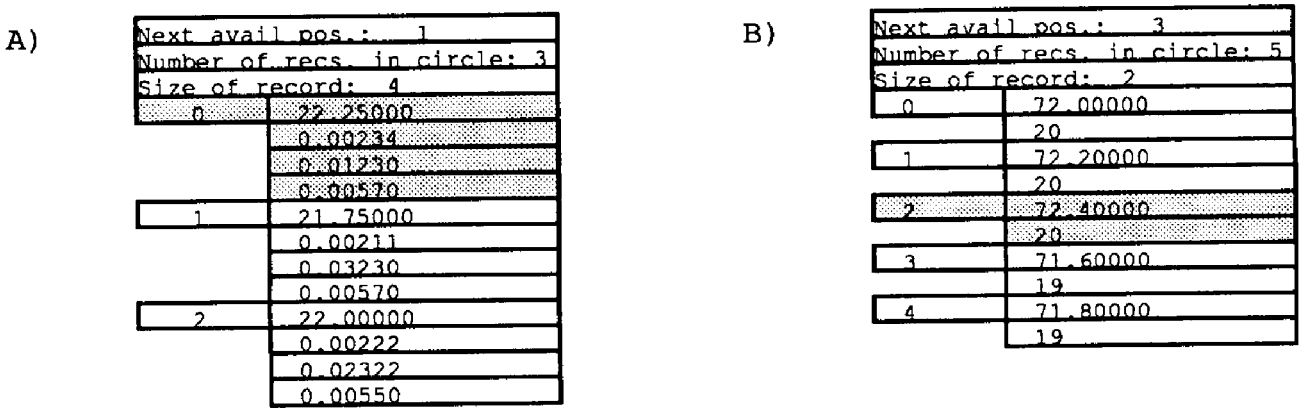

Figure 1. Examples of circles. In (A) the circle is comprised of three records each containing 4 variables. In (B) the circle is comprised of five records each containing two variables. In both cases, the shaded record has been most recently updated.

INITIALIZE_ TELEMETRY_CIRCLE - Initialize a circle to hold a short record of structure values. The size of the structure, the number of structures to be recorded in the circle, as well as a circle identifier must be specified; *UPDATE_TELEMETRY_ CIRCLE* - Copy new structure into circle. The location of the new structure (i.e. the new data sample) as well as the circle identifier must be specified. Observe that when circles are telemetered above the update rate of the data items in the circle, a high degree of redundancy is achieved.

The Packet Structure

When data is telemetered it will reside inside a packet which will contain a packet header and a packet trailer. The packet is of variable length up to some maximum size. The packet header contains the following: *packet Sync* - series of unique bits which identify a packet; *data id* - a series of bits which identify the type of data telemetered; *bit count* - unsigned count of the total number of bits making up the packet; *packet number* - unsigned count indicating which packet (in the series of packets the original data was broken into) the current packet is. Examples of multi-packet data would be images; *request id* - a unique series of bits assigned to the packet. Generally, this is simply a monotonic count. The packet trailer will contain the *data id, bit count, packet number, request id* and a *checksum*. The redundant *data id, bit count, packet number,*, and *request id* aid in the reconstruction of packets if damage is encountered. Figure 2 depicts a packet as it has been described.

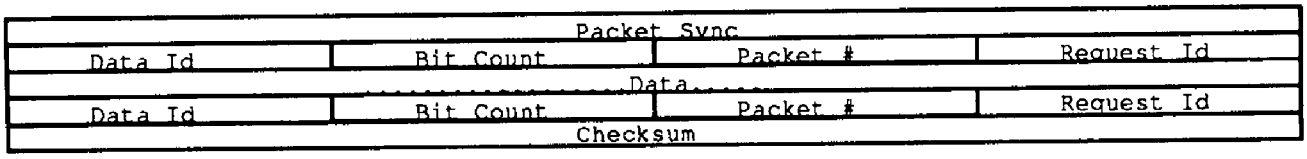

Figure 2. Recommended packet structure.

The Software Structure

Normally the flight telemetry software system will consist of two processes. The first process acts as the interface by which the flight software may request telemetry services (e.g. *TELEMETRY_SEND,TELEMETRY_DELETE_REQUEST*, etc.). This first process validates the request and creates internal structures that conveniently package the request for the second process, the packetizer. The request handler also has an internal queue that is used to hold requests which are to be injected into the telemetry stream many times over a specified interval. It is interesting to note that the internal queue is not necessary. The interface process may choose to use the same mechanism the other flight software tasks use for perpetuating a request sent to it. However, this has the effect of bogging the operating system down with many requests (assuming an event driven operating system). As long as the maximum desired data frequency is

above the minimum interface invoke frequency, the internal queue can be used with no degradation in data fidelity The savings in event traffic is obvious. For example, assume that eight different categories of telemetry data items require telemetry at a 20Hz rate throughout an experiment of 1000 flight seconds. If no internal queue is used, this will result in 160,000 OS events generated to satisfy the eight categories throughout the mission. If an internal queue is used (and we are guaranteed the interface process invokes at a 20Hz rate), the number of OS events required is reduced to 20,000 (i.e. 20 invocations over 1000 seconds). The above can be summarized with the following equations, given the following definitions,

T_d - Telemetry duration (s); T_f - Telemetry frequency (Hz); T_{sose} - Total savings OS events
N_{di} - Number of different data items being telemetered at T_f
Max_{ddf} - Maximum desired data frequency; Min_{iif} - Minimum interface invoke frequency

If $Max_{df} <= Min_{iif}$, the percentage reduction in OS events is given by,

$$\frac{T_d T_f N_{di}}{T_d T_f}(100) = N_{di}(100)$$

where the total savings in OS events is given by,

$T_{sose} = (T_d T_f N_{di} - T_d T_f)$ or $T_{sose} = T_d T_f (N_{di} - 1)$

Once a request has been packaged by the interface process, it must be delivered to the second process, the packetizer. The interface process and the packetizer may communicate in a number of ways; for instance, by a queue, by message, etc. The packetizer does what its name implies: it breaks telemetry data into packets. As this is being done the data is placed at a memory location which is accessible to the hardware that accomplishes the encoding of the digital data on the transponder carrier (this can be considered done in the transmitter). The packetizing process must be run at a frequency that will support the maximum data rate of the telemetry hardware. However, care must be taken to insure that the this process does not hog the CPU. To guarantee this, a time out must be selected for the packetizer, that will balance the need to get telemetry data out, and the Concern for not hogging the CPU.

When the packetizer is run it first checks its input queue to ascertain if new requests have been submitted. If new requests are detected they are removed from the queue and placed on an internal queue based on its priority. Before packetization can begin,

the process must first detect how much space is available in the region of memory used by the hardware encoder. Once this is known, the packetizer can then begin to traverse its internal queues on a priority basis. Within a priority level, requests are handled on a first come first serve basis. If a request is encountered that is too large, the next priority level is examined. This process is continued until one of the following occurs 1) a time out, 2) all the new encoder memory is exhausted, or 3) the request queues were traversed once. It should be noted that many other scheduling algorithms could be used in the packetizer, for instance, shortest job first (SJF), round robin (RR), etc. The priority based first come first serve (FCFS), with the jump to the next priority level when too large of a request is encountered appears to be adequate for most situations. Once the packetization is complete, the digital data must be passed to a transmitter for modulation onto a carrier. Figure 3 depicts the telemetry system discussed.

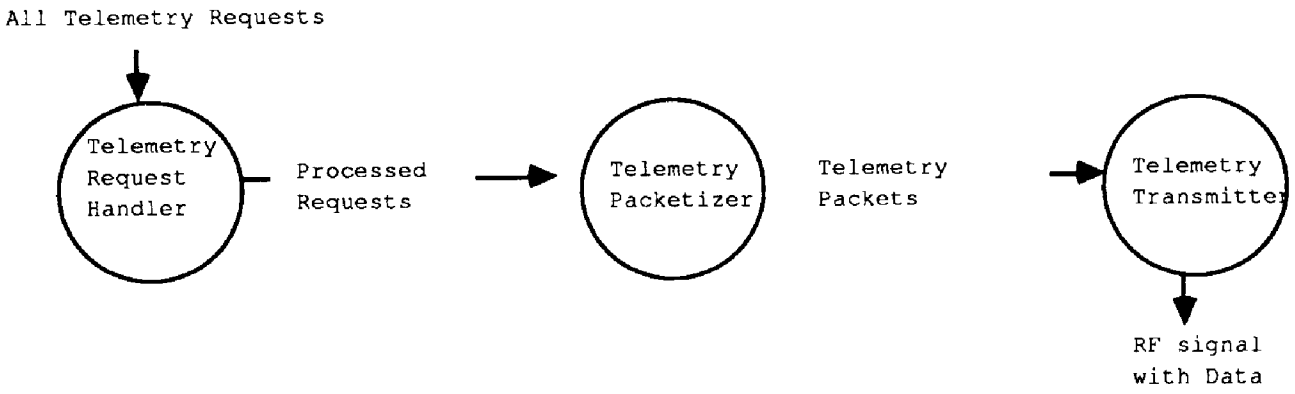

Figure 3. Core Processes in a basic telemetry system.

The Telemetry Decoder

The telemetry decoder has four primary functions i) byte gatherer, ii) byte comparator, iii) data formatter, and iv) error detector. The decoder knows nothing about the actual bytes comprising the packet sync until it reads such information from an input file describing the packet sync (known as a *ps* file). The same holds true for the data structures of the telemetered data. Such information is communicated to the decoder by another input file (known as the *types* file). This input file acts as a bridge between the flight software and the telemetry decoder. The decoder must also know which packets to extract from the telemetry stream. Many times (especially during testing) there is not a need to extract all packets. By reading a *ptp* file (packets to process file) the decoder can be told which packets are of interest. Once data is extracted from a packet, the decoder must verify that the data was not damaged during transmission. To accomplish this, a checksum of the extracted data is calculated and compared with the

checksum of the packet that was telemetered. In order for extracted data to be of any use, it must be in a format that can be understood by a plotting application program. Thus, all data files generated by the decoder (known as break out files) should be of the same format. The only exception to this would be for image or raw binary data.

Post Processing and Data Analysis

It is often necessary to further manipulate telemetry data after decoding before a plotting package is used. If any original break out files are manipulated and new files created, headers (in the newly created break out file) should be updated to reflect all operations performed on the data. The plotting utility itself should offer an easy mechanism for an analyst to examine data. It should be able to parse the break out file format and prompt the user for the data directory in which a telemetry break out has occurred. Once the user indicates this, the plotting package should display the names of all break out files it can find in the directory. The user can then select files of interest. The plotting tool should also allow the user to perform useful mathematical operations on any data element.

A Practical Example

The telemetry system discussed above was implemented for the third flight experiment in the Brilliant Pebbles test program (known as FE-3). The goal of the Brilliant Pebbles Program was to demonstrate an autonomous interception and destruction of a sub-orbital thrusting target by a light weight sub-orbital spacecraft.

The hardware system architecture was a two processor (R3000 based RISC) system with a scanner that could completely read a 256K portion of shared memory (between the processors) in approximately one-half second. The 256K was further divided into 4K blocks. The hardware made available a register that could indicate which of the 64 4K blocks was currently being read. The scanner unit read data from shared memory and turned it into a synchronized serial stream which was fed to an encrypter. After encryption, the serialized data was delivered to a transmitter where the data was encoded onto a carrier for transmission to a ground station. The transmitter output data at 4 Mbits/second.

The OS architecture was essentially a four level (timer, software, background, background1), message based system in which hardware control software ran at timer level, time critical software ran at software level, and remaining software tasks ran at background.

FE-3 Telemetry Implementation

Since the telemetry interface software did not involve time critical operations it ran at background. However, the packetization process had to keep up with the hardware scanner; thus, it was run at the software level. Because a dual processor architecture was used, nearly identical copies of the telemetry interface software and the packetization software were run on the second processor (known as processor B). This was necessitated by the fact that certain portions of processor B's memory (which would be telemetered) was not accessible to the first processor (processor A). Scanner bandwidth was allocated to processor B's packetizer by processor A's packetizer using a simple semaphore convention in another portion of shared memory not used by the scanner. The message passing paradigm could not be used to accomplish this due to the timing delays of using the message system at background, and the added complexity of communicating this information to the packetizer (which resided at the higher, software, priority level) .

The A processor's packetizer allocated bandwidth to the B packetizer based on monitoring the ratio of bandwidth it had used and the bandwidth it had granted to B. This ratio "goal" could be changed at any time of the mission to accommodate mission phase transitions in which the telemetry activity of processor A and processor B might become different. The facility for changing the ratio was placed in the telemetry interface; thereby, providing mission software the ability to easily manipulate bandwidth utilization by processor A and processor B. Of course, the privilege of changing this bandwidth ratio could only be given to software running on one of the processors. For this experiment, processor A's software was given this capability. Figure 4 depicts the processes comprising the FE-3 telemetry system.

FE-3 Telemetry Priorities

Five telemetry priority levels were used for the FE-3 experiment; namely, EMERGENCY_PRIORITY, CRITICAL_CIRCLE PRIORITY, INTERMEDIATE_ PRIORITY, IMAGE_ PRIORITY, and BACKGROUND_PRIORITY. The goal was to balance the need of getting out critical system performance data (e.g. attitude control system data, tracking data) with the desire to obtain phenomenology (i.e. image) data. Since in general the critical system data would occur frequently, but be made up of a small number of bytes and the image data would occur less frequently but require a large number of bytes, it made sense to place the CRITICAL_CIRCLE_ PRIORITY above the IMAGE_ PRIORITY. This was in effect

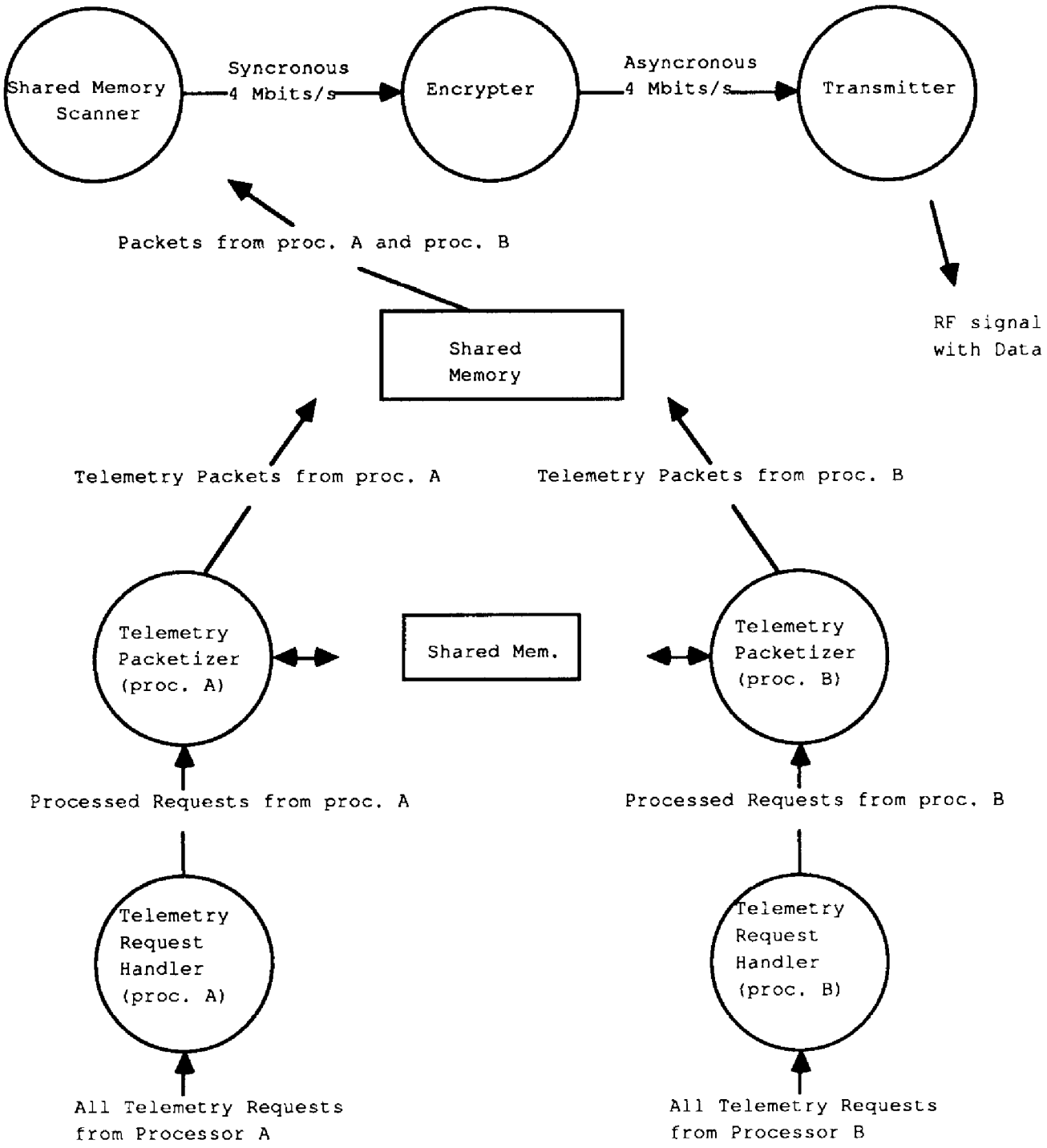

Figure 4. Processes Comprising the FE-3 Telemetry System.

saying when critical system data is not going out; telemeter images. The rational for the other priorities is straightforward. Un-expected events were telemetered at EMERGENCY_ PRIORITY to insure the best chance of telemetry before system failure. INTERMEDIATE_ PRIORITY was used for data that was not system critical

and required a small number of bytes to convey. An example of this might be data obtained from a redundant piece of hardware being flown. BACKGROUND_ PRIORITY was used to scavenge any telemetry bandwidth not being used by the higher priority levels.

FE-3 Data Redundancy

A high degree of redundancy was used for CRITICAL_CIRCLE_PRIORITY data. Nominally this data was always telemetered in a circle having, at minimum, three entries. Further, these circles were telemetered twice upon a measurement update. Thus, theoretically, at minimum a data measurement would appear six different times in the telemetered data. However, practically speaking it turned out this might vary somewhat due to the loading conditions on the priority queues in the telemetry packetizer. EMERGENCY_PRIORITY data was telemetered three times upon an occurrence of an "emergency" event. Each process in the system owned an emergency occurrence circle. These circles had eleven entries each. IMAGE_PRIORITY data was not redundant. Finally, INTERMEDIATE and BACKGROUND_ PRIORITY data had no pre-defined redundancy requirements.

The FE-3 Packet Structure

The processor(s) used for the FE-3 experiment employed 32 bit words. This had the following impact on the bit lengths of the packet header and trailer: *packet sync* - 64 bits (2 words); *data id* - 16 bits (2 bytes); *word count* - 16 bits (2 bytes); *packet number* - 10 bits; *request id* - 22 bits; *checksum* -32 bits (1 word). The high order bit of the request id was 0 for packets generated on processor A and 1 for packets generated on processor B. Each processor could generate over 2 million unique requests (2^{21}). Since almost no double precision numbers were ever telemetered, the choice of a 64 bit packet sync guaranteed a non "counterfeitable" packet identifier. The data id was chosen to be two bytes to insure maximum flexibility in availability of packet types (i.e. only 256 different types was deemed too limited). The sixteen bit word count easily accommodated the maximum packet size of 4K bytes. There was no limit placed on a minimum packet size; however, the maximum number of packets that could be used to deliver one telemetry request was 1024 (2^{10}). Finally, the checksum was selected to be 32 bits in light of the fact that the packetizer accessed data on 32 bit word boundaries. Figure 5 depicts the FE-3 packet structure.

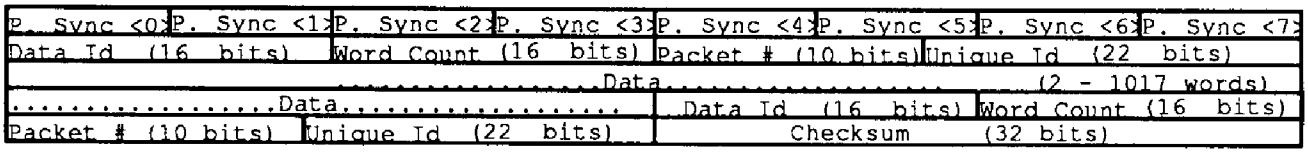

Figure 5. The FE-3 Packet Structure.

The FE-3 Telemetry Decoder

The FE-3 telemetry decoder (known as *tdfe*3) was comprised of four primary parts; specifically, a command line processor, a parser for the types file, a data extractor, and an output file processor. The essence of operation was the output file processor binding parsed data types onto the extracted packets of data. Decoding speed was expedited by making use of the so called packets to process (*ptp*) file. In most testing situations all packet types do not need to be decoded from the telemetry stream. When only a few packet types were extracted from the FE-3 telemetry stream, real time performance of the decoder was increased 7 fold. When all packet types were decoded real time performance was on the order of 1 Mbyte processed per 7 seconds (on a Mips RC 3230 class machine).

FE-3 Post Processing and Data Analysis

Most of the post processing applications developed for the FE-3 mission dealt with the reformatting/merging of information telemetered in many break out files into one file or the extraction of data from one telemetry file into a group of logically related files. Data analysis was performed with two utilities. The first utility known as *p.x11* was a keyboard driven tool which could read files generated by the *tdfe*3 utility. The program used a plotting language to define operations which could be performed on read in data. Upon reading in a break out file, the program would print out the header and place a numeral by each variable in the file. The user could then generate individual or over-plot data by referencing the numerals of interest in the plotting language. Many files could be read in by the utility making access to data relatively easy. The program had a number of built in mathematical functions which could be applied to data (addition, multiplication, differentiation, etc.).

The second data analysis tool (known as *TADA*[1] for Telemetry and Data Analysis) ran in the *IDL* graphics environment. It was a GUI tool. Upon execution, the program would prompt the user for the directory location of the data to be examined. After this

[1]The core *TADA* software was developed under contract by the Ball Aerospace Systems Group in Boulder Colorado. IDL is a trademark of Research Systems, Inc.

had been ascertained, the utility would search and display the filenames of all the break-out files in the directory. In order to access data in each file, the user simply had to click the mouse on the file of interest. This would then open another window displaying the data elements contained in the file. The user could then click on the specific data item. The item would then be recorded in a transaction list. Once all data items of interest had been selected, the user could then generate simple plots, or over-plots. Zooms could be accomplished via mouse pointing. The *TADA* utility also offered an interface for examining image data.

Conclusion

A data driven telemetry system offers many advantages. First and foremost it allows the content of the telemetry stream to be easily changed. This is particularly useful when at the later stages of experimental development it is discovered that telemetry of some other key data items has been overlooked. It is also convenient for debugging purposes and experiments where the telemetry content changes radically during a mission. Second, it allows all tools developed for post processing and data analysis to be re-used on other experiments. These concepts were practically demonstrated in the Brilliant Pebbles Flight Experiment Three (FE-3) experiment.

Acknowledgments

The author gratefully acknowledges the following individuals: D. R. Antelman, W. W. Brown, N. J. Cushing, C. Dorato, D. Eva, J. Fleming, B. Hedeline, B. Henderson, M. Henderson, J. E. Hoag, R. E. Lunow, J. A. Markevitch, J. W. Nally, G. G. Preckshot, J. J. Rhodes, D. Shih, F. Y. Shimamoto, E. J. Toy, T. J. Voss, B. A. Wilson, J.T. Weir, J. J. Yio, and J. Young.

This work was performed under the auspices of the U.S. Department of Energy by LLNL under contract number W-7405-Eng-48.